AIMMS

The Language Reference

AIMMS 4

June 28, 2023

AIMMS

The Language Reference

AIMMS

Marcel Roelofs
Johannes Bisschop

Email: info@aimms.com
WWW: www.aimms.com

About AIMMS

History

AIMMS was introduced as a new type of mathematical modeling tool in 1993—an integrated combination of a modeling language, a graphical user interface, and numerical solvers. AIMMS has proven to be one of the world's most advanced development environments for building optimization-based decision support applications and advanced planning systems. Today, it is used by leading companies in a wide range of industries in areas such as supply chain management, energy management, production planning, logistics, forestry planning, and risk-, revenue-, and asset- management. In addition, AIMMS is used by universities worldwide for courses in Operations Research and Optimization Modeling, as well as for research and graduation projects.

What is AIMMS*?*

AIMMS is far more than just another mathematical modeling language. True, the modeling language is state of the art for sure, but alongside this, AIMMS offers a number of advanced modeling concepts not found in other languages, as well as a full graphical user interface both for developers and end-users. AIMMS includes world-class solvers (and solver links) for linear, mixed-integer, and nonlinear programming such as BARON, CPLEX, CONOPT, GUROBI, KNITRO, PATH, SNOPT and XA, and can be readily extended to incorporate other advanced commercial solvers available on the market today. In addition, concepts as stochastic programming and robust optimization are available to include data uncertainty in your models.

Mastering AIMMS

Mastering AIMMS is straightforward since the language concepts will be intuitive to Operations Research (OR) professionals, and the point-and-click graphical interface is easy to use. AIMMS comes with comprehensive documentation, available electronically and in book form.

Types of AIMMS *applications*

AIMMS provides an ideal platform for creating advanced prototypes that are then easily transformed into operational end-user systems. Such systems can than be used either as

- stand-alone applications, or
- optimization components.

Stand-alone applications

Application developers and operations research experts use AIMMS to build complex and large scale optimization models and to create a graphical end-user interface around the model. AIMMS-based applications place the power of the most advanced mathematical modeling techniques directly into the hands of end-users, enabling them to rapidly improve the quality, service, profitability, and responsiveness of their operations.

Optimization components

Independent Software Vendors and OEMs use AIMMS to create complex and large scale optimization components that complement their applications and web services developed in languages such as C++, Java, .NET, or Excel. Applications built with AIMMS-based optimization components have a shorter time-to-market, are more robust and are richer in features than would be possible through direct programming alone.

AIMMS *users*

Companies using AIMMS include

- ABN AMRO
- Areva
- Bayer
- Bluescope Steel
- BP
- CST
- ExxonMobil
- Gaz de France
- Heineken
- Innovene
- Lufthansa
- Merck
- Owens Corning
- Perdigão
- Petrobras
- Philips
- PriceWaterhouseCoopers
- Reliance
- Repsol
- Shell
- Statoil
- Unilever

Universities using AIMMS include Budapest University of Technology, Carnegie Mellon University, George Mason University, Georgia Institute of Technology, Japan Advanced Institute of Science and Technology, London School of Economics, Nanyang Technological University, Rutgers University, Technical University of Eindhoven, Technische Universität Berlin, UIC Bioengineering, Universidade Federal do Rio de Janeiro, University of Groningen, University of Pittsburgh, University of Warsaw, and University of the West of England.

A more detailed list of AIMMS users and reference cases can be found on our website www.aimms.com.

Contents

Preface

Three AIMMS *books*

The printed AIMMS documentation consists of three books

- AIMMS—*The User's Guide*,
- AIMMS—*The Language Reference*, and
- AIMMS—*Optimization Modeling*.

The first two books emphasize different aspects in the use of the AIMMS system, while the third book is a general introduction to optimization modeling. All books can be used independently.

Available online

In addition to the printed versions, these books are also available on-line in the ADOBE Portable Document Format (PDF). Although new printed versions of the documentation will become available with every new functional AIMMS release, small additions to the system and small changes in its functionality in between functional releases are always directly reflected in the online documentation, but not necessarily in the printed material. Therefore, the online versions of the AIMMS books that come with a particular version of the system should be considered as the authoritative documentation describing the functionality regarding that particular AIMMS version.

Release notes

Which changes and bug fixes are included in particular AIMMS releases are described in the associated release notes.

What's new in AIMMS 4

From AIMMS 4.1 onwards, we will only publish this "What's New" section on our website. It can be found at the following location:

https://aimms.com/english/developers/downloads/product-information/new-features/

What is in the AIMMS documentation

The User's Guide

The AIMMS User's Guide provides a global overview of how to use the AIMMS system itself. It is aimed at application builders, and explores AIMMS' capabilities to help you create a model-based application in an easy and maintainable manner. The guide describes the various graphical tools that the AIMMS system offers for this task. It is divided into five parts.

- Part I—*Introduction to* AIMMS—what is AIMMS and how to use it.
- Part II—*Creating and Managing a Model*—how to create a new model in AIMMS or manage an existing model.
- Part III—*Creating an End-User Interface*—how to create an intuitive and interactive end-user interface around a working model formulation.
- Part IV—*Data Management*—how to work with cases and datasets.
- Part V—*Miscellaneous*—various other aspects of AIMMS which may be relevant when creating a model-based end-user application.

The Language Reference

The AIMMS Language Reference provides a complete description of the AIMMS modeling language, its underlying data structures and advanced language constructs. It is aimed at model builders only, and provides the ultimate reference to the model constructs that you can use to get the most out of your model formulations. The guide is divided into seven parts.

- Part I—*Preliminaries*—provides an introduction to, and overview of, the basic language concepts.
- Part II—*Nonprocedural Language Components*—describes AIMMS' basic data types, expressions, and evaluation structures.
- Part III—*Procedural Language Components*—describes AIMMS' capabilities to implement customized algorithms using various execution and flow control statements, as well as internal and external procedures and functions.
- Part IV—*Sparse Execution*—describes the fine details of the sparse execution engine underlying the AIMMS system.
- Part V—*Optimization Modeling Components*—describes the concepts of variables, constraints and mathematical programs required to specify an optimization model.
- Part VI—*Data Communication Components*—how to import and export data from various data sources, and create customized reports.
- Part VII—*Advanced Language Components*—describes various advanced language features, such as the use of units, modeling of time and communicating with the end-user.

Optimization Modeling

The book on optimization modeling provides not only an introduction to modeling but also a suite of worked examples. It is aimed at users who are new to modeling and those who have limited modeling experience. Both basic concepts and more advanced modeling techniques are discussed. The book is divided into five parts:

- Part I—*Introduction to Optimization Modeling*—covers what models are, where they come from, and how they are used.
- Part II—*General Optimization Modeling Tricks*—includes mathematical concepts and general modeling techniques.
- Part III—*Basic Optimization Modeling Applications*—builds on an understanding of general modeling principles and provides introductory application-specific examples of models and the modeling process.
- Part IV—*Intermediate Optimization Modeling Applications*—is similar to part III, but with examples that require more effort and analysis to construct the corresponding models.
- Part V—*Advanced Optimization Modeling Applications*—provides applications where mathematical concepts are required for the formulation and solution of the underlying models.

Documentation of deployment features

In addition to the three major AIMMS books, there are several separate documents describing various deployment features of the AIMMS software. They are:

- AIMMS—*The Function Reference*,
- AIMMS—*The COM Object User's Guide and Reference*,
- AIMMS—*The Excel Add-In User's Guide*, and
- AIMMS—*The Open Solver Interface User's Guide and Reference*.

These documents are only available in PDF format.

Help files

The AIMMS documentation is complemented with a number of help files that discuss the finer details of particular aspects of the AIMMS system. Help files are available to describe:

- the execution and solver options which you can set to globally influence the behavior of the AIMMS' execution engine,
- the finer details of working with the graphical modeling tools, and
- a complete description of the properties of end-user screens and the graphical data objects which you can use to influence the behavior and appearance of an end-user interface built around your model.

The AIMMS help files are both available as Windows help files, as well as in PDF format.

AIMMS tutorials

Two tutorials on AIMMS in PDF format provide you with some initial working knowledge of the system and its language. One tutorial is intended for beginning users, while the other is aimed at professional users of AIMMS.

Searching the documentation

As the entire AIMMS documentation is available in PDF format, you can use the search functionality of Acrobat Reader to search through all AIMMS documentation for the information you are looking for.

AIMMS model library

AIMMS comes with an extensive model library, which contains a variety of examples to illustrate simple and advanced applications containing particular aspects of both the language and the graphical user interface. You can find the AIMMS model library in the Examples directory in the AIMMS installation directory. The Examples directory also contains an AIMMS project providing an index to all examples, which you can use to search for examples that illustrate specific aspects of AIMMS.

What is in the Language Reference

Preliminaries

Part I of the Language Reference introduces and illustrates the basic concepts of the AIMMS language.

- Chapter 1—*Introduction to the* AIMMS *language*—provides you with a quick overview of AIMMS' modeling capabilities through a simple, and completely worked out example model.
- Chapter 2—*Language preliminaries*—globally describes the basic structure of an AIMMS model, the available data types and execution statements.

Nonprocedural language components

Part II introduces the fundamental concepts of sets and multidimensional parameters, and discusses the expressions and evaluation mechanisms available for these data types.

- Chapter 3—*Set declaration*—discusses the declaration and attributes of index sets.
- Chapter 4—*Parameter declaration*—describes the declaration and available attributes of scalar and multidimensional parameters which can be used to store and manipulate data.
- Chapter 5—*Set, set element and string expressions*—provides a complete overview of all expressions which evaluate to either a set, a set element or a string.
- Chapter 6—*Numerical and logical expressions*—describes all expressions which evaluate to a numerical or logical value, and also explains the concept of macro expansion in AIMMS.

- Chapter 7—*Execution of nonprocedural components*—describes the dependency and automatic execution structure of the system of functional relationships formed by all defined sets and parameters.

Procedural language components

Part III focuses on the procedural aspects of the AIMMS language which allow you to implement you own algorithms, seamlessly making use of the advanced built-in functionality already provided by AIMMS.

- Chapter 8—*Execution statements*—provides a complete overview of all assignment and flow control statements in AIMMS.
- Chapter 9—*Index binding*—specifies the precise rules for the fundamental concept of index binding underlying AIMMS execution engine.
- Chapter 10—*Internal procedures and functions*—explains how to declare and call internal AIMMS procedures and functions.
- Chapter 11—*External procedures and functions*—explains how functions and procedures in an external DLL can be linked to and called from within an existing AIMMS application.

Sparse execution

Part IV of the reference guide tries to make you aware of the differences between a dense versus a sparse execution engine (as used by AIMMS). It provides valuable insight into the inner workings of AIMMS and may help to implement large-scale modeling applications in a correct and efficient manner.

- Chapter 12—*The* AIMMS *sparse execution engine*—provides you with a basic insight into the inner workings AIMMS sparse execution engine, and provides a number of convenience operators to modify the semantics of some operators.
- Chapter 13—*Execution efficiency cookbook*—discusses various techniques that you may apply to find and address performance issues in your AIMMS models.

Optimization modeling components

Part V of the reference guide discusses all concepts offered by AIMMS for specifying and solving optimization models.

- Chapter 14—*Variable and constraint declaration*—discusses the declaration and attributes of variables and constraints.
- Chapter 15—*Solving mathematical programs*—describes the steps necessary for specifying and solving an optimization program in AIMMS.
- Chapter 24—*Node and arc declaration*—discusses the declaration and attributes of node and arc types available in AIMMS to specify single commodity network flow models.
- Chapter 17—*Advanced methods for nonlinear programs*—discusses the multistart algorithm and nonlinear presolver available in AIMMS for nonlinear models.
- Chapter 23—*Mixed complementarity problems*—describes the declaration and attributes of complementarity variables, which can be used to specify mixed complementarity and MPCC models in AIMMS.

- Chapter 19—*Stochastic programming*—discusses the facilities in AIMMS to generate stochastic models and associated scenario trees for existing deterministic model formulations.
- Chapter 20—*Robust optimization*—introduces the facilities in AIMMS to generate and solve robust optimization models for existing deterministic model formulations.
- Chapter 16—*Implementing advanced algorithms for mathematical programs*—describes a library of procedures which allow you to implement advanced algorithms for solving linear and mixed-integer linear programming models.
- Chapter 18—AIMMS *Outer Approximation Algorithm for MINLP*—introduces an open approach to solving MINLP models using the well-known outer approximation algorithm.

Data communication components

Part VI introduces the mechanisms provided by AIMMS to import data from files and databases, as well as its capabilities to export data and produce standardized or customized text reports.

- Chapter 25—*Data initialization, verification and control*—describes your options to initialize the identifiers associated with an AIMMS model. It also introduces the concept of assertions which can be used to verify the consistency of data, as well as a number of data control statements which can help you to keep the data in a consistent state.
- Chapter 26—*The READ and WRITE statements*—describes the basic mechanism offered by AIMMS for data transfer with various data sources.
- Chapter 27—*Communicating with databases*—discusses the specific aspects of setting up a link between AIMMS and a database.
- Chapter 28—*Format of text data files*—presents the various data formats offered by AIMMS for initializing a model through a number of text data files.
- Chapter 29—*Reading and Writing Spreadsheet Data*—provides you with an overview of AIMMS' capabilities to exchange data with Excel or with OpenOffice Calc workbooks.
- Chapter 30—*Reading and Writing XML Data*—discusses AIMMS' facilities to read and write XML data from within AIMMS.
- Chapter 31—*Text reports and listing*—describes the statements and formatting options available for producing standardized and customized text reports.

Advanced language components

Part VII of the reference guide introduces a number of advanced features available in AIMMS both in the area of modeling and communication with external applications.

- Chapter 32—*Units of measurement*—discusses the declaration and use of units and unit conventions in an AIMMS model both for checking the

consistency of a model formulation, scaling of mathematical programs and display of data in the interface and reports.

- Chapter 33—*Time-based modeling*—describes the advanced concepts in AIMMS to deal with time-dependent data and models in a flexible and easy manner.
- Chapter 34—*The* AIMMS *programming interface*—offers a complete description of the application programming interface (API) which can be used to access AIMMS data structures and call AIMMS procedures from within an external DLL or application.
- Chapter 35—*Model structure and modules*—discusses the organizational data structures such as the main model, model sections and modules, which can be used to supply the model with a logical structure, as well as library modules, which facilitate model development by multiple developers.

The authors

Marcel Roelofs

Marcel Roelofs received his Ph.D. in Applied Mathematics from the Technical University of Twente in 1993 on the application of Computer Algebra in Mathematical Physics. From 1993 to 1995 he worked as a post-doc at the Centre for Mathematics and Computer Science (CWI) in Amsterdam in the area of Computer Algebra, and had a part-time position at the Research Institute for the Application of Computer Algebra. In 1995 he accepted his current position as CTO of AIMMS B.V. His main responsibilities are the design and documentation of the AIMMS language and user interface.

Johannes Bisschop

Johannes Bisschop received his Ph.D. in Mathematical Sciences from the Johns Hopkins University in Baltimore USA in 1974. From 1975 to 1980 he worked as a Researcher in the Development Research Center of the World Bank in Washington DC, USA. In 1980 he returned to The Netherlands and accepted a position as a Research Mathematician at Shell Research in Amsterdam. After some years he also accepted a second part-time position as a full professor in the Applied Mathematics Department at the Technical University of Twente. From 1989 to 2003 he combined his part-time position at the University with managing Paragon Decision Technology B.V. and the continuing development of AIMMS. From 2003 to 2005 he held the position of president of Paragon Decision Technology B.V. His main interests are in the areas of computational optimization and modeling.

Other contributors to AIMMS

In addition to the main authors, various current and former employees of AIMMS B.V. (formerly known as Paragon Decision Technology B.V.) and external consultants have made a contribution to the AIMMS documentation. They are (in alphabetical order):

- Pim Beers
- John Boers
- Peter Bonsma
- Mischa Bronstring
- Ximena Cerda Salzmann
- Michelle Chamalaun
- Horia Constantin
- Guido Diepen
- Robert Entriken
- Floor Goddijn
- Thorsten Gragert
- Koos Heerink
- Nico van den Hijligenberg
- Marcel Hunting
- Roel Janssen
- Gertjan Kloosterman
- Joris Koster
- Chris Kuip
- Gertjan de Lange
- Ovidiu Listes
- Peter Nieuwesteeg
- Franco Peschiera
- Bianca Rosegaar
- Diego Serrano
- Giles Stacey
- Richard Stegeman
- Selvy Suwanto
- Jacques de Swart
- Martine Uyterlinde

Part I

Preliminaries

Chapter 1

Introduction to the AIMMS language

Example makes a good starting point

This chapter discusses a simple but complete modeling example containing the most common components of a typical AIMMS application. The aim is to give a quick feel for the language, and to assist you to form a mental picture of its functionality.

Familiarity with algebraic notation

It is assumed that you are familiar with some basic algebraic notation. It is important that you understand the notions of "summation," "simultaneous equations in many variables (unknowns)," and "minimizing or maximizing an objective function, subject to constraints." If you are not acquainted with these notions, refer to the book AIMMS—*Optimization Modeling*.

What to expect in this chapter

This chapter uses a simple depot location problem to introduce the basic AIMMS concepts necessary to formulate and solve the model. The task consists of the following steps.

- Section 1.1 describes the depot location problem, introduces the set notation, and illustrates how sets can be used to declare multidimensional identifiers useful for modeling the problem in AIMMS.
- Section 1.2 discusses the formulation of a mathematical program that can be used to compute the optimal solution of the problem.
- Section 1.3 briefly discusses data initialization, and explains how data can be entered.
- Section 1.4 illustrates how you can use flow control statements in AIMMS to formulate an algorithm for solving your problems in advanced ways.
- Section 1.5 discusses issues to consider when working with more complex models.

1.1 The depot location problem

The modeling process

In translating any real-life problem into a valid AIMMS optimization model (referred to as a mathematical program) several conceptual steps are required. They are:

- describe the input and output data using sets and indexed identifiers,
- specify the mathematical program,

- specify procedures for data pre- and post-processing,
- initialize the input data from files and databases,
- solve the mathematical program, and
- display the results (or write them back to a database).

Problem description

The example in this chapter is based on a simple depot location problem which can be summarized as follows.

> *Consider the distribution of a single product from one or more depots to multiple customers. The objective is to select depots from a predefined set of possible depots (each with a given capacity) such that*
> - *the demand of each customer is met,*
> - *the capacity of each selected depot is not exceeded, and*
> - *the total cost for both depot rental and transport to the customers is minimized.*

Use of sets

In the above problem you can see that there are two entities that determine the size of the problem: depots and customers. With these entities a number of instances are associated, e.g. a particular instance of a depot could be `'Amsterdam'`. The precise collection of instances, however, may differ from run to run. Therefore, when translating the problem into a symbolic model it is customary and desirable not to make any explicit reference to individual instances. Such high-level model specification can be accomplished through the use of *sets*, each with an associated *index* for referencing arbitrary elements in that set.

Initial set declarations

The following set declarations in AIMMS introduce the two sets `Depots` and `Customers` with indices `d` and `c`, respectively. AIMMS has a convenient graphical model editor to create your model. It allows you to enter all model input using graphical forms. However, in the interest of compactness we will use a textual representation for declarations that closely resembles the contents of a graphical form throughout this manual.

```
Set Depots {
    Index :  d;
}
Set Customers{
    Index :  c;
}
```

Parameters for input data

In most models there is input data that can be naturally associated with a particular element or tuple of elements in a set. In AIMMS, such data is stored in Parameters. A good example in the depot location problem is the quantity `Distance`, which can be defined as the distance between depot `d` and customer `c`. To define `Distance` a index tuple (`d`,`c`) is required and it is referred to as the associated `IndexDomain` of this quantity.

Example

In AIMMS, the identifier Distance is viewed as a Parameter (a known quantity), and can be declared as follows.

```
Parameter Distance {
    Index :  (d,c);
}
```

In this example the identifier Distance is referred to as an ndexed identifier, because it has a nonempty index domain.

Scalar data

Not all identifiers in a model need to be indexed. The following declarations illustrate two scalar parameters which are used later.

```
Parameter MaxDeliveryDistance;
Parameter UnitTransportRate;
```

Restricting permitted routes

For real-life applications the collection of all possible routes (d,c) may be huge. In practice, routes (d,c) for which the distance Distance(d,c) is big, will never become a part of the solution. It, therefore, makes sense to exclude such routes (d,c) from the entire solution process altogether. We can do this by computing a set of PermittedRoutes which we will use throughout the sequel of the example.

Example

In AIMMS, the relation PermittedRoutes can be declared as follows.

```
Set PermittedRoutes {
    SubsetOf     :  (Depots, Customers);
    Definition   :  {
        { (d,c) | Distance(d,c) <= MaxDeliveryDistance }
    }
}
```

Explanation

In the SubsetOf attribute of the above declaration it is indicated that the set PermittedRoutes is a subset of the Cartesian product of the simple sets Depots and Customers. The Definition attribute globally defines the set PermittedRoutes as the set of those tuples (d, c) for which the associated Distance(d,c) does not exceed the value of the scalar parameter MaxDeliveryDistance. AIMMS will assure that such a global relationship is valid at any time during the execution of the model. Note that the set notation in the Definition attribute resembles the standard set notation found in mathematical literature.

Applying domain restrictions

Now that we have restricted the collection of permitted routes, we can use the relation PermittedRoutes throughout the model to restrict the domain of identifiers declared over (d,c) to only hold data for permitted routes (d,c).

Example

In AIMMS, the parameter UnitTransportCost can be declared as follows.

```
Parameter UnitTransportCost {
    IndexDomain  :  (d,c) in PermittedRoutes;
    Definition   :  UnitTransportRate * Distance(d,c);
}
```

This parameter is defined through a simple formula. Once an identifier has its own definition, AIMMS will not allow you to make an assignment to this identifier anywhere else in your model text.

Effects of domain restriction

As an effect of applying a domain restriction to the parameter UnitTransportCost, any reference to UnitTransportCost(d,c) for tuples (d,c) outside the set PermittedRoutes is not defined, and AIMMS will evaluate this quantity to 0. In addition, AIMMS will use the domain restriction in its GUI, and will not allow you to enter numerical values of UnitTransportCost(d,c) outside of its domain.

Additional parameter declarations

To further define the depot location problem the following parameters are required:

- the fixed rental charge for every depot d,
- the available capacity of every depot d, and
- the product demand of every customer c.

The AIMMS declarations are as follows.

```
Parameter DepotRentalCost {
    IndexDomain  :  d;
}
Parameter DepotCapacity {
    IndexDomain  :  d;
}
Parameter CustomerDemand {
    IndexDomain  :  c;
}
```

1.2 Formulation of the mathematical program

Constraint-oriented modeling

In programming languages like C it is customary to solve a particular problem through the explicit specification of an algorithm to compute the solution. In AIMMS, however, it is sufficient to specify only the Constraints which have to be satisfied by the solution. Based on these constraints AIMMS generates the input to a specialized numerical solver, which in turn determines the (optimal) solution satisfying the constraints.

Variables as unknowns

In constraint-oriented modeling the unknown quantities to be determined are referred to as `variables`. Like parameters, these variables can either be scalar or indexed, and their values can be restricted in various ways. In the depot location problem it is necessary to solve for two groups of variables.

- There is one variable for each depot `d` to indicate whether that depot is to be selected from the available depots.
- There is another variable for each permitted route `(d,c)` representing the level of transport on it.

Example

In AIMMS, the variables described above can be declared as follows.

```
Variable DepotSelected {
    IndexDomain   :  d;
    Range         :  binary;
}
Variable Transport {
    IndexDomain   :  (d,c) in PermittedRoutes;
    Range         :  nonnegative;
}
```

The Range attribute

For unknown variables it is customary to specify their range of values. Various predefined ranges are available, but you can also specify your own choice of lower and upper bounds for each variable. In this example only predefined ranges have been used. The predefined range `binary` indicates that the variable can only assume the values 0 and 1, while the range `nonnegative` indicates that the value of the corresponding variable must lie in the continuous interval $[0, \infty)$.

Constraints description

As indicated in the problem description in Section 1.1 a solution to the depot location problem must satisfy two constraints:

- the demand of each customer must be met, and
- the capacity of each selected depot must not be exceeded.

Example

In AIMMS, these two constraints can be formulated as follows.

```
Constraint CustomerDemandRestriction {
    IndexDomain  :  c;
    Definition   :  Sum[ d, Transport(d,c) ] >= CustomerDemand(c);
}
Constraint DepotCapacityRestriction {
    IndexDomain  :  d;
    Definition   :  Sum[ c, Transport(d,c) ] <= DepotCapacity(d)*DepotSelected(d);
}
```

Satisfying demand

The constraint CustomerDemandRestriction(c) specifies that for every customer c the sum of transports from every possible depot d to this particular customer must exceed his demand. Note that the Sum operator behaves as the standard summation operator $\sum$ found in mathematical literature. In AIMMS the domain of the summation must be specified as the first argument of the Sum operator, while the second argument is the expression to be accumulated.

Proper domain

At first glance, it may seem that the (indexed) summation of the quantities Transport(d,c) takes place over all tuples (d,c). This is not the case. The underlying reason is that the variable Transport has been declared with the index domain (d,c) in PermittedRoutes. As a result, the transport from a depot d to a customer c not in the set PermittedRoutes is not considered (i.e. not generated) by AIMMS. This implies that transport to c only accumulates along permitted routes.

Satisfying capacity

The interpretation of the constraint DepotCapacityRestriction(d) is twofold.

- Whenever DepotSelected(d) assumes the value 1 (the depot is selected), the sum of transports leaving depot d along permitted routes may not exceed the capacity of depot d.
- Whenever DepotSelected(d) assumes the value 0 (the depot is not selected), the sum of transports leaving depot d must be less than or equal to 0. Because the range of all transports has been declared nonnegative, this constraint causes each individual transport from a nonselected depot to be 0 as expected.

The objective function

The objective in the depot location problem is to minimize the total cost resulting from the rental charges of the selected depots together with the cost of all transports taking place. In AIMMS, this objective function can be declared as follows.

```
Variable TotalCost {
    Definition : {
        Sum[ d, DepotRentalCost(d)*DepotSelected(d) ] +
        Sum[ (d,c), UnitTransportCost(d,c)*Transport(d,c) ];
    }
}
```

Defined variables

The variable TotalCost is an example of a defined variable. Such a variable will not only give rise to the introduction of an unknown, but will also cause AIMMS to introduce an additional constraint in which this unknown is set equal to its definition. Like in the summation in the constraint DepotCapacityRestriction, AIMMS will only consider the tuples (d,c) in PermittedRoutes in the definition of the variable TotalCost, without you having to (re-)specify this restriction again.

The mathematical program

Using the above, it is now possible to specify a mathematical program to find an optimal solution of the depot location problem. In AIMMS, this can be declared as follows.

```
MathematicalProgram DepotLocationDetermination {
    Objective    :  TotalCost;
    Direction    :  minimizing;
    Constraints  :  AllConstraints;
    Variables    :  AllVariables;
    Type         :  mip;
}
```

Explanation

The declaration of DepotLocationDetermination specifies a mathematical program in which the defined variable TotalCost serves as the objective function to be minimized. All previously declared constraints and variables are to be part of this mathematical program. In more advanced applications where there are multiple mathematical programs it may be necessary to reference subsets of constraints and variables. The Type attribute specifies that the mathematical program is a mixed integer program (mip). This reflects the fact that the variable DepotSelected(d) is a binary variable, and must attain either the value 0 or 1.

Solving the mathematical program

After providing all input data (see Section 1.3) the mathematical program can be solved using the following simple execution statement.

```
Solve DepotLocationDetermination ;
```

A SOLVE statement can only be called inside a procedure in your model. An example of such a procedure is provided in Section 1.4.

1.3 Data initialization

Separation of model and data

In the previous section the entire depot location model was specified without any reference

- to specific elements in the sets Depots and Customers, or
- to specific values of parameters defined over such elements.

As a result of this clear separation of model and data values, the model can easily be run for different data sets.

Data sources

A data set can come from various sources. In AIMMS there are six sources you might consider for your application. They are:

- commercial databases,
- text data files,
- AIMMS case files,
- internal procedures,

- external procedures, or
- the AIMMS graphical user interface (GUI).

These data sources are self-explanatory with perhaps the AIMMS case files as an exception. AIMMS case files are obtained by using the case management facilities of AIMMS to store data values from previous runs of your model.

A simple data set in text format

The following fictitious data set is provided in the form of an text data file. It illustrates the basic constructs available for providing data in text format. In this file, assignments are made using the ':=' operator and the keywords of DATA TABLE and COMPOSITE TABLE announce the table format. The exclamation mark denotes a comment line.

```
Depots    := DATA { Amsterdam, Rotterdam };
Customers := DATA { Shell, Philips, Heineken, Unilever };

COMPOSITE TABLE
    d        DepotRentalCost    DepotCapacity
! ---------    ---------------    -------------
  Amsterdam        25550              12500
  Rotterdam        31200              14000
;

COMPOSITE TABLE
    c        CustomerDemand
! ---------    --------------
  Shell            10000
  Philips           5000
  Heineken          3000
  Unilever          5000
;

Distance(d,c) := DATA TABLE
                Shell   Philips   Heineken   Unilever
!               -----   -------   --------   --------
Amsterdam       100       200        50         150
Rotterdam        75       100        50          75
;

UnitTransportRate   := 1.25 ;
MaxDeliveryDistance := 125  ;
```

Reading in the data

Assuming that the text data file specified above was named "initial.dat", then its data can easily be read using the following READ statement.

```
read from file "initial.dat" ;
```

Such READ statements are typically placed in the predefined procedure Main-Initialization. This procedure is automatically executed at the beginning of every session immediately following the compilation of your model source.

Automatic initialization

When AIMMS encounters any reference to a set or parameter with its own definition inside a procedure, AIMMS will automatically compute its value on the basis of its definition. When used inside the procedure `MainInitialization`, this form of data initialization can be viewed as yet another data source in addition to the six data sources mentioned at the beginning of this section.

1.4 An advanced model extension

This section

In this section a single procedure is developed to illustrate the use of execution control structures in AIMMS. It demonstrates a customized solution approach to solve the depot location problem subject to fluctuations in demand. Understanding the precise algorithm described in this section requires more mathematical background than was required for the previous sections. However, even without this background the examples in this section may provide you with a basic understanding of the capabilities of AIMMS to manipulate its data and control the flow of execution.

Finding a robust solution

The mathematical program developed in Section 1.1 does not take into consideration any fluctuations in customer demand. Selecting the depots on the basis of a single demand scenario may result in insufficient capacity under changing demand requirements. While there are several techniques to determine a solution that remains robust under fluctuations in demand, we will consider here a customized solution approach for illustrative purposes.

Algorithm in words

The overall structure of the algorithm can be captured as follows.

- During each major iteration, the algorithm adds a single new depot to a set of already permanently selected depots.
- To determine a new depot, the algorithm solves the depot location model for a fixed number of scenarios sampled from normal demand distributions. During these runs, the variable `DepotSelected(d)` is fixed to 1 for each depot `d` in the set of already permanently selected depots.
- The (nonpermanent) depot for which the highest selection frequency was observed in the previous step is added to the set of permanently selected depots.
- The algorithm terminates when there are no more depots to be selected or when the total capacity of all permanently selected depots first exceeds the average total demand incremented with the observed standard deviation in the randomly selected total demand.

Additional identifiers

In addition to all previously declared identifiers the following algorithmic identifiers will also be needed:

- the set `SelectedDepots`, a subset of the set `Depots`, holding the already permanently selected depots, as well as

- the parameters AverageDemand(c), DemandDeviation(c), TotalAverageDemand, NrOfTrials, DepotSelectionCount(d), CapacityOfSelectedDepots, TotalSquaredDemandDifference and TotalDemandDeviation.

The meaning of these identifiers is either self-explanatory or will become clear when you study the further specification of the algorithm.

Outline of algorithm

At the highest level you may view the algorithm described above as a single initialization block followed by a WHILE statement containing a reference to two additional execution blocks. The corresponding outline is as follows.

```
<<Initialize algorithmic parameters>>

while ( Card(SelectedDepots) < Card(Depots) and
        CapacityOfSelectedDepots < TotalAverageDemand + TotalDemandDeviation ) do
    <<Determine depot frequencies prior to selecting a new depot>>
    <<Select a new depot and update algorithmic parameters>>
endwhile;
```

The AIMMS function Card determines the cardinality of a set, that is the number of elements in the set.

Initializing model parameters

The initialization blocks consists of assignment statements to give each relevant set and parameter its initial value. Note that the assignments indexed with d will be executed for every depot in the Depots, and no explicit FOR statement is required.

```
TotalAverageDemand           := Sum[ c, AverageDemand(c) ];

SelectedDepots               := { };
DepotSelectionCount(d)       := 0;
CapacityOfSelectedDepots     := 0;

TotalDemandDeviation         := 0;
TotalSquaredDemandDifference := 0;

DepotSelected.NonVar(d)      := 0;
```

Explanation

With the exception of TotalAverageDemand, all identifiers are assigned their default value 0 or empty. This is superfluous the first time the algorithm is called during a session, but is required for each subsequent call. The value of global identifiers such as NrOfTrials, AverageDemand(c) and DemandDeviation(c) must be set prior to calling the algorithm.

The .NonVar suffix

The suffix .NonVar indicates a nonvariable status. Whenever the suffix DepotSelected.NonVar(d) is nonzero for a particular d, the corresponding variable DepotSelected(d) is considered to be a parameter (and thus fixed inside a mathematical program).

Determining depot frequencies

The AIMMS program that determines the depot frequencies prior to selecting a new depot consists of just five statements.

```
while ( LoopCount <= NrOfTrials ) do
    CustomerDemand(c) := Normal(AverageDemand(c), DemandDeviation(c));

    Solve DepotLocationDetermination;

    DepotSelectionCount(d | DepotSelected(d)) += 1;

    TotalSquaredDemandDifference += Sum[ c, (CustomerDemand(c) - AverageDemand(c))^2 ];
endwhile;
```

Explanation

Inside the WHILE statement the following steps are executed.

- Determine a demand scenario.
- Solve the corresponding mathematical program.
- Increment the depot selection frequency accordingly.
- Register squared deviations from the average for total demand.

Functions used

The operator LoopCount is predefined in AIMMS, and counts the number of the current iteration in any of AIMMS' loop statements. Its initial value is 1. The function Normal is also predefined, and generates a number from the normal distribution with known mean (the first argument) and known standard deviation (the second argument). The operator += increments the identifier on the left of it with the amount on the right. The operator ^ represents exponentiation.

Selecting a new depot

The AIMMS program to select a new depot and update the relevant algorithmic parameters also consists of just five statements.

```
SelectedDepots           += ArgMax[ d | not d in SelectedDepots,
                                        DepotSelectionCount(d) ];
CapacityOfSelectedDepots := Sum[ d in SelectedDepots, DepotCapacity(d) ];

TotalDemandDeviation     := Sqrt( TotalSquaredDemandDifference ) /
                            (Card(SelectedDepots)*NrOfTrials) ;

DepotSelected(d in SelectedDepots)        := 1;
DepotSelected.NonVar(d in SelectedDepots) := 1;
```

Explanation

In the above AIMMS program the following steps are executed.

- Determine the not already permanently selected depot with the highest frequency, and increment the set of permanently selected depots accordingly.
- Register the new current total capacity as the sum of all capacities of depots that have been permanently selected.
- Register the new value of the estimated standard deviation in total demand.

- Assign 1 to all permanently selected depots, and fix their nonvariable status accordingly.

Functions used

The iterative operator `ArgMax` considers all relevant depots from its first argument, and takes as its value that depot for which the corresponding second arguments is maximal. The AIMMS function `Sqrt` denotes the well-known square root operation.

1.5 General modeling tips

From beginner to advanced

The previous sections introduced you to optimization modeling in AIMMS. In such a small application, the model structure is quite transparent and the formulation in AIMMS is straightforward. This section discusses issues to consider when your model is larger and more complex.

Separation of model and data

The AIMMS language is geared to strictly separate between model formulation and the supply of its data. While this may seem unnatural at first (when your models are still small), there are several major advantages in using this approach.

- By formulating the definitions and assignments associated with your problem in a completely symbolic form (i.e. without any reference to numbers or particular set elements) the intention of the expressions present in your model is more apparent. This is especially true when you have chosen clear and descriptive names for all the identifiers in your model.
- With the separation of model and data it becomes possible to run your model with several data sets. Such data sets may describe completely different problem topologies, all of which is perfectly fine as long as your model formulation has been set up transparently.
- Keeping your model free from explicit references to numbers or particular set elements improves maintainability considerably. Explicit data references inside assignment statements and constraints are essentially undocumented, and therefore subsequent changes in values are error-prone.

Intellectual challenge

Translating a real-life problem into a working modeling application is not always an easy task. In fact, finding a formulation or implementing a solution method that works in all cases is quite often a demanding (but also a very satisfying) intellectual challenge.

Levels of abstraction

Setting up a transparent model involves incorporating an appropriate level of abstraction. For example, when modeling a specific plant with two production units and two products, you might be tempted to introduce just four dedicated identifiers to store the individual production values. Instead, it is better to introduce a single generic identifier for storing production values for all units and all products. By doing so, you incorporate genericity in your application and it will be possible to re-use the application at a later date for a different plant with minimum reformulation.

Finding the proper level

Finding the proper level of abstraction is not always obvious but it becomes easier as your modeling experience increases. In general, it is a good strategy to re-think the consequences—with an eye on the extensibility of your application—before implementing the most straightforward data structures. In most cases the time spent finding a more generic structure is paid back, because the better structure helps you to formulate and extend the model in a clear and structured way.

From small to large-scale

Transforming a small working demo application into a large scale real-life application may result in problems if care is not taken to specify variables and constraints in an accurate manner. In a small model, there is usually no runtime penalty to poorly specified mathematical programs. In contrast, when working with large multidimensional data sets, a poor formulation of a mathematical program can easily cause that

- the available memory resources are exhausted, or
- runtime requirements are not met.

Under these conditions, the physical constraints should be reassessed and appropriate domains, parameter definitions and constraints added as outlined below.

Formulating proper domains of definition

For large applications you should always ask the following questions.

- Have you adequately constrained the domains of high-dimensional identifiers? Often by reassessing the physical situation the domain range can be further reduced. Usually such domain restrictions can be expressed through logical conditions referring to other (input) identifiers.
- Can you predict, for whatever reason, that some index combinations are very unlikely to appear in the solution of a mathematical program, even though they should be allowed formally? If so, you might experiment with omitting such combinations from their respective domains of definition, and see how this domain reduction reduces the size of the mathematical program and affects its solution.

As a result of carefully re-designing index domains you may find that your model no longer exhausts available memory resources and runs in an acceptable amount of time.

Example

In the depot location problem discussed in this chapter, the domain of the variable `Transport` has already restricted to the set of allowed `PermittedRoutes`, as computed on page 4. Thus, the mathematical program will never consider transports on a route that is not desirable. Without this restriction, the mathematical program would consider the transports from *every* depot `d` to *every* customer `c`. The latter may cause the mathematical program size to explode, when the number of depots and customers become large.

Reformulation of algorithm

Finally, you may run into mathematical programs where the runtime of a solution method does not scale well even after careful domain definition. In this case, it may be necessary to reformulate the problem entirely. One approach may be to decompose the original mathematical program into subprograms, and use these together with a customized sequential solution method to obtain acceptable solutions. You can find pointers to many of such decomposition methods in the AIMMS Modeling Guide.

Chapter 2

Language Preliminaries

This reference guide

This language reference describes the syntax and semantics of the AIMMS language. It is recommended that you read the chapters in sequence, but this is not essential. Both the contents and index are useful for locating the specifics of any topic. Illustrative examples throughout the text will give you a quick understanding of each subject.

2.1 Managing your model

Models in AIMMS

AIMMS is a language for the specification and implementation of multidimensional modeling applications. An AIMMS *model* consists of

- a *declarative part* which specifies all sets and multidimensional identifiers defined over these sets, together with the fixed functional relationships defined over these identifiers,
- an *algorithmic part* consisting of one or more procedures which describes the sequence of statements that transform the input data of a model into the output data, and
- a *utility part* consisting of additional identifier declarations and procedures to support a graphical end-user interface for your application.

Optimization included . . .

The declarative part of a model in AIMMS may include the specification of optimization problems containing simultaneous systems of equations. In the algorithmic part you can call a special SOLVE statement to translate such optimization problems to a format suitable for a linear or nonlinear solver.

. . . but not necessary

Although optimization modeling will be an important part of most AIMMS applications, AIMMS is also a convenient tool for other types of applications.

- The purely symbolic representation of set and parameter definitions with their automatic dependency structure provides spreadsheet-like functionality but with the benefit of much greater maintainability.
- Because of its simple data structures and power of expression, AIMMS lends itself for use as a rapid prototyping language.

Interfacing with the GUI

Although it is possible to create a simple end-user interface showing your model's data in the form of tables and graphs, a much more advanced user interface is possible by exploiting the capabilities of the AIMMS interface builder. Mostly, this involves the introduction of various additional sets and parameters in your model, as well as the implementation of additional procedures to perform special interface-related tasks.

The model tree

Modeling in AIMMS is centered around a graphical tool called the *model explorer*. In the model explorer the contents and structure of your model is presented in a tree-like fashion, which is also referred to as the *model tree*. The model tree can contain various types of nodes, each with their own use. They are:

- *structuring* sections, which you can use to partition the declarations and procedures that are part of your model into logical groups,
- *declaration* sections which contain the *declarations* of the global identifiers (like sets, parameters and variables) in your model, and
- *procedures* and *functions* which contain the statements that describe the algorithmic part of your application.

Creating new models

When you start a new model AIMMS will automatically create a skeleton model tree which is suitable for small applications. The skeleton contains the following nodes:

- a single *declaration section* where you can store the declarations used in your model,
- the predefined procedure `MainInitialization` which is called directly after compiling your model and can be used to initialize your model,
- the predefined procedure `MainExecution` where you can put all the statements necessary to execute the algorithmic part of your application, and
- the predefined procedure `MainTermination` which is called just prior to leaving AIMMS.

Changing the skeleton

Whenever the number of declarations in your model grows too large to be easily managed within a single declaration section, or when you want to divide the execution associated with your application into several procedures, you are free to change the skeleton model tree created by AIMMS. You can group particular declarations into separate declaration sections with a meaningful name, and introduce new procedures and functions.

Structuring your model

When you feel that particular groups of declarations, procedures and functions belong together in a logical manner, you are encouraged to create a new structuring section with a descriptive name within the model tree, and store the associated model components underneath it. When your application grows in size, a clear hierarchical structure of all the information stored will help tremendously to find your way within your application easily and quickly.

Storage on disk

The contents of a model is stored in one or more text files with the "`.ams`" (AIMMS model source) extension. By default the entire model is stored in a single file, but for each structural section you can indicate that you want to store the subtree underneath it in a separate source file. This is especially useful when particular parts of your application are shared with other AIMMS applications, or when there are multiple developers, each responsible for a particular part of the model.

Character encoding used in text files

A text is a sequence of characters. A text file contains such a text whereby the characters are encoded into numbers. The mapping between these characters in a text and these numbers in a file is called an encoding. The historically prevailing encoding is ASCII which defines the encoding for some control characters, the English alphabet, digits, and frequently used punctuation characters for the values 1 .. 127. However, as characters are stored in bytes, the values 128 .. 255 are free and these are used at different locales for different purposes. These locale specific extensions of ASCII are also called code pages. As a consequence, the characters displayed of an ASCII file containing some of the numbers 128 .. 255, depend on the active code page selected. The problem here is that the contents of ASCII files were ambiguous when the code page to be used was not known (see also `en.wikipedia.org/wiki/Codepage`). In order to circumvent this problem, the Unicode consortium enumerated all characters into more than 64 thousand so-called code points. The first 127 Unicode code points match the first 127 characters of ASCII. These Unicode code points can be encoded, again, in various ways in a file. To emphasize that a particular number is a Unicode point, such a number is often denoted as U+xxxx whereby xxxx is a hexadecimal number. An example Unicode encoding is `UTF8`, which stores the first 127 code points in a single byte. This makes a `UTF8` file closely resemble ASCII when no values above 127 are used. To identify the Unicode encoding used in a file, a so-called Byte Order Mark (BOM) can be used in the first few bytes of that file. See also `www.unicode.org` and `en.wikipedia.org/wiki/Byte_order_mark`.

UTF8 encoding preferred

UTF8 is a popular encoding; it resembles ASCII for the first 127 code points and can be used by applications deployed at different locales to unambiguously exchange data. Most modern text editors, including the one in AIMMS, are able to handle UTF8 text files. We recommend UTF8 encoding for AIMMS files, especially when AIMMS is used inside international organizations. AIMMS system files, including the `.ams` model file and the `.aimms` project file, use the UTF8 encoding.

Version control

After each editing session AIMMS will only save the last version of your model files, and will not retain a backup of the previous version of your model files. You are therefore strongly encouraged to use a version control system to keep a history of the changes you made to your model.

Other model files

In addition to the model files AIMMS stores a number of other files with each model. They are:

- a *project file* containing the pages of the graphical (end-)user interface that you have created for your application and all other relevant information such as project options, user menus, fonts, etc., and
- a *data tree file* containing all the stored datasets and cases associated with your application.

2.2 Identifier declarations

Identifier types

Identifiers are the unique names through which you can refer to entities in your model. The most common identifier types in AIMMS are:

- *set*—used for indexing parameters and variables,
- *parameter*—for (multidimensional) data storage,
- *variable* and *arc*—entities of constraints that must be determined,
- *constraint* and *node*—relationships between variables or arcs, usually in the form of (in)equalities,
- *mathematical program*—an objective and a collection of constraints, and
- *procedure* and *function*—code segments to initiate execution.

Declaration forms

The declarations of all identifiers, procedures and functions within an AIMMS application can be provided by means of a uniform attribute notation. For every node within the model tree you can view and change the value of these attributes through a graphical declaration form. This form will show all the attributes that are associated with a particular identifier type, along with their values for the identifier at hand.

Notation used in this manual

In this manual we have chosen to use a textual style representation of all model declarations, which closely resembles the graphical representation in the model tree. In view of the large number of declarations in this manual, we found that a purely graphical presentation in the text was visually distracting. In contrast, the adopted textual representation is succinct and integrates well with the surrounding text.

The `Text` and `Comment` attributes

With every declaration in a model you can associate a `Text` and a `Comment` attribute. The `Comment` attribute is aimed at the modeler, and can be used to describe the contents of a particular node in the model tree, or make remarks that are relevant for later reference. The `Text` attribute is intended for use in the graphical user interface and reporting. It can contain a single line description of the identifier at hand. Many objects in the AIMMS user interface allow you to display this text along with the identifier value(s).

Predefined identifiers

Not only does an AIMMS model consist of sets, parameters and variables that have been defined by you, and thus are specific for your application, AIMMS also provides a number of predefined system identifiers. These identifiers characterize either

- a set of *all* objects with a particular property, for instance the set of AllIdentifiers or the set of AllCases, or
- the *current* value of a particular modeling aspect, for instance the parameter CurrentCase or the parameter CurrentPageNumber.

In most cases these identifiers are read-only, and get their value based on the declarations and settings of your model.

Section identifiers

The structuring sections in your model tree are also considered as AIMMS identifiers. The blanks in a section description are replaced by underscores to form a legal AIMMS identifier name. The identifier thus formed is a subset of AllIdentifiers. This subset contains all the model identifiers that have been declared underneath the associated node. You can conveniently use such sets in, for instance, the EMPTY statement to clean a entire group of identifiers in a single statement, or to construct your own subsets of AllIdentifiers using the set operations available in AIMMS.

2.3 Lexical conventions

Lexical conventions

Before treating the more intricate features of the AIMMS language, we have to discuss its lexical conventions. That is, we have to define the basic building blocks of the AIMMS language. Each one is described in a separate paragraph.

Characters

The set of characters recognized by AIMMS consists of the set of all printable characters, together with the tab character. Tab characters are not expanded by AIMMS. The character immediately following a tab character is positioned at column 9, 17, 25, 33, etc. All other unprintable or control characters are illegal. The presence of an illegal character causes a compiler error.

Numbers

Numerical values are entered in a style similar to that in other computer languages. For data storage AIMMS supports the integer data type as well as the real data type (floating point numbers). During execution, however, AIMMS will always use a double precision floating point representation.

Scientific notation

Following standard practice, the letter e denotes the scientific notation allowing convenient representation of very large or small numbers. The number following the e can only be a positive or negative integer. Two examples of the

use of scientific notation are given by

$$\texttt{1.2e5} = 1.2 \times 10^5 = 120,000$$
$$\texttt{2.72e} - \texttt{4} = 2.72 \times 10^{-4} = 0.000272$$

Special numbers

In addition to the ordinary real numbers, AIMMS allows the special symbols INF, -INF, UNDF, NA, and ZERO as numbers. The precise meaning and use of these symbols is described later in Section 6.1.1.

No blanks within numbers

Blanks cannot be used inside a number since AIMMS treats a blank as a separator. Thus, valid examples of expressions recognized as numbers by AIMMS are

```
0         0.0       .0        0.        +1        1.
0.5       .5        +0.5      +.5       -0.3      -.3
2e10      2e+10     2.e10     0.3e-5    .3e-5     -.3e-05
INF       -INF      NA        ZERO
```

Machine precision

The range of values allowed by AIMMS and the number of significant digits is machine-dependent. AIMMS takes advantage of the accuracy of your machine. This may cause different results when a single model is run on two different machines. Expressions that cause arithmetic under- or overflow evaluate to the symbols ZERO and INF, respectively. Functions and operators requiring integer arguments also accept real numbers that lie within a machine-dependent tolerance of an integer.

Identifiers

Identifiers are the unique names given to sets, indices, parameters, variables, etc. Identifiers can be any sequence of the letters a–z, the digits 0–9 and the underscore _. They must start with either a letter or an underscore. The length of an identifier is limited to 255 characters. Examples of legal identifiers include:

```
a       b78     _c_
A_very_long_but_legal_identifier_containing_underscores
```

The following are not identifiers:

```
39      39id    A-ident      a&b
```

Namespaces

In principle, AIMMS operates with a global namespace for all declared identifiers. By introducing modules into your model (see also Section 35.4), you can introduce multiple namespaces, which can be convenient when a particular model section contains logic that can be shared by multiple AIMMS models. Procedures and functions automatically create a separate namespace, allowing for local identifiers with the same name as global identifiers in your model. You can use the *namespace resolution* operator :: to refer to an identifier in a particular namespace (see also Section 35.4).

Redeclaring AIMMS *keywords*

In general, you are not allowed to redeclare AIMMS keywords as identifiers, unless a keyword refers to a non-essential feature of the language. Whenever you try to redeclare an existing AIMMS keyword, AIMMS will produce a compiler error when a keyword cannot be redeclared, or will give you a one-time option to redeclare a non-essential keyword as a model identifier. In the latter case, the non-essential feature will be permanently unavailable within your project.

Case sensitivity

The AIMMS language is *not* case sensitive. This means that upper and lower case letters can be mixed freely in identifier names but are treated identically by AIMMS. However, AIMMS is *case aware*, in the sense that it will try to preserve or restore the original case wherever possible.

Identifiers with suffices

Some AIMMS data types have additional data associated with them. You have access to this extra data through the identifier name plus a suffix, where the suffix is separated from the identifier by a dot. Examples of suffices are:

```
c.Derivative      Transport.ReducedCost      OutputFile.PageSize
```

You can use a suffix expression associated with a particular identifier as if it were an identifier itself.

Case referencing

In addition, AIMMS also uses the dot notation to refer to the data associated from another case file. An example is given below.

```
CaseDifference(i,j) := Transport(i,j) - ReferenceCase.Transport(i,j);
```

In this example the values of a variable `Transport(i,j)` currently in memory are compared to the values in a particular reference case on disk, identified by the case identifier `ReferenceCase`. You will find more information about case references in Section 6.1.3.

Value types

Any constant or parameter in AIMMS must assume one of the following value types:

- number (either integer or floating point),
- string,
- set element, or
- unit expression.

All value types except unit expressions are discussed below. Unit expressions are explained in Section 32.6.

Strings

Constants of string type in AIMMS are delimited by a double quote character """. To include the double quote character itself in a string, it should be escaped by the backslash character "\" (see also Section 5.3.2). Strings can be used as constants in expressions, as arguments of procedures and functions, and in the initialization of string-valued parameters. The size of strings is limited to 64 Kb.

Sets and set elements

A set is a group of like elements. Sets can be *simple* (one-dimensional) or a *relation* (multi-dimensional). The elements of a simple set are represented either by

- an integer number,
- a single-quoted string of a length less than 255 characters, or
- an unquoted string subject to conditions explained below.

The elements of a relation are represented by tuples of such integers or strings.

Integer elements

The elements of an integer set can be used in expressions as if they were integer numbers. Reversely, you can use integer-valued numerical expressions to indicate an element of an integer set. Some operations with integer set elements are ambiguous, and you have to indicate to AIMMS how you want such operations to be interpreted. This is discussed in Section 3.2.2.

Quoted string elements

The characters allowed in a quoted string elements are the set printable characters except for tab and newline.

Unquoted string elements

For your convenience, the elements of a string set need not be delimited by a single quote when all of the following conditions are met:

- the string used as a set element consists only of letters, digits, underscores and the sign characters "+" and "-,"
- the set element is not a reserved word or token, and
- the set element is used inside a constant expression such as a constant *enumerated set* or *list* expression (see also Sections 5.1.1 and 6.1.2), or inside *table* or a *composite table* used for the initialization of parameters and variables (see also Sections 28.2 and 28.3).

String-valued set elements that are referenced explicitly under any circumstance other than the ones mentioned above, must be quoted unconditionally. To include a single quote character in a set element, it should be preceded by the backslash character "\".

Examples of set elements

The following set elements are examples of set elements that can be used without quotation marks under the conditions mentioned above:

```
label1          1998            1997-12         1997_12
january         january-1998    h2so4           04-Mar-47
```

The following character strings are also valid as set elements, but must be quoted in all cases.

```
'An element containing spaces'
'label with nested quotes: "a*b"'
```

String elements do not have a value

Contrary to integer set elements, string elements do *not* have an associated number value. Thus, the string element `'1993'` does not have the value 1993. If you use string elements to represent numbers, you can use the `Val` function to obtain the associated value. Thus, `Val('1993')` represents the number 1993.

Delimiters

The following delimiters are used by AIMMS:

- a space " " separates keywords, identifiers and numbers,
- a pair of single quotes "'" or double quotes """ delimits set elements and strings, respectively,
- a semicolon ";" separates statements,
- braces "{" and "}" denote the beginning and end of sets and lists,
- a comma "," separates elements of sets and lists,
- parentheses "(" and ")" delimit expressions, tuples of indices and set elements, as well as argument lists of functions and references, and
- square brackets "[" and "]" are used to delimit unit expressions as well as numeric and element ranges. They can also be used as parentheses in expressions and argument lists of functions and references, and for grouping elements in components of an element tuple (see also Section 5.1.1).

In most other expressions parentheses and square brackets can be used interchangeably as long as they match. This feature is useful for making deeply nested expressions more readable.

Limits in AIMMS

The following limits apply within AIMMS.

- the length of a line is limited to 255 characters,
- the number of set elements per set is at most 2^{30},
- the number of indices associated with an identifier is at most 32, and
- the number of running indices used in iterative operations such as `SUM` and `FOR` is at most 16.

2.4 Expressions and statements

Model execution

The creation of an AIMMS model is implemented using two separate but interacting mechanisms. They are:

- automatic updating of the functional relationships specified through expressions in the `Definition` attributes of sets and parameters in your model, and
- manual execution of the statements that constitute the `Body` attribute of the procedures and functions defined in your application.

The precise manner in which these components are executed, and the way they interact, is discussed in detail in Chapters 7 and 8. This section discusses

the general structure of an AIMMS model as well as the requirements for the Definition and Body attributes.

Line length and empty lines

The length of any particular line in the Definition attribute of an identifier or the Body attribute of a procedure or function is limited to 255 characters. Although this full line length may be convenient for data instantiation in the form of large tables, it is recommended that you do not exceed a line length of 80 characters in these attributes in order to preserve maximum readability. Empty lines can be inserted anywhere for easier reading.

Commenting

Expressions and statements in the Body attribute of a procedure or function can be interspersed with comments that are ignored during compilation. AIMMS supports two kinds of comments:

- the tokens "/*" and "*/" for a block comment, and
- the exclamation mark "!" for a one line comment.

Each block comment starts with a "/*" token, and runs up to the matching "*/" token, and cannot be nested. It is a useful method for entering pieces of explanatory text, as well as for temporarily commenting out one or more execution statements. A one-line comment starts anywhere on a line with an exclamation mark "!", and runs up to the end of that line.

Expressions

The value of a Definition attribute must be a valid expression of the appropriate type. An expression in AIMMS can result in either

- a set,
- a set element,
- a string,
- a numerical value,
- a logical value, or
- a unit expression.

Set, element and string expressions are discussed in full detail in Chapter 5, numerical and logical expressions in Chapter 6, while unit expressions are discussed in Chapter 32.

Statements

AIMMS statements in the body of procedures and functions constitute the algorithmic part of a modeling application. All statements are terminated by a semicolon. You may enter multiple statements on a single line, or a single statement over several lines.

Execution statements

To specify the algorithmic part of your modeling application, the following statements can be used:

- assignments—to compute a new value for a data item,
- the SOLVE statement—to solve a mathematical program for the values of its variables,
- flow control statements like IF-THEN-ELSE, FOR, WHILE, REPEAT, SWITCH, and HALT—to manage the flow of execution,
- the OPTION and Property statements—to set identifier properties and options dealing with execution, output, progress, and solvers,
- the data control statements EMPTY, CLEANUP, READ, WRITE, DISPLAY, and PUT—to manage the contents of internal and external data.
- procedure calls—to execute the statements contained in a procedure.

The precise syntax of these execution statements is discussed in Chapters 8 and further.

2.5 Data initialization

Initialization syntax

The initialization of sets, parameters, and variables in an AIMMS application can be done in several ways:

- through the *InitialData attribute* of sets, and parameters,
- by reading in data from an *text file* in AIMMS data format,
- by reading in data from a previous AIMMS session stored in a binary *case file*,
- by reading in the data from an external *ODBC-compliant database*, or
- by initializing an identifier through algebraic assignment statements.

Order of initialization

When starting up your AIMMS application, AIMMS will initialize your model identifiers in the following order.

- Following compilation each identifier is initialized with the contents of its InitialData attribute.
- Subsequently, AIMMS will execute the predefined procedure MainInitialization. You can use it to specify READ statements to read in data from text files, case files or databases. In addition, it can contain any other algebraic statement necessary to initialize one or more identifiers in your model. Of course, you can also leave this procedure empty if you so desire.

The full details of model initialization are discussed in Chapter 25.

Entering the InitialData attribute

The InitialData attribute of an identifier can contain any *constant* set-valued, set element-valued, string-valued, or numerical expression. In order to construct such expressions (consisting of mostly tables and lists), AIMMS offers so-called *data pages* which can be created on demand. These pages help you enter the data in a convenient and graphical manner.

Part II

Non-Procedural Language Components

Chapter 3

Set Declaration

This chapter

This chapter covers all aspects associated with the declaration and use of sets in AIMMS models. The main topics are indexing with sets, simple sets with strings, simple sets with integers, relations and indexed sets.

3.1 Sets and indices

General

Sets and indices give your AIMMS model dimension and depth by providing a mechanism for grouping parameters, variables, and constraints. Sets and indices are also used as driving mechanism in arithmetic operations such as summation. The use of sets for indexing expressions helps to describe large models in a concise and understandable way.

Example

Consider a set of *Cities* and an identifier called *Transport* defined between several pairs of cities (i, j), representing the amount of product transported from supply city i to destination city j. Suppose that you are interested in the quantities arriving in each city. Rather than adding many individual terms, the following mathematical notation, using sets and indices, concisely describes the desired computation of these quantities.

$$(\forall\, j \in \mathit{Cities}) \qquad \mathit{Arrival}_j = \sum_{i \in \mathit{Cities}} \mathit{Transport}_{ij}.$$

This multidimensional index notation forms the foundation of the AIMMS modeling language, and can be used in all expressions. In this example, i and j are indices that refer to individual *Cities*.

Several types of sets

A set in AIMMS

- has either *strings* or *integers* as elements,
- is either a *simple* set, or a *relation*, and
- is either *indexed* or *not indexed*.

String versus integer

Sets can either have strings as elements (such as the set *Cities* discussed above), or have integers as elements. An example of an integer set could be a set of *Trials* represented by the numbers $1, \ldots, n$. The resulting integer set can then be used to refer to the results of each single experiment.

Simple versus relation

A *simple* set is a one-dimensional set, such as the set *Cities* mentioned above, while a *relation* or multidimensional set is the Cartesian product of a number of simple sets or a subset thereof. An example of a relation is the set of possible *Routes* between supply and destination cities, which can be represented as a subset of the Cartesian product *Cities* $\times$ *Cities*.

Indexing as basic mechanism

Sets in AIMMS are the basis for creating multidimensional identifiers in your model. Through indices into sets you have access to individual values of these identifiers for each tuple of elements. In addition, the indexing notation in AIMMS is your basic mechanism for expressing iterative operations such as repeated addition, repeated multiplication, sequential search for a maximum or minimum, etc.

Indexed sets

Simple sets may be indexed. An indexed set is a family of sets defined for every element in the index domain of the indexed set. An example of an indexed set is the set of transport destination cities defined for each supply city. On the other hand, the set *Cities* discussed above is not an indexed set.

Sorting of sets

The contents of any simple can be sorted in AIMMS. Sorting can take place either automatically or manually. Automatic sorting is based on the value of some expression defined for all elements of the set. By using an index into a sorted subset, you can access any subselection of data in the specified order. Such a subselection may be of interest in your end-user interface or at a certain stage in your model.

3.2 Set declaration and attributes

Set attributes

Each set has an optional list of attributes which further specify its intended behavior in the model. The attributes of sets are given in Table 3.1. The attributes `IndexDomain` is only relevant to indexed sets.

3.2.1 Simple sets

Definition

A *simple* set in AIMMS is a finite collection of elements. These elements are either strings or integers. Strings are typically used to identify real-world objects such as products, locations, persons, etc.. Integers are typically used for algorithmic purposes. With every simple set you can associate indices through

Attribute	Value-type	See also page
IndexDomain	*index-domain*	37
SubsetOf	*subset-domain*	
Index	*identifier-list*	
Parameter	*identifier-list*	
Text	*string*	19
Comment	*comment string*	19
Property	NoSave, ElementsAreNumerical, ElementsAreLabels	
Definition	*set-expression*	
OrderBy	*expression-list*	

Table 3.1: Set attributes

which you can refer (in succession) to all individual elements of that set in indexed statements and expressions.

Most basic example

An example of the most basic declaration for the set *Cities* from the previous example follows.

```
Set Cities {
    Index      : i,j;
}
```

This declares the identifier Cities as a simple set, and binds the identifiers i and j as indices to Cities throughout your model text.

More detailed example

Consider a set *SupplyCities* which is declared as follows:

```
Set SupplyCities {
    SubsetOf   : Cities;
    Parameter  : LargestSupplyCity;
    Text       : The subset of cities that act as supply city;
    Definition : {
        { i | Exists( j | Transport(i,j) ) }
    }
    OrderBy    : i;
}
```

The "|" operator used in the definition is to be read as "such that" (it is explained in Chapter 5). Thus, SupplyCities is defined as the set of all cities from which there is transport to at least one other city. All elements in the set are ordered lexicographically. The set has no index of its own, but does have an element parameter LargestSupplyCity that can hold any particular element with a specific property. For instance, the following assignment forms one way to specify the value of this element parameter:

```
LargestSupplyCity := ArgMax( i in SupplyCities, sum( j, Transport(i,j) ) );
```

Note that this assignment selects that particular element from the subset of SupplyCities for which the total amount of Transport leaving that element is the largest.

The SubsetOf attribute

With the SubsetOf attribute you can tell AIMMS that the set at hand is a subset of another set, called the *subset domain*. For simple sets, such a subset domain is denoted by a single set identifier. During the execution of the model AIMMS will assert that this subset relationship is satisfied at all times.

Root sets

Each simple set that is not a subset of another set is called a *root set*. As will be explained later on, root sets have a special role in AIMMS with respect to data storage and ordering.

The Index attribute

An index takes the value of *all* elements of a set successively and in the order specified by its declaration. It is used in operations like summation and indexed assignment over the elements of a set. With the Index attribute you can associate identifiers as indices into the set at hand. The index attributes of all sets must be unique identifiers, i.e. every index can be declared only once.

The Parameter attribute

A parameter declared in the Parameter attribute of a set takes the value of a *specific* element of that set. Throughout the sequel we will refer to such a parameter as an *element parameter*. It is a very useful device for referring to set elements that have a special meaning in your model (as illustrated in the previous example). In a later chapter you will see that an element parameter can also be defined separately as a parameter which has a set as its range.

The Text and Comment attributes

With the Text attribute you can specify one line of descriptive text for the end-user. This description can be made visible in the graphical user interface when the data of an identifier is displayed in a page object. You can use the Comment attribute to provide a longer description of the identifier at hand. This description is intended for the modeler and cannot be made visible to an end-user. The Comment attribute is a multi-line string attribute.

Quoting identifier names in Comment

You can make AIMMS aware that specific words in your comment text are intended as identifier names by putting them in single quotes. This has the advantage that AIMMS will update your comment when you change the name of that identifier in the model editor, or, that AIMMS will warn you when a quoted name does not refer to an existing identifier.

The OrderBy attribute

With the OrderBy attribute you can indicate that you want the elements of a certain set to be ordered according to a single or multiple ordering criteria. Only simple sets can be ordered.

Ordering root sets

A special word of caution is in place with respect to specifying an ordering principle for root sets. Root sets play a special role within AIMMS because all data defined over a root set or any of its subsets is stored in the original *data entry* order in which elements have been added to that root set. Thus, the data entry order defines the natural order of execution over a particular domain, and specifying the OrderBy attribute of a root set may influence overall execution times of your model in a negative manner. Section 13.2.7 discusses these efficiency aspects in more detail, and provides alternative solutions.

Ordering criteria

The value of the OrderBy attribute can be a comma-separated list of one or more ordering criteria. The following ordering criteria (numeric, string or user-defined) can be specified.

- If the value of the OrderBy attribute is an indexed numerical expression defined over the elements of the set, AIMMS will order its elements in increasing order according to the numerical values of the expression.
- If the value of the OrderBy attribute is either an index into the set, a set element-valued expression, or a string expression over the set, then its elements will be ordered lexicographically with respect to the strings associated with the expression. By preceding the expression with a minus sign, the elements will be ordered reverse lexicographically.
- If the value of the OrderBy attribute is the keyword User, the elements will be ordered according to the order in which they have been added to the subset, either by the user, the model, or by means of the Sort operator.

Specifying multiple criteria

When applying a single ordering criterion, the resulting ordering may not be unique. For instance, when you order according to the size of transport taking place from a city, there may be multiple cities with equal transport. You may want these cities to be ordered too. In this case, you can enforce a more refined ordering principle by specifying multiple criteria. AIMMS applies all criteria in succession, and will order only those elements that could not be uniquely distinguished by previous criteria.

Example

The following set declarations give examples of various types of automatic ordering. In the last declaration, the cities with equal transport are placed in a lexicographical order.

```
Set LexicographicSupplyCities {
    SubsetOf  : SupplyCities;
    OrderBy   : i;
}
Set ReverseLexicographicSupplyCities {
    SubsetOf  : SupplyCities;
    OrderBy   : - i;
}
Set SupplyCitiesByIncreasingTransport {
    SubsetOf  : SupplyCities;
    OrderBy   : sum( j, Transport(i,j) );
}
```

```
Set SupplyCitiesByDecreasingTransportThenLexicographic {
    SubsetOf  : SupplyCities;
    OrderBy   : - sum( j, Transport(i,j) ), i;
}
```

The Property attribute

In general, you can use the Property attribute to assign additional properties to an identifier in your model. The applicable properties depend on the identifier type. Sets, at the moment, only support a single property.

- The property NoSave specifies that the contents of the set at hand will never be stored in a case file. This can be useful, for instance, for intermediate sets that are necessary during the model's computation, but are never important to an end-user.
- The properites ElementsAreNumerical and ElementsAreLabels are only relevant for integer sets (see also Section 3.2.2). They will ignored for non-integer sets.

Dynamic property selection

The properties selected in the Property attribute of an identifier are on by default, while the nonselected properties are off by default. During execution of your model you can also dynamically change a property setting through the Property statement. The PROPERTY statement is discussed in Section 8.5.

The Definition attribute

If an identifier can be uniquely defined throughout your model by a single expression, you can (and should) use the Definition attribute to specify this global relationship. AIMMS stores the result of a Definition and recomputes it only when necessary. For sets where a global Definition is not possible, you can make assignments in procedures and functions. The value of the Definition attribute must be a valid expression of the appropriate type, as exemplified in the declaration

```
Set SupplyCities {
    SubsetOf   : Cities;
    Definition : {
        { i | Exists( j | Transport(i,j) ) }
    }
}
```

3.2.2 Integer sets

Integer sets

A special type of simple set is an integer set. Such a set is characterized by the fact that the value of the SubsetOf attribute must be equal to the predefined set Integers or a subset thereof. Integer sets are most often used for algorithmic purposes.

Usage in expressions

Elements of integer sets can also be used as integer values in numerical expressions. In addition, the result of an integer-valued expression can be added as an element to an integer set. Elements of non-integer sets that represent numerical values cannot be used directly in numerical expressions. To obtain the numerical value of such non-integer elements, you can use the `Val` function (see Section 5.2.1).

Interpret values as integer or label?

The interpretation of integer set elements will as integer values in numerical expressions, raises an ambiguity for certain types of expressions. If `anInteger` is an element parameter into an integer set `anIntegerSet`,

- how should AIMMS handle the expression

  ```
  if (anInteger) then
      ...
  endif;
  ```

  where `anInteger` holds the value `'0'`. On the one hand, it is not the empty element, so if AIMMS would interpret this as a logical expression with a non-empty element parameter, the `if` statement would evaluate to true. If AIMMS would interpret this as a numerical expression, the element parameter would evaluate to the numerical value 0, and the `if` statement would evaluate to false.
- how should AIMMS handle the assignment

  ```
  anInteger := anInteger + 3;
  ```

  if the values in `anIntegerSet` are non-contiguous? If AIMMS would interpret `anInteger` as an ordinary element parameter, the `+` operator would refer to a lead operator (see also Section 5.2.3), and the assignment would assign the third next element of `anInteger` in the set `anIntegerSet`. If AIMMS would interpret `anInteger` as an numerical value, the assignment would assign the numerical value of `anInteger` plus 3, assuming that this is an element of `anIntegerSet`.

You can resolve this ambiguity assigning one of the properties `ElementsAreLabels` and `ElementsAreNumerical` to `anIntegerSet`. If you don't assign either property, and you use one of these expressions in your model, AIMMS will issue a warning about the ambiguity, and the end result might be unpredictable.

Construction

In order to fill an integer set AIMMS provides the special operator "`..`" to specify an entire range of integer elements. This powerful feature is discussed in more detail in Section 5.1.1.

Example

The following somewhat abstract example demonstrates some of the features of integer sets. Consider the following declarations.

```
Parameter LowInt {
    Range      : Integer;
}
Parameter HighInt {
    Range      : Integer;
}
Set EvenNumbers {
    SubsetOf    : Integers;
    Index       : i;
    Parameter   : LargestPolynomialValue;
    OrderBy     : - i;
}
```

The following statements illustrate some of the possibilities to compute integer sets on the basis of integer expressions, or to use the elements of an integer set in expressions.

```
    ! Fill the integer set with the even numbers between
    ! LowInt and HighInt. The first term in the expression
    ! ensures that the first integer is even.

    EvenNumbers := { (LowInt + mod(LowInt,2)) .. HighInt by 2 };

    ! Next the square of each element i of EvenNumbers is added
    ! to the set, if not already part of it (i.e. the union results)

    for ( i | i <= HighInt ) do
        EvenNumbers += i^2;
    endfor;

    ! Finally, compute that element of the set EvenNumbers, for
    ! which the polynomial expression assumes the maximum value.

    LargestPolynomialValue := ArgMax( i, i^4 - 10*i^3 + 10*i^2 - 100*i );
```

Ordering integer sets

By default, integer sets are ordered according to the numeric value of their elements. Like with ordinary simple sets, you can override this default ordering by using the `OrderBy` attribute. When you use an index in specifying the order of an integer set, AIMMS will interpret it as a numeric expression.

3.2.3 Relations

Relation

A *relation* or multidimensional set is the Cartesian product of a number of simple sets or a subset thereof. Relations are typically used as the domain space for multidimensional identifiers. Unlike simple sets, the elements of a relation cannot be referenced using a single index.

Tuples and index components

An element of a relation is called a *tuple* and is denoted by the usual mathematical notation, i.e. as a parenthesized list of comma-separated elements. Throughout, the word *index component* will be used to denote the index of a particular position inside a tuple.

Index tuple

To reference an element in a relation, you can use an *index tuple*, in which each tuple component contains an index corresponding to a simple set.

The `SubsetOf` attribute

The `SubsetOf` attribute is mandatory for relations, and must contain the *subset domain* of the set. This subset domain is denoted either as a parenthesized comma-separated list of simple set identifiers, or, if it is a subset of another relation, just the name of that set.

Example

The following example demonstrates some elementary declarations of a relation, given the two-dimensional parameters `Distance(i,j)` and `TransportCost(i,j)`. The following set declaration defines a relation.

```
Set HighCostConnections {
    SubsetOf   : (Cities, Cities);
    Definition : {
        { (i,j) | Distance(i,j) > 0 and TransportCost(i,j) > 100 }
    }
}
```

3.2.4 Indexed sets

Definition

An *indexed set* represents a family of sets defined for all elements in another set, called the *index domain*. The elements of all members of the family must be from a single (sub)set. Although membership tables allow you to reach the same effect, indexed sets often make it possible to express certain operations very concisely and intuitively.

The `IndexDomain` attribute

A set becomes an indexed set by specifying a value for the `IndexDomain` attribute. The value of this attribute must be a single index or a tuple of indices, optionally followed by a logical condition. The precise syntax of the `IndexDomain` attribute is discussed on page 42.

Example

The following declarations illustrate some indexed sets with a content that varies for all elements in their respective index domains.

```
Set SupplyCitiesToDestination {
    IndexDomain  : j;
    SubsetOf     : Cities;
    Definition   : {
        { i | Transport(i,j) }
    }
}
```

```
Set DestinationCitiesFromSupply {
    IndexDomain  : i;
    SubsetOf     : Cities;
    Definition   : {
        { j | Transport(i,j) }
    }
}
Set IntermediateTransportCities {
    IndexDomain  : (i,j);
    SubsetOf     : Cities;
    Definition   : DestinationCitiesFromSupply(i) * SupplyCitiesToDestination(j);
    Comment      : {
        All intermediate cities via which an indirect transport
        from city i to city j with one intermediate city takes place
    }
}
```

The first two declarations both define a one-dimensional family of subsets of `Cities`, while the third declaration defines a two-dimensional family of subsets of `Cities`. Note that the `*` operator is applied to sets, and therefore denotes intersection.

Subset domains

The subset domain of an indexed set family can be either a simple set identifier, or another family of indexed simple sets of the same or lower dimension. The subset domain of an indexed set *cannot* be a relation.

No default indices

Declarations of indexed sets do not allow you to specify either the `Index` or `Parameter` attribute. Consequently, if you want to use an indexed set for indexing, you must locally bind an index to it. For more details on the use of indices and index binding refer to Sections 3.3 and 9.1.

3.3 INDEX declaration and attributes

Direct versus indirect declaration

Every index used in your model must be declared exactly once. You can declare indices *indirectly*, through the `Index` attribute of a simple set, or *directly* using an `Index` declaration. Note that all previous examples show indirect declaration of indices.

Index declaration

When you choose to declare an index not as an attribute of a set declaration, you can use the `Index` declaration. The attributes of each single index declaration are given in Table 3.2.

The Range attribute

You can assign a default binding with a specific set to directly declared indices by specifying the `Range` attribute. If you omit this `Range` attribute, the index has no default binding to a specific set and can only be used in the context of local or implicit index binding. The details of index binding are discussed in Section 9.1.

Attribute	Value-type	See also page
Range	*set-identifier*	
Text	*string*	19
Comment	*comment string*	19

Table 3.2: Index attributes

The following declaration illustrates a direct Index declaration. *Example*

```
Index c {
    Range : Customers;
}
```

Chapter 4

Parameter Declaration

Terminology

The word parameter does not have a uniform meaning in the scientific community. When you are a statistician, you are likely to view a parameter as an unknown quantity to be estimated from observed data. *In* AIMMS *the word parameter denotes a known quantity that holds either numeric or string-valued data.* In programming languages the term variable is used for this purpose. However, this is not the convention adopted in AIMMS, where, in the context of a mathematical program, the word variable is reserved for an unknown quantity. Outside this context, a variable behaves as if it were a parameter. The terminology in AIMMS is consistent with the standard operations research terminology that distinguishes between parameters and variables.

Why use parameters

Rather than putting the explicit data values directly into your expressions, it is a much better practice to group these values together in parameters and to write all your expressions using these symbolic parameters. Maintaining a model that contains explicit data is a painstaking task and error prone, because the meaning of each separate number is not clear. Maintaining a model in symbolic form, however, is much easier and frequently boils down to simply adjusting the data of a few clearly named parameters at a single point.

Example

Consider the set Cities introduced in the previous chapter and a parameter FixedTransport(i,j). Suppose that the cost of each unit of transport between cities i and j is stored in the parameter UnitTransportCost(i,j). Then the definition of TotalTransportCost can be expressed as

```
TotalTransportCost := sum[(i,j), UnitTransportCost(i,j)*FixedTransport(i,j)];
```

Not only is this expression easy to understand, it also makes your model extendible. For instance, an extra city can be added to your model by simply adding an extra element to the set Cities as well as updating the tables containing the data for the parameters UnitTransportCost and FixedTransport. After these changes the above statement will automatically compute TotalTransportCost based on the new settings without any explicit change to the symbolic model formulation.

4.1 Parameter declaration and attributes

Declaration and attributes

There are four parameter types in AIMMS that can hold data of the following four data types:

- **Parameter** for numeric values,
- **StringParameter** for strings,
- **ElementParameter** for set elements, and
- **UnitParameter** for unit expressions.

Prior to declaring a parameter in the model editor you need to decide on its data type. In the model tree parameters of each type have their own icon. The attributes of parameters are given in Table 4.1.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	
Range	*range*	
Default	*constant-expression*	
Unit	*unit-expression*	
Property	NoSave, Stochastic, Uncertain, Random, *numeric-storage-property*	45
Text	*string*	19
Comment	*comment string*	19, 32
Definiton	*expression*	34
InitialData	*data enumeration*	423
Uncertainty	*expression*	46, 337
Region	*expression*	46, 334
Distribution	*expression*	46, 340

Table 4.1: Parameter attributes

Basic examples

The following declarations demonstrate some basic parameter declarations

```
Parameter Population {
     IndexDomain  :  i;
     Range        :  [0,inf);
     Unit         :  [ 1000 ];
     Text         :  Population of city i in thousands;
}
Parameter Distance {
     IndexDomain  :  (i,j);
     Range        :  [0,inf);
     Unit         :  [ km ];
     Text         :  Distance from city i to city j in km;
}
```

```
ElementParameter cityWithLargestPopulation {
     Range        :  cities;
     Definition   :  argMax( i, Population( i ) );
}
StringParameter emergencyMessage {
     InitialData :  "Warning";
}
Quantity Currencies {
     BaseUnit     :  dollar;
     Conversions  :  euro -> dollar : # -> # * 1.3;
}
UnitParameter selectedCurrency {
     InitialData :  [euro];
}
```

The IndexDomain attribute

For each multidimensional identifier you need to specify its dimensions by providing a list of index bindings at the IndexDomain attribute. Identifiers without an IndexDomain are said to be *scalar*. In the index domain you can specify default or local bindings to simple sets. The totality of dimensions of all bindings determine the total dimension of the identifier. Any references outside the index domain, either through execution statements or from within the graphical user interface are skipped.

Domain condition

You can also use the IndexDomain attribute to specify a logical expression which further restricts the valid tuples in the domain. During execution, assignments to tuples that do not satisfy the domain condition are ignored. Also, evaluation of references to such tuples in expressions will result in the value zero. Note that, if the domain condition contains references to other data in your model, the set of valid tuples in the domain may change during a single interactive session.

Example

Consider the sets ConnectedCities with default index cc and DestinationCitiesFromSupply(i) from the previous chapter. The following statements illustrate a number of possible declarations of the two-dimensional identifier UnitTransportCost with varying index domains.

```
Parameter UnitTransportCost {
    IndexDomain : (i,j);
}
Parameter UnitTransportCostWithCondition {
    IndexDomain : (i,j) in ConnectedCities;
}
Parameter UnitTransportCostWithIndexedDomain {
    IndexDomain : (i, j in DestinationCitiesFromSupply(i));
}
```

Explanation

The identifiers defined in the previous example will behave as follows.

- The identifier UnitTransportCost is defined over the full Cartesian product Cities × Cities by means of the default bindings of the indices i and j. You will be able to assign values to every pair of cities (i,j), even though there is no connection between them.
- The identifier UnitTransportCostWithCondition is defined over the same Cartesian product of sets. Its domain, however, is restricted by an additional condition (i,j) in ConnectedCities which will exclude assignments to tuples that do not satisfy this condition, or evaluate to zero when referenced.
- Finally, the identifier UnitTransportCostWithIndexedDomain is defined over a subset of the Cartesian product Cities × Cities. The second element j must lie in the subset DestinationCities(i) associated with i. AIMMS will produce a domain error if this condition is not satisfied.

The Range attribute

With the Range attribute you can restrict the values to certain intervals or sets. The Range attribute is not applicable to a StringParameter nor to a UnitParameter. The possible values for the Range attribute are:

- one of the predefined ranges Real, Nonnegative, Nonpositive, Integer, or Binary,
- any one of the interval expressions $[a,b]$, $[a,b)$, $(a,b]$, or (a,b), where a square bracket implies inclusion into the interval and a round bracket implies exclusion,
- any enumerated integer set expression, e.g. $\{a \text{ .. } b\}$ covering all integers from a until and including b,
- a set reference, if you want the values to be elements of that set. For set element-valued parameters this entry is mandatory.

The values for a and b can be a constant number, inf, -inf, or a parameter reference involving some or all of the indices on the index domain of the declared identifier.

Example

Consider the following declarations.

```
Parameter UnitTransportCost {
    IndexDomain  :  (i,j);
    Range        :  [ UnitLoadingCost(i), 100 ];
}
Parameter  DefaultUnitsShipped {
    IndexDomain  :  (i,j);
    Range        :  {
        { MinShipment(i) .. MaxShipment(j) }
    }
}
Set States {
    Index        :  s;
}
Set adjacentStates {
    SubsetOf     :  States;
```

```
    IndexDomain  :  s;
}
ElementParameter nextState {
    IndexDomain  :  s;
    Range        :  adjacentStates(s);
}
```

It limits the values of the identifier UnitTransportCost(i,j) to an interval from UnitLoadingCost(i) to 100. Note that the lower bound of the interval has a smaller dimension than the identifier itself. The integer identifier Default-UnitsShipped(i,j) is limited to an integer range through an enumerated integer range inside the set brackets.

The Default attribute

In AIMMS, parameters that have not been assigned an explicit value are given a default value automatically. You can specify the default value with the Default attribute. The value of this attribute *must* be a constant expression. If you do not provide a default value for the parameter, AIMMS will assume the following defaults:

- 0 for numbers,
- 1 for unit-valued parameters,
- the empty string "" for strings, and
- the empty element '' for set elements.

The Definition attribute

The Definition attribute of a parameter can contain a valid (indexed) numerical expression. Whenever a defined parameter is referenced inside your model, AIMMS will, by default, recompute the associated data if (data) changes to any of the identifiers referenced in its definition make its current data out-of-date. In the definition expression you can refer to any of the indices in the index domain as if the definition was the right-hand side of an assignment statement to the parameter at hand (see also Section 8.2).

Example

The following declaration illustrates an indexed Definition attribute.

```
Parameter MaxTransportFrom {
    IndexDomain  : i;
    Definition   : Max(j, Transport(i,j));
}
```

Care when used in loops

Whenever you provide a definition for an *indexed* parameter, you should carefully verify whether and how that parameter is used in the context of one of AIMMS' loop statements (see also Section 8.3). When, due to changes in only a slice of the dependent data of a definition during a previous iteration, AIMMS (in fact) only needs to evaluate a single slice of a defined parameter during the actual iteration, you should probably not be using a defined parameter. AIMMS' automatic evaluation scheme for defined identifiers will always recompute the data for such identifiers *for the whole domain of definition*, which can lead to

severe inefficiencies for high-dimensional defined parameters. You can find a more detailed discussion on this issue in Section 13.2.3.

The Unit attribute

By associating a Unit to every numerical identifier in your model, you can let AIMMS help you check your model's consistency. AIMMS also uses the Unit attribute when presenting data and results in both the output files of a model and the graphical user interface. You can find more information on the use of units in Chapter 32.

The Property attribute

The Property attribute can hold various properties of the identifier at hand. The allowed properties for a parameter are NoSave or one of the numerical storage properties Integer, Integer32, Integer16, Integer8 or Double, in addition to the properties Stochastic, Uncertain, Random which are discussed in Section 4.1.1.

- The property NoSave indicates whether the identifier values are stored in cases. It is discussed in detail in Section 3.2.
- By default, the values of numeric parameters are stored as double precision floating point numbers. By specifying one of the storage properties Integer, Integer32, Integer16, Integer8, or Double AIMMS will store the values of the identifier as (signed) integers of default machine length, 4 bytes, 2 bytes or 1 byte, or as a double precision floating point number respectively. These properties are only applicable to parameters with an integer range.

The Property statement

During execution you can change the properties of a parameter through the Property statement. The syntax of the Property statement and examples of its use can be found in Section 8.5.

The Text attribute

With the Text attribute you can provide one line of descriptive text for the end-user. If the Text string of an indexed parameter or variable contains a reference to one or more indices in the index domain, then the corresponding elements are substituted for these indices in any display of the identifier text.

4.1.1 Properties and attributes for uncertain data

Stochastic programming and robust optimization

The AIMMS modeling language allows you to specify both stochastic programs and robust optimization models. Both methodologies are designed to deal with models involving data uncertainty. In stochastic programming the uncertainty is expressed by specifying multiple scenarios, each of which can define scenario-specific values for certain parameters in your model. Stochastic programming is discussed in full detail in Chapter 19. For robust optimization, parameters can be declared to not have a single fixed value, but to take their

values from an user-defined uncertainty set. Robust optimization is discussed in Chapter 20.

Properties

The following Parameter properties are available in support of stochastic programming and robust optimization models.

- The property Stochastic indicates that the identifier can hold stochastic event data for a stochastic model. It is discussed in detail in Section 19.2.
- The property Uncertain indicates that the identifier can hold uncertain values from an uncertainty set specified through the Uncertainty and/or Region attributes. Uncertain parameters are used in AIMMS' robust optimization facilities, and are discussed in detail in Section 20.2.
- The property Random indicates that the identifier can hold random values with respect to a distribution with characteristics specified through the Distribution attribute. Random parameters are used in AIMMS' robust optimization facilities, and are discussed in detail in Section 20.3.

The Uncertainty and Region attributes

The Uncertainty and Region attributes are available if the parameter at hand has been declared uncertain using the Uncertain property. Uncertain parameters are used by AIMMS' robust optimization framework, and are discussed in full detail in Section 20.2. With the Region attribute you can specify an uncertainty set using one of the predefined uncertainty sets Box, ConvexHull or Ellipsoid. The Uncertainty attribute specifies a relationship between the uncertain parameter at hand, and one or more other (uncertain) parameters in your model. The Uncertainty and Region attributes are not exclusive, i.e., you are allowed to specify both, in which case AIMMS' generation process of the robust counterpart will make sure that both conditions are satisfied by the final solution.

The Distribution attribute

The Distribution attribute is available if the parameter at hand has been declared random using the Random property. Random parameters are used by AIMMS' robust optimization framework, and are discussed in full detail in Section 20.3. With the Distribution attribute you can declare that the values for the random parameter at hand adhere to one of the predefined distributions discussed in Section 20.3.

Chapter 5

Set, Set Element and String Expressions

Several types of expressions

Expressions are organized arrangements of operators, constants, sets, indices, parameters, and variables that evaluate to either a set, a set element, a numerical value, a logical value, a string value, or a unit value. Expressions form the core of the AIMMS language. In the previous chapters you already have seen some elementary examples of expressions.

This chapter

In this chapter, set, set element and string expressions are presented in detail. For expressions that evaluate to either numerical or logical values, you are referred to Chapter 6. Expressions that evaluate to unit values are discussed in Section 32.6

5.1 Set expressions

Set expressions

Set expressions play an important role in the construction of index domains of indexed identifiers, as well as in constructing the domain of execution of particular indexed statements. The AIMMS language offers a powerful set of set expressions, allowing you to express complex set constructs in a clear and concise manner.

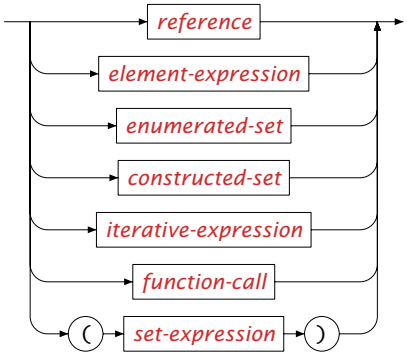
Constant set expressions

A set expression is evaluated to yield the value of a set. As with all expressions in AIMMS, set expressions come in two forms, *constant* and *symbolic*. Constant set expressions refer to explicit set elements directly, and are mainly intended for set initialization. The tabular format of set initialization is treated in Section 28.2.

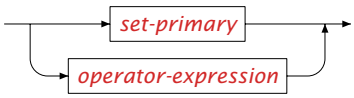
Symbolic set expressions

Symbolic set expressions are formulas that can be executed to result in a set. The contents of this set can vary throughout the execution of your model depending on the values of the other model identifiers referenced inside the symbolic formulas. Symbolic set expressions are typically used for specifying index domains. In this section various forms of set expressions will be treated.

set-primary : *Syntax*



set-expression :



The simplest form of set expression is the reference to a set. The reference can be scalar or indexed, and evaluates to the current contents of that set. *Set references*

5.1.1 Enumerated sets

An *enumerated* set is a set defined by an explicit enumeration of its elements. Such an enumeration includes literal elements, set element expressions, and (constant or symbolic) element ranges. An enumerated set can be either a simple or a relation. If you use an *integer element range*, an integer set will result. *Enumerated sets*

Enumerated sets come in two flavors: *constant* and *symbolic*. Constant enumerated sets are preceded by the keyword DATA, and must only contain literal set elements. These set elements do not have to be contained in single quotes unless they contain characters other than the alpha-numeric characters, the underscore, the plus or the minus sign. *Constant enumerated sets*

The following simple set and relation assignments illustrate constant enumerated set expressions. *Example*

```
Cities := DATA { Amsterdam, Rotterdam, 'The Hague', London, Paris, Berlin, Madrid } ;

DutchRoutes := DATA { (Amsterdam, Rotterdam  ), (Amsterdam, 'The Hague'),
                      (Rotterdam, Amsterdam  ), (Rotterdam, 'The Hague') } ;
```

Symbolic enumerated sets

Any enumerated set not preceded by the keyword DATA is considered symbolic. Symbolic enumerated sets can also contain element parameters. In order to distinguish between literal set elements and element parameters, all literal elements inside symbolic enumerated sets must be quoted.

Examples

The following two set assignments illustrate the use of enumerated sets that depend on the value of the element parameters SmallestCity, LargestCity and AirportCity.

```
ExtremeCities := { SmallestCity, LargestCity } ;

Routes        := { (LargestCity, SmallestCity), (AirportCity, LargestCity )  } ;
```

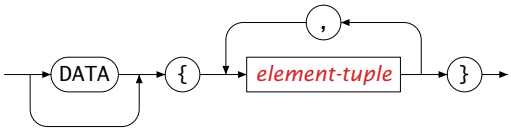
The following two set assignments contrast the semantics between constant and symbolic enumerated sets.

```
SillyExtremes := DATA { SmallestCity, LargestCity } ;
! contents equals { 'SmallestCity', 'LargestCity' }

ExtremeCities := { SmallestCity, LargestCity, 'Amsterdam' };
! contents equals e.g. { 'The Hague', 'London', 'Amsterdam' }
```

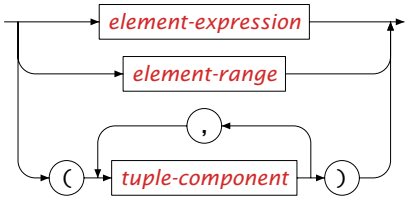
The syntax of enumerated set expressions is as follows.

Syntax

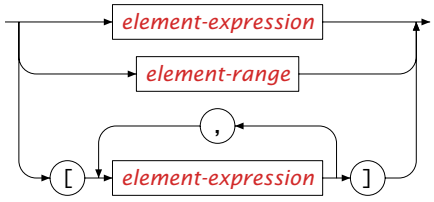
enumerated-set :



element-tuple :



tuple-component :



All elements in an enumerated set must have the same dimension.

Element range

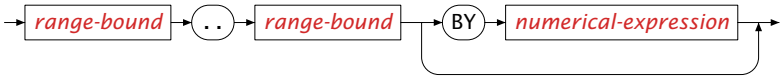
By using the .. operator, you can specify an *element range*. An element range is a sequence of consecutively numbered elements. The following set assignments illustrate both constant and symbolic element ranges. Their difference is explained below.

```
NodeSet       := DATA { node1 .. node100 } ;
```

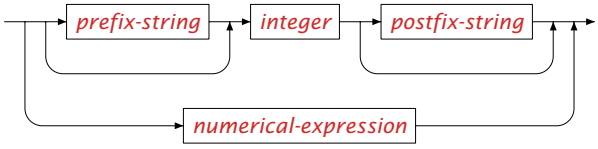
```
FirstNode     := 1;
LastNode      := 100;

IntegerNodes  := { FirstNode .. LastNode } ;
```

The syntax of element ranges is as follows.

element-range : *Syntax*



range-bound :



A range bound must consists of an integer number, and can be preceded or followed by a common prefix or postfix string, respectively. The prefix and postfix strings used in the lower and upper range bounds must coincide. *Prefix and postfix strings*

If you use an element range in a static enumerated set expression (i.e. preceded by the keyword DATA), the range can only refer to explicitly numbered elements, which need not be quoted. By padding the numbered elements with zeroes, you indicate that AIMMS should create all elements with the same element length. *Constant range*

As the begin and end elements of a constant element range are literal elements, you cannot use a constant element range to create sets with dynamically changing border elements. If you want to accomplish this, you should use the ElementRange function, which is explained in detail in Section 5.1.4. Its use in the following example is self-explanatory. The following set assignments illustrate a constant element range and its equivalent formulation using the ElementRange function. *Constant range versus ElementRange*

```
NodeSet     := DATA { node1   .. node100 } ;
PaddedNodes := DATA { node001 .. node100 } ;

NodeSet     := ElementRange( 1, 100, prefix: "node", fill: 0 );
PaddedNodes := ElementRange( 1, 100, prefix: "node", fill: 1 );
```

Symbolic integer range

Element ranges in a symbolic enumerated set can be used to create integer ranges. Now, both bounds can be numerical expressions. Such a construct will result in the *dynamic* creation of a number of *integer* elements based on the value of the numerical expressions at the range bounds. Such integer element ranges can only be assigned to *integer* sets (see Section 3.2.2). An example of a dynamic integer range follows.

```
IntegerNodes := { FirstNode .. LastNode } ;
```

In this example IntegerNodes must be an integer set.

Nonconsecutive range

If the elements in the range are not consecutive but lie at regular intervals from one another, you can indicate this by adding a BY modifier with the proper interval length. For static enumerated sets the interval length must be a constant, for dynamic enumerated sets it can be any numerical expression. The following set assignments illustrate a constant and symbolic element range with nonconsecutive elements.

```
EvenNodes         := DATA { node2 .. node100 by 2 } ;

StepSize          := 2;
EvenIntegerNodes  := { FirstNode .. LastNode by StepSize } ;
```

Element tuples

When specifying element tuples in an enumerated set expression, it is possible to create multiple tuples in a concise manner using cross products. You can specify multiple elements for a particular tuple component in the cross product either by grouping single elements using the [and] operators or by using an element range, as shown below.

```
DutchRoutes := DATA { ( Amsterdam, [Rotterdam, 'The Hague'] ),
                      ( Rotterdam, [Amsterdam, 'The Hague'] )  } ;
! creates  { ( 'Amsterdam', 'Rotterdam' ), ( 'Amsterdam', 'The Hague' ),
!            ( 'Rotterdam', 'Amsterdam' ), ( 'Rotterdam', 'The Hague' ) }

Network     := DATA { ( node1 .. node100, node1 .. node100 ) } ;
```

The assignment to the set Network will create a set with 10,000 elements.

5.1.2 Constructed sets

Constructed sets

A *constructed set* expression is one in which the selection of elements is constructed through filtering on the basis of a particular condition. When a constructed set expression contains an index, AIMMS will consider the resulting tuples for every element in the binding set.

Example

The following set assignments illustrate some constructed set expressions, assuming that i and j are indices into the set Cities.

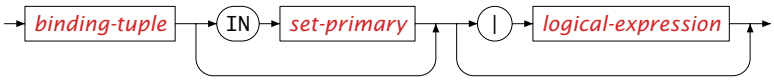
```
LargeCities := { i | Population(i) > 500000 } ;

Routes := { (i,j) | Distance(i,j) } ;

RoutesFromLargestCity := { (LargestCity, j) in Routes } ;
```

In the latter assignment route tuples are constructed from LargestCity (an element-valued parameter) to every city j, where additionally each created tuple is required to lie in the set Routes.

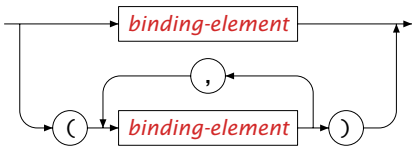
Syntax

constructed-set :

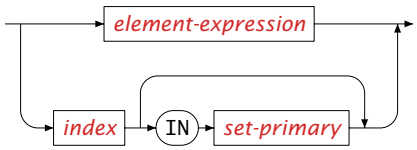
binding-domain :



binding-tuple :



binding-element :



Binding domain

The tuple selection in a constructed set expression behaves exactly the same as the tuple selection on the left-hand side of an assignment to an indexed parameter. This means that all tuple components can be either an explicit quoted set element, a general set element expression, or a binding index. The tuple can be subject to a logical condition, further restricting the number of elements constructed.

5.1.3 Set operators

Four set operators

There are four binary set operators in AIMMS: Cartesian product, intersection, union, and difference. Their notation and precedence are given in Table 5.1. Expressions containing these set operators are read from left to right and the operands can be any set expression. There are no unary set operators.

Operator	Notation	Precedence
intersection	*	3 (high)
difference	-	2
union	+	2
Cartesian product	CROSS	1 (low)

Table 5.1: Set operators

Example

The following set assignments to integer sets and Cartesian products of integer sets illustrate the use of all available set operators.

```
S := {1,2,3,4} * {3,4,5,6} ;     ! Intersection of integer sets: {3,4}.

S := {1,2} + {3,4} ;             ! Union of simple sets:
S := {1,3,4} + {2} + {1,2} ;     ! {1,2,3,4}

S := {1,2,3,4} - {2,4,5,7} ;     ! Difference of integer sets: {1,3}.

T := {1,2} cross {1,2} ;         ! The cross of two integer sets:
                                 ! {(1,1),(1,2),(2,1),(2,2)}.
```

The precedence and associativity of the operators is demonstrated by the assignments

```
T := A cross B - C ;      ! Same as A cross (B - C).
T := A - B * C + D ;      ! Same as (A - (B * C)) + D.
T := A - B * C + D * E ;  ! Same as (A - (B * C)) + (D * E).
```

The operands of union, difference, and intersection must have the same dimensions.

```
T := {(1,2),(1,3)} * {(1,3)} ;              ! Same as {(1,3)}.

T := {(1,2),(1,3)} + {(i,j) | a(i,j) > 1} ; ! Union of enumerated
                                            ! and constructed set of
                                            ! the same dimension.

T := {(1,2),(1,3)} + {(1,2,3)} ;            ! ERROR: dimensions differ.
```

5.1.4 Set functions

Set functions

A special type of set expression is a call to one of the following set-valued functions

- ElementRange,
- SubRange,
- ConstraintVariables,
- VariableConstraints, or
- A user-defined function.

The ElementRange and SubRange functions are discussed in this section, while the functions ConstraintVariables and VariableConstraints are discussed in Section 15.1. The syntax of and use of tags in function calls is discussed in Section 10.2.

The function ElementRange

The ElementRange function allows you to *dynamically* create or change the contents of a set of non-integer elements based on the value of integer-valued scalars expressions.

Arguments

The ElementRange function has two mandatory integer arguments.

- *first*, the integer value for which the first element must be created, and
- *last*, the integer value for which the last element must be created.

In addition, it allows the following four optional arguments.

- *incr*, the integer-valued interval length between two consecutive elements (default value 1),
- *prefix*, the prefix string for every element (by default, the empty string),
- *postfix*, the postfix string (by default, the empty string), and
- *fill*, a logical indicator (0 or 1) whether the numbers must be padded with zeroes (default value 1).

If you use any of the optional arguments you must use their formal argument names as tags.

Example

Consider the sets S and T initialized by the constant set expressions

```
NodeSet     := DATA { node1   .. node100 } ;
PaddedNodes := DATA { node001 .. node100 } ;
EvenNodes   := DATA { node2   .. node100 by 2 } ;
```

These sets can also be created in a dynamic manner by the following applications of the ElementRange function.

```
NodeSet     := ElementRange( 1, 100, prefix: "node", fill: 0 );
PaddedNodes := ElementRange( 1, 100, prefix: "node", fill: 1 );
EvenNodes   := ElementRange( 2, 100, prefix: "node", fill: 0, incr: 2 );
```

The SubRange function

The SubRange function has three arguments:

- a simple *set*,
- the *first* element, and
- the *last* element.

The result of the function is the subset ranging from the *first* to the *last* element. If the first element is positioned after the last element, the empty set will result.

Example

Assume that the set Cities is organized such that all foreign cities are consecutive, and that FirstForeignCity and LastForeignCity are element-valued parameters into the set Cities. Then the following assignment will create the subset ForeignCities of Cities

```
ForeignCities := SubRange( Cities, FirstForeignCity, LastForeignCity ) ;
```

5.1.5 Iterative set operators

Iterative operators

Iterative operators form an important class of operators that are especially designed for indexed expressions in AIMMS. There are set, element-valued, arithmetic, statistical, and logical iterative operators. The syntax is always similar.

Syntax

iterative-expression :

Explanation

The first argument of all iterative operators is a *binding domain*. It consists of a single index or tuple of indices, optionally qualified by a logical condition. The second argument and further arguments must be expressions. These expressions are evaluated for every index or tuple in the binding domain, and the result is input for the particular iterative operator at hand. Indices in the expressions that are not part of the binding domain of the iterative operators are referred to as *outer indices*, and must be bound elsewhere.

Set-related iterative operators

AIMMS possesses the following set-related iterative operators:

- the Sort operator for sorting the elements in a domain,
- the NBest operator for obtaining the n best elements in a domain according to a certain criterion, and
- the Intersection and Union operators for repeated intersection or union of indexed sets.

Reordering your data

Sorting the elements of a set is a useful tool for controlling the flow of execution and for presenting reordered data in the graphical user interface. There are two mechanism available to you for sorting set elements

- the OrderBy attribute of a set, and
- the Sort operator.

Sorting semantics

The second and further operands of the Sort operator must be numerical, element-valued or string expressions. The result of the Sort operator will consist of precisely those elements that satisfy the domain condition, sorted according to the single or multiple ordering criteria specified by the second and further operands. Section 3.2 discusses the expressions that can be used for specifying an ordering principle.

Receiving set

Note that the set to which the result of the Sort operator is assigned must have the OrderBy attribute set to User (see also Section 3.2.1) for the operation to be useful. Without this setting AIMMS will store the elements of the result set of the Sort operator, but will discard the underlying ordering.

Example

The following assignments will result in the same set orderings as in the example of the OrderBy attribute in Section 3.2.

```
LexicographicSupplyCities := Sort( i in SupplyCities, i ) ;

ReverseLexicographicSupplyCities := Sort( i in SupplyCities, -i );

SupplyCitiesByIncreasingTransport :=
    Sort( i in SupplyCities, Sum( j, Transport(i,j) );

SupplyCitiesByDecreasingTransportThenLexicographic :=
    Sort( i in SupplyCities, - Sum( j, Transport(i,j) ), i );
```

Sorting root sets

AIMMS will even allow you to sort the elements of a root set. Because the entire execution system of AIMMS is built around a fixed ordering of the root sets, sorting root sets may influence the overall execution in a negative manner. Section 13.2.7 explains the efficiency considerations regarding root set ordering in more detail.

Obtaining the n best elements

You can use the NBest operator, when you need the n best elements in a set according to a single ordering criterion. The syntax of the NBest is similar to that of the Sort operator. The first expression after the binding domain is the criterion with respect to which you want elements in the binding domain to be ordered. The second expression refers to the number of elements n in which you are interested.

Example

The following assignment will, for every city i, select the three cities to which the largest transports emanating from i take place. The result is stored in the indexed set LargestTransportCities(i).

```
LargestTransportCities(i) := NBest( j, Transport(i,j), 3 );
```

With the Intersection and Union operators you can perform repeated set intersection or union respectively. A typical application is to take the repeated intersection or union of all instances of an indexed set. However, any set valued expression can be used on the second argument.

Repeated intersection and union

Consider the following indexed set declarations.

Example

```
Set IndSet1 {
    IndexDomain : s1;
    SubsetOf    : S;
}
Set IndSet2 {
    IndexDomain : s1;
    SubsetOf    : S;
}
```

With these declarations, the following assignments illustrate valid uses of the Union and Intersection operators.

```
SubS := Union( s1, IndSet1(s1) );
SubS := Intersection( s1, IndSet1(s1) + IndSet2(s1) );
```

5.1.6 Set element expressions as singleton sets

Element expressions can be used in a set expression as well. In the context of a set expression, AIMMS will interpret an element expression as the singleton set containing only the element represented by the element expression. Set element expressions are discussed in full detail in Section 5.2.

Element expressions . . .

Using an element expression as a set expression can equivalently be expressed as a symbolic enumerated set containing the element expression as its sole element. Whenever there is no need to group multiple elements, AIMMS allows you to omit the surrounding braces.

. . . versus enumerated sets

The following set assignment illustrate some simple set element expressions used as a singleton set expression.

Example

```
! Remove LargestCity from the set of Cities
Cities -= LargestCity ;

! Remove first element from the set of Cities
Cities -= Element(Cities,1) ;

! Remove LargestCity and SmallestCity from Cities
Cities -= LargestCity + SmallestCity ;

! The set of Cities minus the CapitalCity
NonCapitalCities := Cities - CapitalCity ;
```

5.2 Set element expressions

Use of set element expressions

Set element expressions reference a particular element or element tuple model from a set or a tuple domain. Set element expressions allow for *sliced assignment*—executing an assignment only for a lesser-dimensional subdomain by fixing certain dimensions to a specific set element. Potentially, this may lead to a vast reduction in execution times for time-consuming calculations.

Passing elements from the GUI

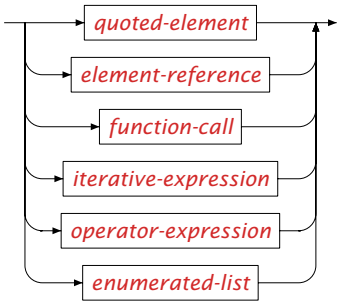
The most elementary form of a set element expression is an element parameter, which turns out to be a useful device for communicating set element information with the graphical interface. You can instruct AIMMS to locate the position in a table or other object where an end-user made changes to a numerical value, and have AIMMS pass the corresponding set element(s) to an element parameter. As a result, you can execute data input checks defined over these element parameters, thereby limiting the amount of computation. This issue is discussed in more detail in the help regarding the Identifier Selection dialog.

Element expressions

AIMMS supports several types of set element expressions, including *references* to parameters and (bound) indices, *lag-lead-expressions*, element-valued *functions*, and *iterative-expressions*. The last category turns out to be a useful device for computing the proper value of element parameters in your model.

Syntax

element-expression :



The format of list expressions are the same for element and numerical expressions. They are discussed in Section 6.1.2.

Element references

An element reference is any reference to either an element parameter or a (bound) index.

5.2.1 Intrinsic functions for sets and set elements

The element-related functions...

AIMMS supports functions to obtain the position of an element within a set, the cardinality (i.e. number of elements) of a set, the n-th element in a set, the element in a non-compatible set with the identical string representation, and the numerical value represented by a set element. If S is a set identifier, i an index bound to S, l an element, and n a positive integer, then possible calls to the Ord, Card, Element, ElementCast and Val functions are given in Table 5.2.

Function	Value	Meaning
Ord(i)	*integer*	Ordinal, returns the relative position of the index i in the set S. Does *not* bind i.
Ord(l,S)	*integer*	Returns the relative position of the element l in set S. Returns zero if l is not an element of S.
Card(S)	*integer*	Cardinality of set S.
Element(S,n)	*element*	Returns the element in set S at relative position n. Returns the empty element tuple if S contains less then n elements.
ElementCast(S,l)	*element*	Returns the element in set S, which corresponds to the textual representation of an element l in any other index set.
Val(l)	*numerical*	Returns the numerical value represented by l, or a runtime error if l cannot be interpreted as a number
Max($e_1,\ldots,e_n$)	*Max*	Returns the set element with the highest ordinal
Min($e_1,\ldots,e_n$)	*Min*	Returns the set element with the lowest ordinal

Table 5.2: Intrinsic functions operating on sets and set elements

... for simple sets

The Ord, Card and Element functions can be applied to simple sets. In fact you can even apply Card to parameters and variables—it simply returns the number of nondefault elements associated with a certain data structure.

Crossing root set boundaries

By default, AIMMS does not allow you to use indices associated with one root set hierarchy in your model, in references to index domains associated with another root set hierarchy of your model. The function ElementCast allows you to cross root set boundaries, by returning the set element in the root set associated with the first (set) argument that has the identical name as the element (in another root set) passed as the second argument. The function ElementCast has an optional third argument *create* (values 0 or 1, with a default

of 0), through which you can indicate whether you want elements which cannot be cast to the indicated set must be created within that set. In this case, a call to ElementCast will never fail. You can find more information about root sets, as well as an illustrative example of the use of ElementCast, in Section 9.1.

Example

In this example, we again use the set Cities initialized through the statement

```
Cities := DATA { Amsterdam, Rotterdam, 'The Hague', London, Paris, Berlin, Madrid } ;
```

The following table illustrates the intrinsic element-valued functions.

Expression	Result
Ord('Amsterdam', Cities)	1
Ord('New York', Cities)	0 (i.e. not in the set)
Card(Cities)	7
Element(Cities, 1)	'Amsterdam'
Element(Cities, 8)	'' (i.e. no 8-th element)

The Val function

If your model contains a set with elements that represent numerical values, you cannot directly use such elements as a numerical value in numerical expressions, unless the set is an integer set (see Section 3.2.2). To obtain the numerical value of such set elements, you can use the Val function. You can also apply the Val function to strings that represent a numerical value. In both cases, a runtime error will occur if the element or string argument of the Val function cannot be interpreted as a numerical value.

The Min and Max functions

The element-valued Min and Max functions operate on two or more element-valued expressions *in the same (sub-)set hierarchy*. If the arguments are references to element parameters (or bound indices), then the Range attributes of these element parameters or indices must be sets in a single set hierarchy. Through these functions you can obtain the elements with the lowest and highest ordinal relative to the set equal to highest ranking range set in the subset hierarchy of all its arguments. If one or more of the arguments are explicit labels, then AIMMS will verify that these labels are contained in that set, or will return an error otherwise. A compiler error will result, if no such set can be determined (i.e., when the function call refers to explicit labels only).

5.2.2 Element-valued iterative expressions

Selecting elements

AIMMS offers special iterative operators that let you select a specific element from a domain. Table 5.3 shows all such operators that result in a set element value. The syntax of iterative operators is explained in Section 5.1.5.

The second column in this table refers to the required number of expression arguments following the binding domain argument.

Name	# Expr.	Computes for all elements in the domain
First	0	the first element (tuple)
Last	0	the last element (tuple)
Nth	1	the n-th element (tuple)
Min	1	the value of the element expression for which the expression reaches its minimum ordinal value
Max	1	the value of the element expression for which the expression reaches its maximum ordinal value
ArgMin	1	the first element (tuple) for which the expression reaches its minimum value
ArgMax	1	the first element (tuple) for which the expression reaches its maximum value

Table 5.3: Element-valued iterative operators

Single index

The binding domain of the First, Last, Nth, Min, Max, ArgMin, and ArgMax operator can only consist of a single index in either a simple set, and the result is a single element in that domain. You can use this result directly for indexing or referencing an indexed parameter or variable. Alternatively, you can assign it to an element parameter in the appropriate domain.

Compared expressions

The ArgMin and ArgMax operators return the element for which an expression reaches its minimum or maximum value. The allowed expressions are:

- numerical expressions, in which case AIMMS performs a numerical comparison,
- string expressions, in which case AIMMS uses the normal alphabetic ordering, and
- element expressions, in which case AIMMS compares the ordinal numbers of the resulting elements.

For element expressions, the iterative Min and Max operators return expression *values* with the minimum and maximum ordinal value.

Example

The following assignments illustrate the use of some of the domain related iterative operators. The identifiers on the left are all element parameters.

```
FirstNonSupplyCity    := First ( i | not Exists(j | Transport(i,j)) ) ;
SecondSupplyCity      := Nth   ( i | Exists(j | Transport(i,j)), 2  ) ;
SmallestSupplyCity    := ArgMin( i, Sum(j, Transport(i,j))          ) ;
LargestTransportRoute := ArgMax( r, Transport(r)                    ) ;
```

Note that the iterative operators `Exists` and `Sum` are used here for illustrative purposes, and are not set- or element-related. They are treated in Sections 6.2.5 and 6.1.6, respectively.

5.2.3 Lag and lead element operators

Lag and lead operators...

There are four binary element operators, namely the lag and lead operators `+`, `++`, `-` and `--`. The first operand of each of these operators must be an element reference (such as an index or element parameter), while the second operand must be an integer numerical expression. There are no unary element operators.

... explained

Lag and lead operators are used to relate an index or element parameter to preceding and subsequent elements in a set. Such correspondence is well-defined, except when a request extends beyond the bounds of the set.

Noncircular versus circular

There are two kinds of lag and lead operators, namely *noncircular* and *circular* operators which behave differently when pushed beyond the beginning and the end of a set.

- The noncircular operators (`+` and `-`) consider the ordered set elements as a *sequence* with no elements before the first element or after the last element.
- The circular operators (`++` and `--`) consider ordered set elements as a *circular chain*, in which the first and last elements are linked.

Definition

Let `S` be a set, `i` a set element expression, and k an integer-valued expression. The lag and lead operators `+`, `++`, `-`, `--` return the element of `S` as defined in Table 5.4. Please note that these operators are also available in the form of `+=`, `-=`, `++=` and `--=`. The operators in this form can be used in statements like:

```
CurrentCity := 'Amsterdam';
CurrentCity --= 1; ! Equal to CurrentCity := CurrentCity -- 1;
```

Lag/lead expr.	Meaning
`i` + k	The element of `S` positioned k elements after `i`; the empty element if there is no such element.
`i` ++ k	The circular version of `i` + k.
`i` − k	The member of `S` positioned k elements before `i`; the empty element if there is no such element.
`i` −− k	The circular version of `i` − k.

Table 5.4: Lag and lead operators

Lag and lead operators for integer sets

For elements in integer sets, AIMMS may interpret the + and - operators either as lag/lead operators or as numerical operators. Section 3.2.2 discusses the way in which you can steer which interpretation AIMMS will employ.

Not for literal elements

You cannot always use lag and lead operators in combination with literal set elements. The reason for this is clear: a literal element can be an element of more than one set, and in general, unless the context in which the lag or lead operator is used dictates a particular (domain) set, it is impossible for AIMMS to determine which set to work with.

Verify the effect of lags and leads

Lag and lead operators are frequently used in indexed parameters and variables, and may appear on the left- and right-hand side of assignments. You should be careful to check the correct use of the lag and lead operators to avoid making conceptual errors. For more specific information on the lag and lead operators refer to Section 8.2, which treats assignments to parameters and variables.

Example

Consider the set Cities initialized through the assignment

```
Cities := DATA { Amsterdam, Rotterdam, 'The Hague', London, Paris, Berlin, Madrid } ;
```

Assuming that the index i and the element parameter CurrentCity both currently refer to 'Rotterdam', Table 5.5 illustrates the results of various lag/lead expressions.

Lag/lead expression	Result
i+1	'The Hague'
i+6	''
i++6	'Amsterdam'
i++7	'Rotterdam'
i-2	''
i--2	'Madrid'
CurrentCity+2	'London'
'Rotterdam' + 1	ERROR

Table 5.5: Example of lag and lead operators

5.3 String expressions

String expressions

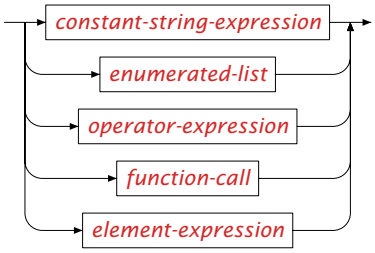
String expressions are useful for

- creating descriptive texts associated with particular set elements and identifiers, or

- forming customized messages for display in the graphical user interface or in output reports.

This section discusses all available string expressions in AIMMS.

string-expression : *Syntax*



The format of list expressions are the same for string-valued and numerical expressions. They are discussed in Section 6.1.2.

5.3.1 String operators

String operators

There are three binary string operators in AIMMS, string concatenation (+ operator), string subtraction (- operator), and string repetition (* operator). There are no unary string operators.

String concatenation

The simplest form of composing strings in AIMMS is by the concatenation of two existing strings. String concatenation is represented as a simple addition of strings by means of the + operator.

String subtraction

In addition to string concatenation, AIMMS also supports subtraction of two strings by means of the - operator. The result of the operation $s_1 - s_2$ where s_1 and s_2 are string expressions will be the substring of s_1 obtained by

- omitting s_2 on the right of s_1 when s_1 ends in the string s_2, or
- just s_1 otherwise.

String repetition

You can use the multiplication operator * to obtain the string that is the result of a given number of repetitions of a string. The left-hand operand of the repetition operator * must be a string expression, while the right-hand operand must be an integer numerical expression.

The following examples illustrate some basic string manipulations in AIMMS. *Examples*

```
"This is "     + "a string"          ! "This is a string"
"Filename.txt" - ".txt"              ! "Filename"
"Filename"     - ".txt"              ! "Filename"
"--"           * 5                   ! "----------"
```

5.3.2 Formatting strings

With the FormatString function you can compose a string that is built up from combinations of numbers, strings and set elements. Its arguments are: *The function FormatString*

- a *format string*, which specifies how the string is composed, and
- one or more *arguments* (number, string or element) which are used to form the string as specified.

The first argument of the function FormatString is a mixture of ordinary text plus *conversion specifiers* for each of the subsequent arguments. A conversion specifier is a code to indicate that data of a specified type is to be inserted as text. Each conversion specifier starts with the % character followed by a letter indicating its type. The conversion specifier for every argument type are given in Table 5.6. *The format string*

Conversion specifiers	Argument type
%s	String expression
%e	Element expression
%f	Floating point number
%g	Exponential format number
%i	Integer expression
%n	Numerical expression
%u	Unit expression
%%	% sign

Table 5.6: Conversion codes for the FormatString function

When using the %f or %g conversion specifier you explicitly choose a floating point or exponential format, respectively. The %n conversion specifier makes this choice for you. If the absolute value of the corresponding argument is greater or equal to 1, %n assures that you get the shortest representation of %f or %g (or even %i if the argument value is integral). However when a non zero width is specified, AIMMS assumes that the alignment of the decimal point is important and thus %n will stick to the use of the floating point format as long as that fits within the given width. If the absolute value of the corresponding *Floating point vs. exponential format*

argument is less than 1, %n uses the floating point format as long as the result shows at least 1 significant digit.

Example

In the example below, the current value of the parameter SmallVal and LargeVal are 10 and 20, the current value of CapitalCity is the element 'Amsterdam', and UnitPar is a unit-valued parameter with value kton/hr. The following calls to FormatString illustrate its use.

```
FormatString("The numbers %i and %i", 10, 20)              ! "The numbers 10 and 20"
FormatString("The numbers %i and %i", SmallVal, LargeVal)  ! "The numbers 10 and 20"
FormatString("The string %s", "is printed")                ! "The string is printed"
FormatString("The element %e", CapitalCity)                ! "The element Amsterdam"
FormatString("The unit is %u", UnitPar)                    ! "The unit is kton/hr"
FormatString("The number %n", 4*ArcTan(1))                 ! "The number 3.141"
FormatString("The large number %n", 1e+6)                  ! "The large number 1.000e+06"
FormatString("The integer %n", 10)                         ! "The integer 10"
FormatString("The fraction %n", 0.01)                      ! "The fraction 0.010"
FormatString("The fraction %n", 0.0001)                    ! "The fraction 1.000e-04"
```

Modification flags

By default, AIMMS will use a default representation for arguments of each type. By modifying the conversion specifier, you further dictate the manner in which a particular argument of the FormatString function is printed. *This is done by inserting modification flags in between the %-sign and the conversion character.* The following modification directives can be added:

- *flags*:
  - **<** for left alignment
  - **<>** for centered alignment
  - **>** for right alignment
  - **+** add a plus sign (nonnegative numbers)
  - **␣** add a space (instead of the above + sign)
  - **0** fill with zeroes (right-aligned numbers only)
  - **t** print number using thousand separators, using local convention for both the thousand separator and decimal separator. Controlling these separators is via the options Number 1000 separator and Number decimal separator.
- *field width*: the converted argument will be printed in a field of at least this width, or wider if necessary
- *dot*: separating the field width from the precision
- *precision*: the number of decimals for numbers, or the maximal number of characters for strings or set elements.

Note the order

It is important to note that the modification flags must be inserted in the order as described above.

Field width and precision

Both the field width and precision of a conversion specifier can be either an integer constant, or a wildcard, *. In the latter case the FormatString expects one additional integer argument for each wildcard just before the argument of the associated conversion specifier. This allows you to compute and specify either the field width or precision in a dynamic manner. If you do not specify a precision as modification directive, the default precision is taken from the option Listing_number_precision. Similarly, the default width is taken from the option Listing_number_width.

Example

The following calls to FormatString illustrate the use of modification flags.

```
FormatString("The number %>+08i", 10)              ! "The number +0000010"
FormatString("The number %>t8i", 100000)           ! "The number  100,000"
FormatString("The number %> 8.2n", 4*ArcTan(1))    ! "The number     3.14"
FormatString("The number %> *.*n", 8,2,4*ArcTan(1)) ! "The number     3.14"
FormatString("The element %<5e", CapitalCity)      ! "The element Amsterdam"
FormatString("The element %<>5.3e", CapitalCity)   ! "The element  Ams "
FormatString("The large number %10.1n", 1e+6)      ! "The large number  1000000.0"
```

Special characters

AIMMS offers a number of special characters to allow you to use the full range of characters in composing strings. These special characters are contained in Table 5.7.

Special character	text code	Meaning
\f	FF	Form feed
\t	HT	Horizontal tab
\n	LF	Newline character
\"	"	Double quote
\\	\	Backslash
\n	n	character n ($001 \leq n \leq 65535$)

Table 5.7: Special characters

Example

Examples of the use of special characters within FormatString follow.

```
FormatString("%i \037 \t %i %%", 10, 11)     ! "10 %       11 %"
FormatString("This is a \"%s\" ", "string")  ! "This is a "string" "
```

Case conversion functions

With the functions StringToUpper, StringToLower and StringCapitalize you can convert the case of a string to upper case, to lower case, or capitalize it, as illustrated in the following example.

```
StringToUpper("Convert to upper case")    ! "CONVERT TO UPPER CASE"
StringToLower("CONVERT to lower case")    ! "convert to lower case"
StringCapitalize("capitaLIZED senTENCE")  ! "Capitalized sentence"
```

5.3.3 String manipulation

Other string related functions

In addition to the FormatString function, AIMMS offers a number of other functions for string manipulation. They are:

- Substring to obtain a substring of a particular string,
- StringLength to determine the length of a particular string,
- FindString to obtain the position of the first occurrence of a particular substring,
- FindNthString to obtain the position of the n-th occurrence of a particular substring, and
- StringOccurrences to obtain the number of occurrences of a particular substring.

The function SubString

With the SubString function you can obtain a substring from a particular begin position m to an end position n (or to the end of the string if the requested end position exceeds the total string length). The positions m and n can both be negative (but with $m \leq n$), in which case AIMMS will start counting backwards from the end of the string. Examples are:

```
SubString("Take a substring of me", 8, 16)    ! returns "substring"
SubString("Take a substring of me", 18, 100)  ! returns "of me"
SubString("Take a substring of me", -5, -1)   ! returns "of me"
```

The function StringLength

The function StringLength can be used to determine the length of a string in AIMMS. The function will return 0 for an empty string, and the total number of characters for a nonempty string. An example follows.

```
StringLength("Guess my length")              ! returns 15
```

The functions FindString and FindNthString

With the functions FindString and FindNthString you can determine the position of the second argument, the *key*, within the first argument, the *search* string. The functions return zero if the key is not contained in the search string. The function FindString returns the position of the first occurrence of the key in the search string starting from the left, while the function FindNthString will return the position of the n-th appearance of the key. If n is negative, the function FindNthString will search backwards starting from the right. Examples are:

```
FindString      ("Find a string in a string", "string"     )  ! returns 8
FindNthString   ("Find a string in a string", "string",  2 )  ! returns 20
FindNthString   ("Find a string in a string", "string", -1 )  ! returns 20

FindString      ("Find a string in a string", "this string")  ! returns 0
FindNthString   ("Find a string in a string", "string",  3 )  ! returns 0
```

Case sensitivity

By default, the functions FindString and FindNthString will use a case sensitive string comparison when searching for the key. You can modify this behavior through the option Case_Sensitive_String_Comparison.

The function StringOccurrences

The function StringOccurrences allows you to determine the number of occurrences of the second argument, the key, within the first argument, the *search* string. You can use this function, for instance, to delimit the number of calls to the function FindNthString a priori. An example follows.

```
StringOccurrences("Find a string in a string", "string"     )  ! returns 2
```

5.3.4 Converting strings to set elements

Converting strings to set elements

Converting strings to new elements to or renaming existing elements in a set is not an uncommon action when end-users of your application are entering new element interactively or when you are obtaining strings (to be used as set elements) from other applications through external procedures. AIMMS offers the following support for dealing with such situations:

- the procedure SetElementAdd to add a new element to a set,
- the procedure SetElementRename to rename an existing element in a set, and
- the function StringToElement to convert strings to set elements.

Adding new set elements

The procedure SetElementAdd lets you add new elements to a set. Its arguments are:

- the *set* to which you want to add the new element,
- an *element parameter* into *set* which holds the new element after addition, and
- the *stringname* of the new element to be added.

When you apply SetElementAdd to a root set, the element will be added to that root set. When you apply it to a subset, the element will be added to the subset as well as to all its supersets, up to and including its associated root set.

Renaming set elements

Through the procedure SetElementRename you can provide a new name for an existing element in a particular set whenever this is necessary in your application. Its arguments are:

- the *set* which contains the element to be renamed,
- the *element* to be renamed, and
- the *stringname* to which the element should be renamed.

After renaming the element, all data defined over the old element name will be available under the new element name.

The function StringToElement

With the function StringToElement you can convert string arguments into (existing) elements of a set. If there is no such element, the function evaluates to the empty element. Its arguments are:

- the *set* from which the element corresponding to *stringname* must be returned,
- the *stringname* for which you want to retrieve the corresponding element, and
- the optional *create* argument (values 0 or 1, with a default of 0) indicating whether nonexisting elements must be added to the set.

With the *create* argument set to 1, a call to StringToElement will always return an element in *set*. Alternatively to setting the *create* argument to 1, you can call the procedure SetElementAdd to add the element to the set.

Example

The following example illustrates the combined use of StringToElement and SetElementAdd. It checks for the existence of the string parameter CityString in the set Cities, and adds it if necessary.

```
ThisCity := StringToElement( Cities, CityString );
if ( not ThisCity ) then
   SetElementAdd( Cities, ThisCity, CityString );
endif;
```

Alternatively, you can combine both statements by setting the optional *create* argument of the function StringToElement to 1.

```
ThisCity := StringToElement( Cities, CityString, create: 1 );
```

Converting element to string

Reversely, you can use the %e specifier in the FormatString function to get a pure textual representation of a set element, as illustrated in the following assignment.

```
CityString := FormatString("%e", ThisCity );
```

Chapter 6

Numerical and Logical Expressions

This chapter

AIMMS has a comprehensive set of built-in numerical and logical operators which allow you quickly and concisely express the details of your model. The subject of `Macros`, which are a parametric form of expression, is also explained. For expressions that evaluate to sets, set elements or strings, see Chapter 5.

6.1 Numerical expressions

Constant numerical expressions

Like any expression in AIMMS, a numerical expression can either be a *constant* or a *symbolic* expression. Constant expressions are those that contain references to explicit set elements and values, but do not contain references to other identifiers. Constant expressions are mostly intended for the initialization of sets, parameters and variables. Such an initialization must conform to one of the following formats:

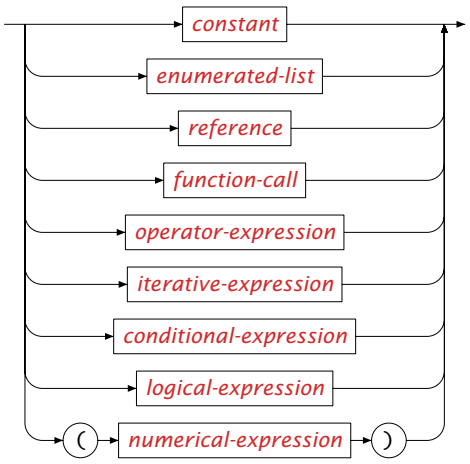
- a *scalar* value,
- a *list* expression,
- a *table* expression, or
- a *composite table*.

Table expressions and composite tables are mostly used for data initialization from *external* files. They are discussed in Chapter 28.

Symbolic numerical expressions

Symbolic expressions are those expressions that contain references to other AIMMS identifiers. They can be used in the `Definition` attributes of sets, parameters and variables, or as the right-hand side of assignment statements. AIMMS provides a powerful notation for expressions, and complicated numerical manipulations can be expressed in a clear and concise manner.

numerical-expression : *Syntax*



6.1.1 Real values and arithmetic extensions

Extension of the real line

Traditional arithmetic is defined on the real line, $\mathbb{R} = (-\infty, \infty)$, which does not contain either $+\infty$ or $-\infty$. AIMMS' arithmetic is defined on the set $\mathbb{R} \cup \{$-INF, INF, NA, UNDF, ZERO$\}$ and summarized in Table 6.1. The symbols INF and -INF are mostly used to model unbounded variables. The symbols NA and UNDF stand for *not available* and *undefined* data values respectively. The symbol ZERO denotes the numerical value zero, but has the logical value true (not zero).

Symbol	Description	Logical value	MapVal value
number	any valid real number		0
UNDF	undefined (result of an arithmetic error)	1	4
NA	not available	1	5
INF	$+\infty$	1	6
-INF	$-\infty$	1	7
ZERO	numerically indistinguishable from zero, but has the logical value of one.	1	8

Table 6.1: Extended values of the AIMMS language

Numerical behavior

AIMMS treats these special symbols as ordinary real numbers, and the results of the available arithmetic operations and functions on these symbols are defined. The values INF, -INF and ZERO are accessible by the user and are dealt with as expected: 1 + INF evaluates to INF, 1/INF to 0, 1 + ZERO to 1, etc. However, the values of INF and -INF are undetermined and therefore, it makes no

sense to consider INF/INF, −INF + INF, etc. These expressions are therefore evaluated to UNDF. A runtime error will occur if the value UNDF is assigned to an identifier.

The symbol ZERO

The symbol ZERO behaves like zero numerically, but its logical value is one. Using this symbol, you can make a distinction between the default value of 0 and an assigned ZERO. As an illustration, consider a distance matrix with distances between selected factory-depot combinations. A missing distance value evaluates to 0, and could mean that the particular factory-depot combination should not be considered. A ZERO value in that case could be used to indicate that the combination should be considered even though the corresponding distance is zero because the depot and factory happen to be one facility.

Expressions with 0 and ZERO

Whenever the values 0 and ZERO appear in the same expression with equal priority, the value of ZERO prevails. For example, the expressions 0 + ZERO or max(0,ZERO) will both result in a numerical value of ZERO. In this way, the logically distinctive effect of ZERO is retained as long as possible. You should note, however, that AIMMS will evaluate the multiplication of 0 with *any* special number to 0.

The symbol NA

The symbol NA can be used for missing data. The interpretation is "this number is not yet known". Any operation that uses NA and does not use the symbol UNDF will also produce the result NA. AIMMS can reason with this value as it propagates the value NA through its computations and assignments. The only exception is the condition in control flow statements where it must be known whether the result of that condition is equal to 0.0 or not, see also Section 8.3.

The symbol UNDF

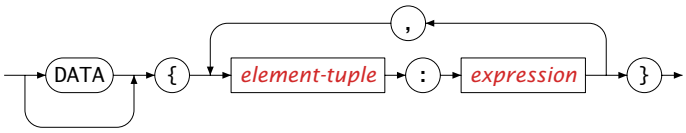
The symbol UNDF cannot be input directly by a user, but is, besides an error message, the result of an undefined or illegal arithmetic operation. For example, 1/ZERO, 0/0, (-2)^0.1 all result in UNDF. Any operation containing the UNDF symbol evaluates to UNDF.

6.1.2 List expressions

Element-value pairs

A *list* is a collection of *element-value pairs*. In a list a single element or range of elements is combined with a numerical, element-, or string-valued expression, separated by a *colon*. List expressions are the numerical extension of enumerated set expressions. The elements to which a value is assigned inside a list, are specified in exactly the same manner as in an enumerated set expression as explained in Section 5.1.1.

enumerated-list : *Syntax*



By preceding the list expression with the keyword DATA, it becomes a *constant* list expression, in a similar fashion as with constant set expressions (see Section 5.1.1). In a constant list expression, set elements need not be quoted and the assigned values must be constants. All other list expressions are symbolic, in which both the elements and the assigned values are the result of expression evaluation. *Constant versus symbolic*

The following assignments illustrate the use of list expressions. *Example*

- The following constant list expression assigns distances to tuples of cities.

```
Distance(i,j) := DATA {
    (Amsterdam, Rotterdam  ) : 85 [km] ,
    (Amsterdam, 'The Hague') : 65 [km] ,
    (Rotterdam, 'The Hague') : 25 [km]
} ;
```

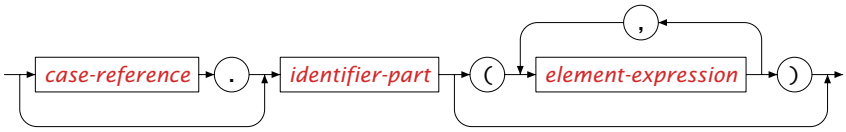
- The following symbolic list expression assigns a certain status to every node in a number of dynamically computed ranges.

```
NodeUsage(i) := {
    FirstNode            .. FirstNode + Batch - 1   : 'InUse'   ,
    FirstNode + Batch    .. FirstNode + 2*Batch - 1 : 'StandBy' ,
    FirstNode + 2*Batch  .. LastNode                : 'Reserve'
} ;
```

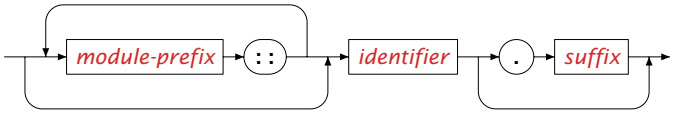
6.1.3 References

Sets, parameters and variables can be referred to by name resulting in a set-, set element-, string-valued, or numerical quantity. A reference can be scalar or multidimensional, and index positions may contain either indices or element expressions. By specifying a case reference in front, a reference can refer to data from cases that are not in memory. *References*

reference : *Syntax*



identifier-part :



Scalar versus indexed

A *scalar* set, parameter or variable has no indexing (dimension) and is referenced simply by using its identifier. Indexed sets, parameters and variables have dimensions equal to the number of indices.

Example

The right-hand sides of the following assignments are examples of references to scalar and indexed identifiers.

```
MainCity                := 'Amsterdam' ;

DistanceFromMainCity(i) := Distance( MainCity, i );

SecondNextCity(i)       := NextCity( NextCity(i) );

NextPeriodStock(t)      := Stock( t + 1 );
```

Undefined references

The last two references, which make use of lag and lead operators and element parameters, may sometimes be undefined. When used in an expression such undefined references evaluate to the empty set, zero, the empty element, or the empty string, depending on the value type of the identifier. When an undefined lag or lead operator or element parameter occurs on the left-hand side of an assignment, the assignment is skipped. For more details, refer to Section 8.2.

Referring to module identifiers

When your model contains one or more `Modules`, your model will be supplied multiple additional namespaces besides the global namespace, one for each module. Identifiers declared within a module are, by default, not contained in the global namespace. To refer to such identifiers outside the module, you have to prefix the identifier name with a module-specific prefix and the `::` namespace resolution operator. `Modules` and the namespace resolution operator are discussed in full detail in Section 35.4.

Referring to other cases

When a reference is preceded by a *case reference*, AIMMS will not retrieve the requested identifier data from the case in memory, but from the case file associated with the case reference. Case references are elements of the (predefined) set `AllCases`, which contains all the cases available in the data manager of AIMMS. The AIMMS User's Guide describes all the mechanisms that are available and functions that you can use to let an end-user of your application select one or more cases from the set of all available cases. Case referencing is useful when you want to perform advanced case comparison over multiple cases.

Example

The following computes the differences of the values of the variable Transport in the current case compared to its values in all cases in the set CurrentCaseSelection.

```
for ( c in CurrentCaseSelection ) do
    Difference(c,i,j) := c.Transport(i,j) - Transport(i,j) ;
endfor;
```

During execution, AIMMS will (temporarily) retrieve the values of Transport from all requested cases to compute the difference with the data of the current case.

6.1.4 Arithmetic functions

Standard functions

AIMMS provides the commonly used standard arithmetic functions such as the trigonometric functions, logarithms, and exponentiations. Table 6.2 lists the available arithmetic functions with their arguments and result, where x is an extended range arithmetic expressions, m, n are integer expressions, i is an index, l is a set element, I is a set identifier, and e is a scalar reference.

Functions and extended arithmetic

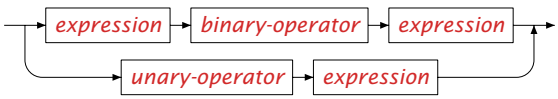
Special caution is required when one or more of the arguments in the functions are special symbols of AIMMS' extended range arithmetic. If the value of any of the arguments is UNDF or NA, then the result will also be UNDF or NA. If the value of any of the arguments is ZERO and the numerical value of the result is zero, the function will return ZERO.

6.1.5 Numerical operators

Using unary or binary numerical operators you can construct numerical expressions that consist of multiple terms and/or factors. The syntax follows.

Syntax

operator-expression :



Standard numerical operators

The order of precedence of the standard numerical operators in AIMMS is given in Table 6.3. Parentheses may be used to override the precedence order. Expression evaluation is from left to right.

Function	Meaning
Abs(x)	absolute value $\|x\|$
Exp(x)	e^x
Log(x)	$\log_e(x)$ for $x > 0$,UNDF otherwise
Log10(x)	$\log_{10}(x)$ for $x > 0$, UNDF otherwise
Max($x_1,\ldots,x_n$)	$\max(x_1,\ldots,x_n)$ $(n > 1)$
Min($x_1,\ldots,x_n$)	$\min(x_1,\ldots,x_n)$ $(n > 1)$
Mod(x_1,x_2)	$x_1 \bmod x_2 \in [0,x_2)$ for $x_2 > 0$ or $\in (x_2,0]$ for $x_2 < 0$
Div(x_1,x_2)	$x_1 \operatorname{div} x_2$
Sign(x)	$\operatorname{sign}(x) = +1$ if $x > 0$, -1 if $x < 0$ and 0 if $x = 0$
Sqr(x)	x^2
Sqrt(x)	$\sqrt{x}$ for $x \geq 0$, UNDF otherwise
Power(x_1,x_2)	$x_1^{x_2}$, alternative for x^y (see Section 6.1.5)
ErrorF(x)	$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}}\, dt$
Cos(x)	$\cos(x)$; x in radians
Sin(x)	$\sin(x)$; x in radians
Tan(x)	$\tan(x)$; x in radians
ArcCos(x)	$\arccos(x)$; result in radians
ArcSin(x)	$\arcsin(x)$; result in radians
ArcTan(x)	$\arctan(x)$; result in radians
Degrees(x)	converts x from radians to degrees
Radians(x)	converts x from degrees to radians
Cosh(x)	$\cosh(x)$
Sinh(x)	$\sinh(x)$
Tanh(x)	$\tanh(x)$
ArcCosh(x)	$\operatorname{arccosh}(x)$
ArcSinh(x)	$\operatorname{arcsinh}(x)$
ArcTanh(x)	$\operatorname{arctanh}(x)$
Card(I[, *suffix*])	cardinality of (suffix of) set, parameter or variable I
Ord(i)	ordinal number of index i in set I (see also Table 5.2)
Ord(l[, I])	ordinal number of element l in set I
Ceil(x)	$\lceil x \rceil$ = smallest integer $\geq x$
Floor(x)	$\lfloor x \rfloor$ = largest integer $\leq x$
Precision(x,n)	x rounded to n significant digits
Round(x)	x rounded to nearest integer
Round(x,n)	x rounded to n decimal places left ($n < 0$) or right ($n > 0$) of the decimal point
Trunc(x)	truncated value of x: Sign(x)*Floor(Abs(x))
NonDefault(e)	1 if e is not at its default value, 0 otherwise
MapVal(x)	MapVal value of x according to Table 6.1

Table 6.2: Intrinsic numerical functions of AIMMS

Operator	Meaning	Precedence
Unary		
+	positive	n/a
-	negative	n/a
Binary		
^	exponentiation	3 (high)
*	multiplication	2
/	division	2
+	addition	1
-	subtraction	1 (low)

Table 6.3: Numerical operators

Example

The expression

```
p1 + p2 * p3 / p4^p5
```

is parsed by AIMMS as if it had been written

```
p1 + [(p2 * p3) / (p4^p5)]
```

In general, it is better to use parentheses than to rely on the precedence and associativity of the operators. Not only because it prevents you from making unwanted mistakes, but also because it makes your intentions clearer.

Exponential operator

Special restrictions apply to the exponential operator "^". AIMMS accepts the following combinations of left-hand side operand (called the *base*), and right-hand side operand (called the *exponent*):

- a positive base with a real exponent,
- a negative base with an integer exponent,
- a zero base with a positive exponent, and
- a zero base with a zero exponent results in one (as controlled by the option power_0_0).

6.1.6 Numerical iterative operators

Arithmetic iterative operators

Iterative operators are used to express repeated arithmetic operations, such as summation, in a concise manner. The arithmetic iterative operators supported by AIMMS are listed in Table 6.4. The second column in this table refers to the required number of expression arguments following the binding domain argument, while the last column refers to the result of the operator in case of an empty domain.

Name	# Expr.	Computes over all elements in the domain	Default
Sum	1	the sum of the expression	0
Prod	1	the product of the expression	1
Count	0	the total number of elements in the domain	0
Min	1	the minimum value of the expression	INF
Max	1	the maximum value of the expression	-INF

Table 6.4: Arithmetic iterative operators

Compared expressions

The Min and Max operators return the minimum or maximum value of an expression. The allowed expressions are:

- numerical expressions, in which case AIMMS returns the lowest or highest numerical values,
- string expressions, in which case AIMMS returns the strings which are first or last with respect to the normal alphabetic ordering, and
- element expressions, in which case AIMMS returns the elements with the lowest or highest ordinal numbers (see also Section 5.2.1).

Example

The following assignments are valid examples of the use of the arithmetic iterative operators.

```
NumberOfRoutes      := Count( (i,j) | Distance(i,j) ) ;
NettoTransport(i)   := Sum( j, Transport(i,j) - Transport(j,i) ) ;
MaximumTransport(i) := Max( j, Transport(i,j) ) ;
```

6.1.7 Statistical functions and operators

Distributions

AIMMS provides the most commonly used distributions. They are listed in Table 6.5, together with the required type of arguments and a description of the result. You can find a more detailed description of these distributions in Appendices A.1 and A.2. When called as functions inside your model, they behave as random number generators.

Setting the seed

You can set the seed of the random number generators for all distributions using the execution option seed. By setting the seed explicitly you can guarantee that your model results are reproducible.

Cumulative distributions and their derivatives

Each distribution in Table 6.5 can be used as an argument for four operators: DistributionCumulative and DistributionInverseCumulative, and their derivatives DistributionDensity and DistributionInverseDensity. In the explanation below it is assumed that $\alpha \in [0,1]$, $x \in (-\infty,\infty)$, and X a random variable distributed according to the given distribution *distr*.

- DistributionCumulative(*distr*,x) computes the probability $P(X \leq x)$.

Distribution	Meaning
`Binomial`(p, n)	Binomial distribution with probability p and number of trials n
`NegativeBinomial`(p, r)	Negative Binomial distribution with probability p and number of successes r
`Poisson`(λ)	Poisson distribution with rate λ
`Geometric`(p)	Geometric distribution with probability p
`HyperGeometric`(p, n, N)	Hypergeometric distribution with initial probability of success p, number of trials n and population size N
`Uniform`(*min*,*max*)	Uniform distribution with lower bound *min* and upper bound *max*
`Triangular`(β, *min*, *max*)	Triangular distribution with shape β, lower bound *min*, and upper bound *max*, where $\beta = (x_{\text{peak}} - min)/(max - min)$
`Beta`(α, β, *min*, *max*)	Beta distribution with shapes α, β, lower bound *min*, and upper bound *max*
`LogNormal`(β, *min*, s)	Lognormal distribution with shape β, lower bound *min*, and scale s
`Exponential`(*min*, s)	Exponential distribution with lower bound *min* and scale s
`Gamma`(β, *min*, s)	Gamma distribution with shape β, lower bound *min*, and scale s
`Weibull`(β, *min*, s)	Weibull distribution with shape β, lower bound *min*, and scale s
`Pareto`(β, l, s)	Pareto distribution with shape β, location l, and scale s (lower bound = $l + s$)
`Normal`(μ, σ)	Normal distribution with mean μ and standard deviation σ
`Logistic`(μ, s)	Logistic distribution with mean μ and scale s
`ExtremeValue`(l, s)	Extreme Value distribution with location l and scale s

Table 6.5: Distributions available in AIMMS

- `DistributionInverseCumulative`(*distr*,α) computes the smallest x such that the probability $P(X \le x) \ge \alpha$, except for $\alpha = 0$ which returns the lowest possible value for X.
- `DistributionDensity`(*distr*,x) computes for continuous distributions the probability density $\lim_{\alpha \downarrow 0} P(x \le X \le x+\alpha)/\alpha$. For discrete distributions, the operator is only defined for integer values of x and returns $P(X = x)$.
- `DistributionInverseDensity`(*distr*,α) is the derivative of `DistributionInverseCumulative`. For more details you are referred to Appendix A.3.

Use in constraints

For the continuous distributions in Table 6.5 AIMMS can compute the derivatives of the cumulative and inverse cumulative distribution functions. As a consequence, you may use these functions in the constraints of a nonlinear model when the second argument is a variable.

Example

The following statements demonstrate how the distributions can be used to perform statistical tasks.

1. Draw a random number from a distribution.

```
Draw := Normal(0,1);
Draw := Uniform(LowestValue, HighestValue);
```

2. Compute the probability of at most 10 successes out of 50 trials, with a 0.25 probability of success.

```
Probability := DistributionCumulative( Binomial(0.25,50), 10 );
```

3. Compute a two-sided 90% confidence interval of a Normal(0,1) distribution.

```
LeftBound  := DistributionInverseCumulative( Normal(0,1), 0.05);
RightBound := DistributionInverseCumulative( Normal(0,1), 0.95);
```

Statistical operators

The distributions, listed in Table 6.5, make it possible for you to execute a stochastic experiment based on your model representation. In order to analyze the subsequent results, AIMMS provides a number of statistical iterative operators which are listed in Table 6.6. The second column in this table refers to the required number of expression arguments following the binding domain argument. For the most common sample operators, AIMMS provides distribution operators to calculate the corresponding expected values, assuming the sample is drawn from a given distribution. These distribution operators are listed in Table 6.7. A more detailed description of these operators is provided in Appendix A.

Example

Assume that `p` is an index into a set that has been used to index a number of experiments resulting in observables `x(p)` and `y(p)`. Then the following assignments demonstrate the use of the statistical operators in AIMMS.

```
MeanX         := Mean(p, x(p));
MeanX         := Mean(p | x(p), x(p));
DeviationX    := SampleDeviation(p, x(p));
CorrelationXY := Correlation(p, x(p), y(p));
```

In case the x values are drawn from a `Binomial(0.6,8)` distribution the expected value of `MeanX` is given by

```
ExpectedMeanX := DistributionMean(Binomial(0.6,8));
```

Name	# Expr.	Computes over all elements in the domain
Mean,	1	the (arithmetic) mean
GeometricMean	1	the geometric mean
HarmonicMean	1	the harmonic mean
RootMeanSquare	1	the root mean square
Median	1	the median
SampleDeviation	1	the standard deviation of a sample
PopulationDeviation	1	the standard deviation of a population
Skewness	1	the coefficient of skewness
Kurtosis	1	the coefficient of kurtosis
Correlation	2	the correlation coefficient
RankCorrelation	2	the rank correlation coefficient

Table 6.6: Statistical sample operators

Units of measurement

For all distributions, the units of measurement (see also Chapter 32) of parameters and result should be consistent. The unit relationships for each distribution are described in Appendix A in full detail. In the presence of units of measurement within your model, AIMMS will perform a unit consistency check.

Histogram support

For easy visualization of statistical data, AIMMS offers support for creating histograms based on a large collection of observed values. Through a number of predefined procedures and functions, AIMMS allows you to flexibly create interval-based histogram data, which can easily be displayed, for instance, using the standard (graphical) AIMMS bar chart object. For further information about creating and displaying histograms, as well as an illustrative example, you are referred to section A.6 in the Appendix.

Combinatoric functions

In addition to the distribution and statistical operators listed above, AIMMS also offers support for the most common combinatoric calculations. Table 6.8 contains the list of combinatoric functions that are available in AIMMS.

6.1.8 Financial functions

Financial functions

AIMMS provides an extensive library of financial functions for a variety of financial applications. The available functions can be classified as follows.

- Functions for the computation of the depreciation of assets using various methods such as fixed-declining balance method, double-declining balance method, etc.
- Functions for computing various quantities regarding investments that consist of a series of constant or variable periodic cash flows. The computed quantities include present value, net present value, future value, internal rate of return, interest and principal payments, etc.

Name	Computes for a given distribution
DistributionMean	the (arithmetic) mean
DistributionDeviation	the (standard) deviation
DistributionVariance	the variance (the square of the deviation)
DistributionSkewness	the coefficient of skewness
DistributionKurtosis	the coefficient of kurtosis

Table 6.7: Statistical distribution operators

- Functions for computing various security-related quantities of, for instance, discounted securities, securities that pay periodic interest and securities that pay interest at maturity. The computed quantities include yield, interest rate, redemption, price, accrued interest, etc.

Consult the online function reference

The precise description of all financial functions available in AIMMS is not included in this Language Reference. You can find a complete list of the available financial functions on pages **??** and further of the AIMMS Function Reference. The Function Reference provides a description as well as the prototype of every financial function present in AIMMS.

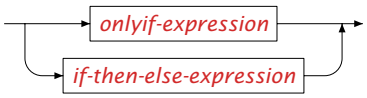
6.1.9 Conditional expressions

Two conditional expressions

There are two ways to specify expressions that adopt different values depending on one or more logical conditions. The ONLYIF operator is the simpler and operates as it sounds. The IF-THEN-ELSE expression is more powerful in its ability to distinguish several cases.

Syntax

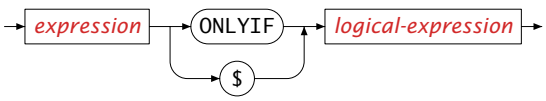
conditional-expression :



The ONLYIF operator

The simplest way of specifying a conditional expression is to use the ONLYIF operator. Its syntax is given by

onlyif-expression :



The ONLYIF expression evaluates to the arithmetic expression in the first argu-

Function	Meaning
`Factorial`(n)	$n!$
`Combination`(n,m)	$\binom{n}{m}$
`Permutation`(n,m)	$m! \cdot \binom{n}{m}$

Table 6.8: Combinatoric functions

ment if the logical condition of the second argument is true. Otherwise, it is zero. The "`$`" symbol can be used as a synonym for the `ONLYIF` operator.

A simple example of the use of the `ONLYIF` operator is given by the assignment *Example*

```
AverageVelocity := (Distance / TravelTime) ONLYIF TravelTime ;
```

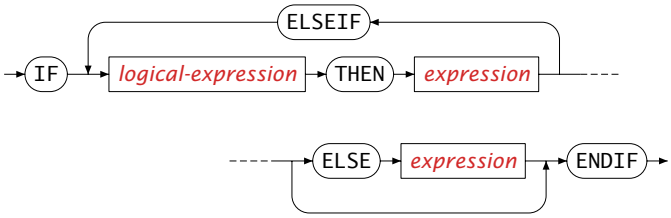
or equivalently, using the `$` operator,

```
AverageVelocity := (Distance / TravelTime) $ TravelTime ;
```

Both expressions evaluate to `Distance / TravelTime` if `TravelTime` assumes a nonzero value, or to zero otherwise. In Section 12.2 you will see that this particular expression can be written even more concisely using the sparsity modifier "`$`".

A much more flexible way for specifying conditional expressions is given by the `IF-THEN-ELSE` operator. The syntax of the `IF-THEN-ELSE` expression is given below. *IF-THEN-ELSE expressions*

if-then-else-expression: *Syntax*



The `IF-THEN-ELSE` expression works like a *switch statement*—a series of `ELSEIF`s can be used to denote numerous special cases. The value of the `IF-THEN-ELSE` expression is the first numerical expression for which the corresponding logical condition is true. If none of the conditions are true, then the value will be the numerical expression after the `ELSE` keyword if present or zero otherwise. *Explanation*

A simple illustration of the use of the IF-THEN-ELSE construction is given by the assignments *Example*

```
AverageVelocity := IF TravelTime THEN Distance / TravelTime ENDIF ;
```

which is equivalent to the ONLYIF expression above. A more elaborate example is given by the assignment

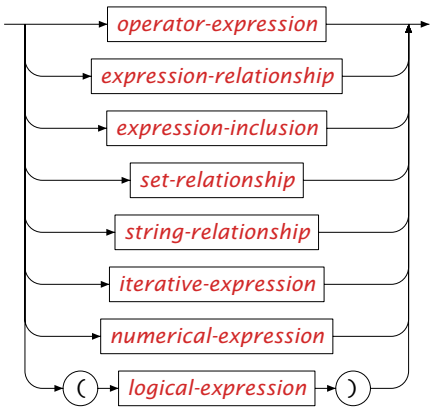
```
WeightedDistance(i) :=
    IF     Distance(i) <= 100 THEN Distance(i)
    ELSEIF Distance(i) <= 200 THEN (100 + Distance(i)) / 2
    ELSEIF Distance(i) <= 300 THEN (250 + Distance(i)) / 3
    ELSE   550 / 3
    ENDIF ;
```

The expression takes the value associated with the first logical expression that is true.

6.2 Logical expressions

Logical expressions are expressions that evaluate to a logical value—0.0 for false and 1.0 for true. AIMMS supports several types of logical expressions. *Logical expressions*

logical-expression :



As AIMMS permits numerical expressions as logical expressions it is important to discuss how numerical expressions are interpreted logically, and how logical expressions are interpreted numerically. Numerical expressions that evaluate to zero (0.0) are false, while all others (including ZERO, NA and UNDF) are true. A false logical expression evaluates to zero (0.0), while a true logical expression evaluates to one (1.0). If one or more of the operands of a logical operator is UNDF or NA, the numerical value is also UNDF or NA. Note that AIMMS will not accept expressions that evaluate to UNDF or NA in the condition in control flow *Numerical expressions as logical*

statements, where it must be known whether the result of that condition is equal to 0.0 or not (see also Section 8.3).

Example

Table 6.9 illustrates the different interpretation of a number of numerical and logical expressions as either a numerical or a logical expression. See also Table 6.10 for the results associated with the AND operator.

Expression	Numerical value	Logical value
3*(2 > 1)	3.0	true
3*(1 > 2)	0.0	false
(1 < 2) + (2 < 3)	2.0	true
max((1 < 2),(2 < 3))	1.0	true
2 AND 0.0	0.0	false
2 AND ZERO	1.0	true
2 AND NA	NA	true
UNDF < 0	UNDF	true

Table 6.9: Numerical and logical values

6.2.1 Logical operator expressions

Unary and binary logical operators

AIMMS supports the unary logical operator NOT and the binary logical operators AND, OR, and XOR. Table 6.10 gives the logical results of these operators for zero and nonzero operands.

Operands		Result			
a	b	a AND b	a OR b	a XOR b	NOT a
0	0	0	0	0	1
0	nonzero	0	1	1	1
nonzero	0	0	1	1	0
nonzero	nonzero	1	1	0	0

Table 6.10: Logical operators

Precedence order

The precedence order of these operators from highest to lowest is given by NOT, AND, OR, and XOR respectively. Whenever the precedence order is not immediately clear, it is advisable to use parentheses. Besides preventing unwanted mistakes, it also make your model easier to understand and maintain.

Example

The expression

```
NOT a AND b XOR c OR d
```

is parsed by AIMMS as if it were written

```
((NOT a) AND b) XOR (c OR d).
```

Execution order

Due to the sparse execution system underlying AIMMS it is not guaranteed that logical expressions containing binary logical operators are executed in a strict left-to-right order. If you are a C/C++ programmer (where logical conditions are executed in a strict left-to-right order), you should take extra care to ensure that your logical conditions do not depend on this assumption.

6.2.2 Numerical comparison

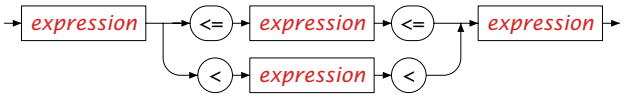
Numerical comparison

Numerical relationships compare two numerical expressions, using one of the relational operators =, <>, >, >=, <, or <=. Numerical inclusions are equivalent to two numerical relationships, and indicate whether a given expression lies within two bounds.

Syntax

expression-relationship :

expression-inclusion :



Numerical tolerances

For two real numbers x and y the result of the comparison $x \gtrless y$, where $\gtrless$ denotes any relational operator, depends on two tolerances

- Equality_Absolute_Tolerance (denoted as ε_a), and
- Equality_Relative_Tolerance (denoted as ε_r).

You can set these tolerances through the options dialog box. Their default values are 0 and 10^{-13}, respectively. If the number $\varepsilon_{x,y}$ is given by the formula

$$\varepsilon_{x,y} = \max(\varepsilon_a, \varepsilon_r \cdot x, \varepsilon_r \cdot y),$$

then the relational operators evaluate as shown in the Table 6.11.

AIMMS expression	Evaluates as
$x=y$	$\|x-y\| \leq \varepsilon_{x,y}$
$x<>y$	$\|x-y\| > \varepsilon_{x,y}$
$x<=y$	$x-y \leq \varepsilon_{x,y}$
$x<y$	$x-y < -\varepsilon_{x,y}$

Table 6.11: Interpretation of numerical tolerances

Comparison for extended arithmetic

For any combination of an ordinary real number with one of the special symbols ZERO, INF, and -INF, the relational operators behave as expected. If any of the operands is either NA or UNDF, relationships other than = and <> also evaluate to NA or UNDF and hence, as a logical expression, to true. In addition, the logical expressions INF = INF and -INF = -INF evaluate to true.

Testing for zero value

One can formulate numerous logical expressions to test for a zero value, and one should be clear on the desired result. The following example makes the point.

```
p_inv(i)             := 1 / p(i);
p_inv(i | p(i))      := 1 / p(i);
p_inv(i | p(i) <> 0) := 1 / p(i);
```

The first assignment will produce a runtime error when p(i) assumes a value of 0 or ZERO. The second assignment will filter out the 0's, but not the ZERO values because ZERO evaluates to the logical value "true". The last assignment will never produce runtime errors, because of the *numerical* comparison to 0.

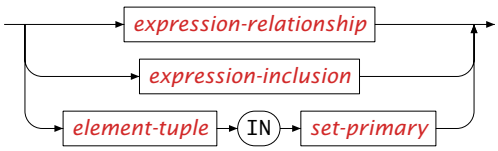
6.2.3 Set and element comparison

Set relationships

AIMMS features very powerful logical set comparison operators. Not only can sets and their elements be compared using relational operators, but you can also check for set membership with the IN operator.

Syntax

set-relationship :



Element relationship and inclusion

Set elements that lie in the same set can be compared according to their relative position inside that set. You can also compare the positions of arbitrary set element expressions, as long as AIMMS is able to determine a unique domain set in which the comparison has to take place. The allowed relational

operators are =, <>, <, <=, >, and >=. As with numerical expression, AIMMS also allows you to specify an inclusion relationship as a form of repeated comparison to verify whether an element lies within two boundary elements.

Element comparison

The relational operators for element relationships are conveniently defined in terms of the Ord function. Let S be a simple set, i and j indices or element parameters in S, ± any of the lag or lead operators +, ++, - or --, m and n integer expressions, and $\gtrless$ one of the operators =, <>, <, <=, >, or >=. The relational operators $\gtrless$ have the following definition for set elements, provided that the set elements on both sides of the relational operator exist.

$$\mathtt{i} \pm m \gtrless \mathtt{j} \pm n \quad \Leftrightarrow \quad \begin{cases} \mathtt{i} \pm m \text{ and } \mathtt{j} \pm n \text{ are both defined, and} \\ \mathtt{Ord}(\mathtt{i} \pm \mathtt{m}, \mathtt{S}) \gtrless \mathtt{Ord}(\mathtt{j} \pm \mathtt{n}, \mathtt{S}) \end{cases}$$

Note that this type of relational expression evaluates to “false” if one or both of the operands do not refer to existing set elements.

Compare within the same set

Only elements that lie in the same set are comparable using the <, <=, >, and >= operators. The = and <> operators can also be used when the operands merely share the same root set.

Example

The following set assignments demonstrate the correct use of element comparisons.

```
FuturePeriods := { t in Periods | CurrentPeriod <= t <= PlanningHorizon } ;

BandMatrix := { (i,j) | i - BandWidth <= j <= i + BandWidth } ;
```

Set membership

Set membership can be tested using the IN operator. This operator checks whether a set element or an element tuple on the left-hand side is a member of the set expression on the right-hand side. Both operands must have the same root set.

Example

Assume that all one-dimensional sets in the following two assignments share the same root set Cities. Then these statements illustrate the correct use of the logical IN operator.

```
NeighborhoodRoutes := { (i,j) in Routes | j in NeighborhoodCities(i) } ;
ExcludedCities     := { i in ( SmallCities + ForeignCities ) } ;
```

Set comparisons

Sets can be logically compared using any of the relational operators =, <>, <, <=, > and >=. The inequality operators denote the usual subset relationships. They replace the standard ”contained in” operators $\subsetneq$, $\subseteq$, $\supsetneq$ and $\supseteq$ which are not available on many keyboards.

The following statement illustrates a logical set comparison operator. *Example*

```
IF ( RoutesWithTransport <= NeighborhoodRoutes ) THEN
    DialogMessage( "Solution only contains neighborhood transports" );
ENDIF;
```

6.2.4 String comparison

Besides their use for comparison of numerical, element- and set-valued expressions, the relational operators =, <>, <, <=, >, and >= can also be used for string comparison. When used for string comparison, AIMMS employs the usual lexicographical ordering. String comparison in AIMMS is case sensitive by default, i.e. strings that only differ in case are considered to be unequal. You can modify this behavior through the option Case_Sensitive_String_Comparison. *String comparison*

All the following string comparisons evaluate to true. *Examples*

```
"The city of Amsterdam" <> "the city of amsterdam"     ! Note case
"The city of Amsterdam" <> "The city of Amsterdam "    ! Note last space
"The city of Amsterdam" <  "The city of Rotterdam"
```

6.2.5 Logical iterative expressions

Logical iterative operators verify whether some or all elements in a domain satisfy a certain logical condition. Table 6.12 lists all logical iterative operators supported by AIMMS. The second column in this table refers to the required number of expression arguments following the binding domain argument. *Logical iterative operators*

Name	# Expr.	Meaning
Exists	0	true if the domain is not empty
Atleast	1	true if the domain contains at least n elements
Atmost	1	true if the domain contains at most n elements
Exactly	1	true if the domain contains at exactly n elements
ForAll	1	true if the expression is true for all elements in the domain

Table 6.12: Logical iterative operators

The following statements illustrate the use of some of the logical iterative operators listed in Table 6.12. *Example*

```
MultipleSupplyCities := { i | Atleast( j | Transport(i,j), 2 ) } ;

IF ( ForAll( i, Exists( j | Transport(i,j) ) ) ) THEN
    DialogMessage( "There are no cities without a transport" );
ENDIF ;
```

6.3 Operator precedence

Combined precedence order

In the previous sections we have introduced unary and binary operators for several types of expressions, together with their relative precedence order. Table 6.13 provides an overview of all of them. The last column lists the expression types in which the operator is used, where the letters "N", "L", "E", and "S" stand for *N*umerical, *L*ogical, set *E*lement and *S*et expressions, respectively.

Precedence	Name	Type
14	ONLYIF $	N
13	^	N
12	+ - (unary)	N
11	* /	N,S
10	+ - ++ -- (binary)	N,E,S
9	CROSS	S
8	IN	L
7	< <= > >= = <>	L
6	NOT	L
5	AND	L
4	OR	L
3	XOR	L
2	|	S
1	IF THEN ELSEIF ELSE ENDIF	N

Table 6.13: Operator precedence (highest to lowest)

6.4 MACRO declaration and attributes

Macro facility

The MACRO facility offers a mechanism for parameterizing expressions. Macros are useful for enhancing the readability of models, and avoiding inconsistencies in frequently used expressions.

Declaration and attributes

Macros are declared as ordinary identifiers in your model. They can have arguments. The attributes of a Macro declaration are listed in Table 6.14.

The Definition attribute

The Definiton attribute of a macro declaration is the replacement text that is substituted when a macro is used in the model text. The (optional) Arguments of a macro must be scalar entities. Unlike function arguments, however, you do not have to declare Macro arguments as local identifiers. The Definition of a macro must be a valid expression in its arguments.

Attribute	Value-type	See also page
Text	*string*	19
Arguments	*argument-list*	
Comment	*comment string*	19
Definition	*expression*	34

Table 6.14: Macro attributes

Example

When you define a macro with arguments, the actual replacement text depends on the arguments that are supplied to it, as illustrated in the following example. Using the macro declaration

```
Macro MyAverage {
    Arguments  : (dom, expr);
    Definition : Sum(dom, expr) / Count(dom);
}
```

the assignments

```
   AverageTransport   := MyAverage( (i,j), Transport(i,j) );
   AverageNZTransport := MyAverage( (i,j) | Transport(i,j),  Transport(i,j) );
```

are compiled as if they read:

```
   AverageTransport   := Sum( (i,j), Transport(i,j) ) / Count( (i,j) );
   AverageNZTransport :=
          Sum  ( (i,j) | Transport(i,j),  Transport(i,j) ) /
          Count( (i,j) | Transport(i,j) );
```

Expression substitution

When you use a macro with arguments, the actual arguments *must* be valid expressions. As a result, there is no need to add additional braces to the replacement text of the macro, like, for instance, in the C programming language. The following example illustrates this point.

```
Macro MyMult {
    Arguments  : (x,y);
    Definition : x*y;
}
```

Using this macro, the expression

```
    a + MyMult(b+c,d+e) + f
```

will evaluate to

```
    a + ((b+c)*(d+ e)) + f
```

instead of

```
    a + b + c*d + e + f
```

Macro versus defined parameters

In many execution statements you have a choice to use either macros or defined parameters as a mechanism to replace complicated expressions by descriptive names. While a macro is purely substituted by its replacement text, the current value of a defined parameter is stored and looked up when needed. When deciding whether to use a macro or a defined parameter, you should consider both storage and computational consequences. Macros are recomputed every time they are referenced, and therefore there may be an unnecessary time penalty if the macro is called with identical arguments in more than one place within your model. When storage considerations are important, a macro may be attractive since it does not introduce additional parameters.

Macro versus defined variables

You should also consider your choices when you use a macro with variables as arguments in a constraint. In this case, you also have the option to use a defined variable, or a defined `Inline` variable (see also Section 14.1). The following considerations are of interest.

- A macro can produce different expressions of the same structure for different identifier arguments, but does not allow you to specify a domain restriction that will reduce the number of generated columns in the matrix.
- Defined and `Inline` variables support an index domain to restrict the number of generated columns, but only allow an expression in terms of fixed identifiers. Compared to a macro or an `Inline` variable, the number of rows and columns increases for a defined variable, but if the variable is referenced more than once in the other constraints, it will result in a smaller number of nonzeros.
- An advantage of variables (both defined and `Inline`) over macros is that their final values are stored by AIMMS, and can be retrieved in other execution statements or in the graphical user interface, whereas a macro has to be recomputed all the time.

Chapter 7

Execution of Nonprocedural Components

This chapter

The collection of all set and parameter definitions form a system of functional relationships which AIMMS keeps up-to-date automatically. This chapter discusses the dependency structure of the system, the kind of expressions and statements allowed inside the definitions, and the way in which the relationships are re-computed.

Spreadsheets

The nonprocedural execution mechanism discussed in this chapter resembles the execution of spreadsheets. Definitions can be placed in any order by the model builder, but the logical order of execution is determined by the system. As a result, you can easily formulate spreadsheet-based applications in the AIMMS modeling language by merely using definitions for sets and parameters. Of course, the modeling language in AIMMS goes beyond the modeling paradigm of spreadsheets, as AIMMS also offers procedural execution which is found in programming languages but not in spreadsheets.

7.1 Dependency structure of definitions

Dependency graph

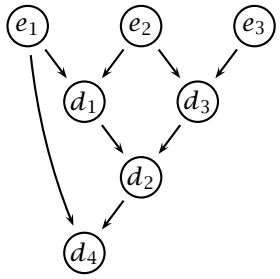
The definitions inside the declarations of global sets and parameters together form a system of interrelated functional relationships. AIMMS automatically determines the dependency between the defined identifiers and the inputs that are used inside these relationships. Such dependencies can be depicted in the form of a directed graph, called the *dependency graph*. From this dependency graph, AIMMS determines the minimal set of identifiers that must be recomputed—and in which order—to get the total system of functional relationships up-to-date.

Example

Consider the system of definitions

$$
\begin{aligned}
d_1 &\equiv e_1 + e_2 \\
d_2 &\equiv d_1 + d_3 \\
d_3 &\equiv e_2 + e_3 \\
d_4 &\equiv e_1 + d_2 .
\end{aligned}
$$

Its dependency graph, with identifiers as nodes and dependencies as directed arcs, looks as follows. Note that a change to the input parameter e_3, for in-



stance, requires the re-computation of the defined parameters $d_2, \ldots, d_4$—but not of d_1—to update the entire system.

Dependencies must be a-cyclic

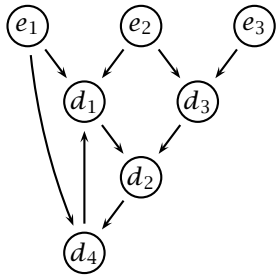
The dependency graph associated with the set and parameter definitions must be a-cyclic, i.e. must not contain circular references. In this case, every change to one or more input parameters of defined sets or parameters will result in a *finite* sequence of assignments to update the system. If the dependency graph is cyclic, a simultaneous system of relations will result. Such a system may not have a (unique) solution, and can only be solved by a specialized solver. Simultaneous systems of relations are handled inside AIMMS through the use of constraints and mathematical programs.

Example

An illegal set of dependencies results if the definition of d_1 in the last example is changed as follows.

$$d_1 \equiv d_4 + e_1 + e_2.$$

This results in the following cyclic dependency graph. Now, a change to any



of the input parameters $e_1, \ldots, e_3$ will result in a simultaneous system for the parameters d_1, d_2 and d_4.

AIMMS will check

AIMMS computes the dependency structure between the parameter and set definitions while compiling your model. If AIMMS detects a cyclic dependency, an error will result, because AIMMS can, in general, not deal with cyclic dependencies without relying on specialized numerical solvers. In that case you need to remove the cyclic dependencies before you can execute the model without further modifications. If you are unable to remove the cyclic dependencies, you have essentially two alternatives. You can either formulate a mathematical program, or define your own solution method inside a procedure.

Variables for simultaneous systems

The cyclic system can be turned into a mathematical program by changing the parameters with cyclic definitions into variables. This results in a simultaneous system of equalities which can be solved through a SOLVE statement. The declaration of mathematical programs is discussed in Chapter 15.

Feedback loops

The alternative is to implement a customized solution procedure by breaking the simultaneous system into a simulation with a feedback loop linking inputs and outputs. To accomplish this, you must first remove the cyclic definitions from the declarations, and then add a procedure that implements the feedback loop. If you have sufficient knowledge of the process you are describing, this route may result in fast convergence behavior.

Dependency is global only

AIMMS only allows a definition for globally declared sets and parameters. Consequently, a single global dependency graph suffices to express the functional relationships between all defined sets and parameters.

Dependency is symbolic

In addition, the dependency structure between set and parameter definitions is purely based on symbol references. As a result, AIMMS' automatic evaluation scheme will always recompute an indexed (output) parameter depending on an indexed (input) parameter *in its entirety*, even when only a single input value has changed.

Inefficiency may occur

This evaluation behavior may lead to severe inefficiencies when you use a high-dimensional defined parameter that is re-evaluated repeatedly during the execution of a loop in your model. In such cases it is advisable to refrain from using a definition for such a parameter, but replace it by one or more assignments at the appropriate places in your model. This issue is discussed in full detail in Section 13.2.3.

7.2 Expressions and statements allowed in definitions

Complicated definitions

In most applications, the functional relationship between input and output identifiers in the definition of a set or a parameter can be expressed as an ordinary set-valued, set element-valued or numerical expression. In rare occa-

sions where a functional relationship cannot be written as a single symbolic statement, a function or procedure can be used instead.

Allowed definitions

In summary, you may use one of the following items in set and parameter definitions:

- a set-valued expression,
- an element-valued expression,
- a numerical expression,
- a call to a function, or
- a call to a procedure.

Limited self-referencing allowed

Under some conditions, expressions used in the definition of a particular parameter can contain references to the parameter itself. Such self-referencing is allowed if the *serial* computation of the definition over all elements in the index domain of the parameter does not result in a cyclic reference to the parameter at the individual level. This is useful, for instance, when expressing stock balances in a functional manner with the use of lag operators.

Example

The following definition illustrates a valid example of a self-reference.

```
Parameter Stock {
    IndexDomain  : t;
    Definition   : {
        if ( t = FirstPeriod ) then BeginStock
            else Stock(t-1) + Supply(t) - Demand(t) endif
    }
}
```

If `t` is an index into a set `Periods = {0..3}`, and `FirstPeriod` equals `0`, then at the individual level the assignments with self-references are:

```
    Stock(0) := BeginStock ;
    Stock(1) := Stock(0) + Supply(1) - Demand(1) ;
    Stock(2) := Stock(1) + Supply(2) - Demand(2) ;
    Stock(3) := Stock(2) + Supply(3) - Demand(3) ;
```

Since there is no cyclic reference, the above definition is allowed.

Functions and procedures

You can use a call to either a function or a procedure to compute those definitions that cannot be expressed as a single statement. If you use a procedure, then only a single output argument is allowed. In addition, the procedure cannot have any side-effects on other global sets or parameters. This means that no direct assignments to other global sets or parameters are allowed.

Arguments and global references

The identifiers referenced in the actual arguments of a procedure call, as well as the global identifiers that are referenced in the body of the procedure, will be considered as input parameters for the computation of the current definition. That is, data changes to any of these input identifiers will trigger the re-execution of the procedure to make the definition up-to-date. The same applies to functions used inside definitions.

Examples

The following two examples illustrate the use of functions and procedures in definitions.

- Consider a function TotalCostFunction which has a single argument for individual cost coefficients. Then the following declaration illustrates a definition with a function reference.

  ```
  Parameter TotalCost {
      Definition : TotalCostFunction( CostCoefficient );
  }
  ```

  AIMMS will consider the actual argument CostCoefficient, as well any other global identifier referenced in the body of TotalCostFunction as input parameters of the definition of TotalCost.
- Similarly, consider a procedure TotalCostProcedure which performs the same computation as the function above, but returns the result via a (single) output argument. Then the following declaration illustrates an equivalent definition with a procedure reference.

  ```
  Parameter TotalCost {
      Definition : TotalCostProcedure( CostCoefficient, TotalCost );
  }
  ```

One procedure for several definitions

Whenever the values of a number of identifiers are computed simultaneously inside a single procedure without arguments, then this procedure must be referenced inside the definition of each and all of the corresponding identifiers. If you do not reference the procedure for all corresponding identifiers, a compile-time error will result. All other global identifiers used inside the body of the procedure count as input identifiers.

Example

Consider a procedure ComputeCosts which computes the value of the global parameters FixedCost(m,p) and VariableCost(m,p) simultaneously. Then the following example illustrates a valid use of ComputeCosts inside a definition.

```
Parameter FixedCost {
    IndexDomain  : (m,p);
    Definition   : ComputeCosts;
}
Parameter VariableCost {
    IndexDomain  : (m,p);
    Definition   : ComputeCosts;
}
```

Omitting ComputeCosts in either definition will result in a compile-time error.

7.3 Nonprocedural execution

Execution based on definitions

Execution based on definitions is typically not controlled by the user. It takes place automatically, but only when up-to-date values of defined sets or parameters are needed. Basically, execution can be triggered automatically from within:

- the body of a function or procedure, or
- an object in the graphical user interface.

Relating definitions and procedures

Consider a set or a parameter with a definition which is referenced in an execution statement inside a function or a procedure. Whenever the value of such a set or parameter is not up-to-date due to previous data changes, AIMMS will compute its current value just prior to executing the corresponding statement. This mechanism ensures that, during execution of functions or procedures, the functional relationships expressed in the definitions are always valid.

Lazy evaluation

During execution AIMMS minimizes its efforts and updates only those values of defined identifiers that are needed at the current point of execution. Such *lazy evaluation* can avoid unnecessary computations and reduces computational time significantly when the number of dependencies is large, and when relatively few dependencies need to be resolved at any particular point in time.

GUI requests

For the graphical objects in an end-user interface you may specify whether the data in that object must be up-to-date at all times, or just when the page containing the object is opened. AIMMS will react accordingly, and automatically update all corresponding identifiers as specified.

The set CurrentAutoUpdatedDefinitions

Which definitions are automatically updated in the graphical user interface whenever they are out-of-date, is determined by the contents of the predefined set `CurrentAutoUpdatedDefinitions`. This set is a subset of the predefined set `AllIdentifiers`, and is initialized by AIMMS to the union of the sets `AllDefinedSets` and `AllDefinedParameters` by default.

Exclude from auto-updating

To prevent auto-updating of particular identifiers in your model, you should remove such identifiers from the set `CurrentAutoUpdatedDefinitions`. You can change its contents either from within the language or from within the graphical user interface. Typically, you should exclude those identifiers from auto-updating whose computation takes a long time to finish. Instead of waiting for their computation on every input change, it makes much more sense to collect all input changes for such identifiers and request their re-computation on demand.

Requesting updates

All identifiers that are not contained in CurrentAutoUpdatedDefinitions must be updated manually under your control. AIMMS provides several mechanisms:

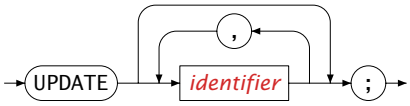
- you can call the UPDATE statement from within the language, or
- you can attach update requests of particular identifiers as actions to buttons and pages in the end-user interface.

The UPDATE statement

The UPDATE statement can be used to update the contents of one or more identifiers during the execution of a procedure that is called by the user. In this way, selected identifiers which are shown in the graphical user interface and not kept up-to-date automatically, can be made up-to-date once the procedure is activated by the user.

Syntax

update-statement :



Allowed identifiers

The following selections of identifiers are allowed in the UPDATE statement:

- identifiers with a definition,
- identifiers associated with a structural section in the model-tree, and
- identifiers in a subset of the predefined set AllIdentifiers.

Example

The following execution statement inside a procedure will trigger AIMMS to update the values of the identifiers FixedCost, VariableCost and TotalCost upon execution.

```
Update FixedCost, VariableCost, TotalCost;
```

Part III

Procedural Language Components

Chapter 8

Execution Statements

This chapter

This chapter describes the interaction between the nonprocedural and procedural execution mechanisms in AIMMS. In addition, the major execution statements like the assignment statement, the flow control statements, and the OPTION statement are discussed. Other important execution statements such as procedure calls, the SOLVE statement, as well as data control and display statements are discussed in various other chapters.

8.1 Procedural and nonprocedural execution

Execution based on definitions

The definitions specified inside the declarations of sets and parameters together form a system of functional relationships. As discussed in Chapter 7 AIMMS automatically determines the dependency between the identifiers that are used inside these relationships. Based on the (required) *a-cyclic* dependency structure between identifiers (see also Section 7.1), AIMMS knows the exact order in which identifiers need to be computed. Execution based on definitions is not controlled by the user, but takes place automatically when values are needed.

Execution based on procedures

Procedures are self-contained programs with a body consisting of execution statements. These statements typically determine the value of those identifiers which cannot be defined using a single functional relationship. Execution using procedures proceeds according to the order of execution statements encountered inside each procedure, and is therefore controlled by the user.

Relating definitions and procedures

Whenever a set or a parameter with a definition is used in an execution statement inside a procedure, and its value is not up-to-date due to previous data changes, AIMMS will compute its current value just prior to executing the corresponding statement. This updating facility in AIMMS forms the necessary and powerful connection between automatic execution based on definitions and user-initiated execution based on procedures.

Definitions and algorithms

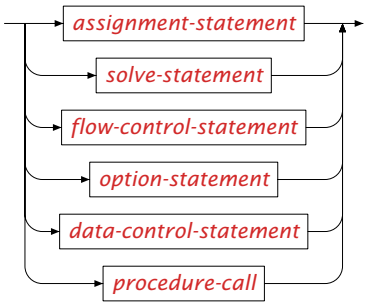
Procedural and nonprocedural execution both have their own natural role in an AIMMS application. Identifier definitions are the most convenient way to define unique functional relationships between various identifiers in your model—and keep them up-to-date at all times. Procedures provide a powerful tool to specify the algorithms that are needed to compute the identifier values without a direct functional relationship. Procedural statements are also required to communicate data between AIMMS and external data sources such as files and databases.

Execution statements

AIMMS provides a rich set of execution statements that you can use to compose your procedures. Available statements include a versatile assignment statement, statements for data and option management, the most common flow control statements, calls to other procedures, and a powerful SOLVE statement to solve various types of optimization programs.

Syntax

statement :



8.2 Assignment statements

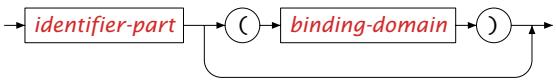
Assignment

Assignment statements are used to set or change the values of sets, parameters and variables during the execution of a procedure or a function. The syntax of an assignment statement is straightforward.

Syntax

assignment-statement :

data-selection :



Assignment operators

AIMMS offers several assignment operators. The standard *replacement* assignment operator := replaces the value of all elements specified on the left hand side with the value of the expression on the right hand side. The *arithmetic* assignment operators +=, -=, *=, /= and ^= combine an assignment with an arithmetic operation. Thus, the assignments

```
a += b,     a -= b,     a *= b,     a /= b,     a ^= b
```

form a shorthand notation for the assignments

```
a := a + b,   a := a - b,   a := a * b,   a := a / b,   a := a ^ b.
```

Index binding

Assignment is an *index binding* statement. AIMMS also binds unbound indices in (nested) references to element-valued parameters that are used for indexing the left-hand side. AIMMS will execute the assignment repeatedly for *all* elements in the binding domain, and in the order as specified by the declaration(s) of the binding set(s). The precise rules for index binding are explained in Section 9.1.

Allowed binding domains

In contrast to the binding domain of iterative operators and the FOR statements, the binding domain of an indexed assignment can contain the full range of element expressions:

- references to unbound indices, which will be bound by the assignment,
- references to scalar element parameters and bound indices,
- references to indexed element parameters, for which any nested unbound index will be bound as well,
- calls to element-valued functions, and
- element-valued iterative operators.

If the element expression inside the binding domain of an indexed assignment is too lengthy, it may be better to use an intermediate element parameter to improve readability.

Conditional assignments

Like any binding domain, the binding domain of an indexed assignment can be subject to a logical condition. Such an assignment is referred to as a *conditional assignment*, and is only executed for those elements in the binding domain that satisfy the logical condition.

Domain checking

In addition, if the identifier on the left-hand side of the assignment has its own domain restriction, then the assignment is limited to those elements of the binding domain that satisfy this restriction. Assignments to elements outside the restricted domain are not considered.

Example

The following five examples illustrate some simple assignment statements. In all examples we assume that `i` and `j` are unbound indices into a set `Cities`, and that `LargestCity` is an element parameter into `Cities`.

1. The first example illustrates a simple *scalar assignment*.

   ```
   TotalTransportCost := sum[(i,j), UnitTransportCost(i,j)*Transport(i,j)];
   ```

   The value of the scalar identifier on the left-hand side is replaced with the value of the expression on the right-hand side.
2. The second example illustrates an *index binding assignment*.

   ```
   UnitTransportCost(i,j) *= CostWeightFactor(i,j) ;
   ```

   For *all* cities `i` and `j` in the index domain of `UnitTransportCost` , the old values of the identifier `UnitTransportCost(i,j)` are multiplied with the values of the identifier `CostWeightFactor(i,j)` and then used to replace the old values.
3. The third example illustrates a *conditional assignment*.

   ```
   Transport((i,j) | UnitTransportCost(i,j) > 100) := 0;
   ```

   The zero assignment to `Transport` is made to only those cities `i` and `j` for which the `UnitTransportCost` is too high.
4. The fourth example illustrates a *sliced assignment*, i.e. an assignment that only changes the values of a lower-dimensional subspace of the index domain of the left-hand side identifier.

   ```
   Transport(LargestCity,j) := 0;
   ```

   The sliced assignment in this example binds only the index `j`. The values of the parameter `Transport` are set to zero from the city `LargestCity` to *every* city `j`, but the values from every other city *i* to all cities `j` remain unchanged.
5. The fifth example illustrates a *nested index binding statement*.

   ```
   PreviousCity( NextCity(i) ) := i;
   ```

   The index `i` is bound, because it is used in the nested reference of the element parameter `NextCity(i)`, which in turn is used for indexing the identifier `PreviousCity`. Note that, in a tour, city `i` by definition is the previous city of the specific (next) city it is linked with.

Sequential execution

Indexed assignments are executed in a sequential manner, i.e. as if it was replaced by a sequence of individual assignments to every element in the binding domain. Thus, if `Periods` is the integer set `{0 .. 3}` with index `t`, then the indexed assignment

```
Stock( t | t > 0 ) := Stock(t-1) + Supply(t) - Demand(t);
```

is executed (conceptually) as the sequence of individual statements

```
Stock(1) := Stock(0) + Supply(1) - Demand(1);
Stock(2) := Stock(1) + Supply(2) - Demand(2);
Stock(3) := Stock(2) + Supply(3) - Demand(3);
```

Therefore, in the right hand side expression it is possible to refer to elements of the identifier on the left which have received their value prior to the execution of the current individual assignment. This type of behavior is typically observed and wanted in stock balance type applications which use lag references as shown above. The same argument also applies to assignments that use element parameters for indexing on either the left- or right-hand side of the assignment.

Indexed assignment versus FOR

In addition to the indexed assignment, AIMMS also possesses a more general FOR statement which repeatedly executes a group of statements for all elements in its binding domain (see also Section 8.3.4). If you are familiar with programming languages like C or PASCAL you might be tempted to embed every indexed assignment into one or more FOR statements with the proper domain. Although this will conceptually produce the same results, we strongly recommend against it for two reasons.

- By omitting the FOR statements you improve to the readability and maintainability of your model code.
- By including the FOR statement unnecessarily you are effectively degrading the performance of your model, because AIMMS can execute an indexed assignment much more efficiently than the equivalent FOR statement.

Whenever you use a FOR statement unnecessarily, AIMMS will produce a compile time warning to tell you that the code would be more efficient by removing the FOR statement.

Example

Consider the indexed assignment

```
Transport((i,j) | UnitTransportCost(i,j) > 100) := 0;
```

and the equivalent FOR statement

```
for ((i,j) | UnitTransportCost(i,j) > 100) do
    Transport(i,j) := 0;
endfor;
```

Notice that the indexed assignment is more compact than the FOR statement and is easier to read. In this example AIMMS will warn against this use of the FOR statement, because it can be removed without any change in semantics, and will lead to more efficient execution.

Undefined left-hand references

When there are undefined references with lag and lead operators on the left-hand side of an assignment (i.e. references that evaluate to the empty element), the corresponding assignments will be skipped. The same is true if the identifier on the left contains undefined references to element parameters. Notice that this behavior is different from the behavior of a reference containing undefined lag and lead expressions on the right-hand side of an assignment. These evaluate to zero.

Example

Consider the assignment to the parameter Stock above. It could also have been written as

```
Stock(t+1) := Stock(t) + Supply(t+1) - Demand(t+1);
```

In this case, there is no need to add a condition to the assignment for t = 3. The reference to t+1 is undefined, and hence the assignment will be skipped. Similarly, the assignment

```
PreviousCity( NextCity(i) ) := i;
```

will only be executed for those cities i for which NextCity(i) is defined.

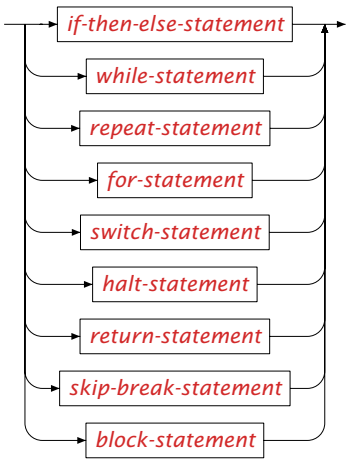
8.3 Flow control statements

Six forms of flow control

Execution statements such as assignment statements, SOLVE statements or data management statements are normally executed in their order of appearance in the body of a procedure. However, the presence of control flow statements can redirect the flow of execution as the need arises. AIMMS provides six forms of flow control:

- the IF-THEN-ELSE statement for conditional execution,
- the WHILE statement for repetitive conditional execution,
- the REPEAT statement for repetitive unconditional execution,
- the FOR statement for repetitive domain-driven execution,
- the SWITCH statement for branching on set and integer values,
- the HALT and RETURN statement for terminating the current execution,
- the SKIP and BREAK statements for terminating the current repetitive execution, and
- the BLOCK statement for visually grouping together multiple statements.

flow-control-statement : *Syntax*



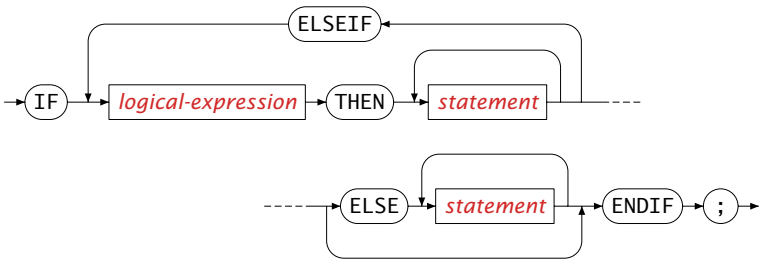
Flow control statements and special numbers

In the condition of flow control statements such as IF-THEN-ELSE, WHILE and REPEAT it is needed to know whether the result is equal to 0.0 or not in order to take the appropriate branch of execution. The special number NA has the interpretation "not yet available" thus it is also not yet known whether it is equal to 0.0 or not. The special number UNDF is the result of an illegal operation, so its value cannot be known. Therefor, AIMMS will issue an error message if the result of a condition in these statements evaluates to NA or UNDF. Special numbers and their interpretation as logical values are discussed in full detail in Sections 6.1.1 and 6.2.

8.3.1 The IF-THEN-ELSE statement

The conditional IF-THEN-ELSE statement is used to choose between the execution of several groups of statements depending on the outcome of one or more logical conditions. The syntax of the IF-THEN-ELSE statement is given in the following diagram.

if-then-else-statement : *Syntax*



AIMMS will evaluate all logical conditions in succession and stops at the first condition that is satisfied. The statements associated with that particular branch are executed. If none of the conditions is satisfied, the statements of the ELSE branch, if present, will be executed.

The following code illustrates the use of the IF-THEN-ELSE statement. *Example*

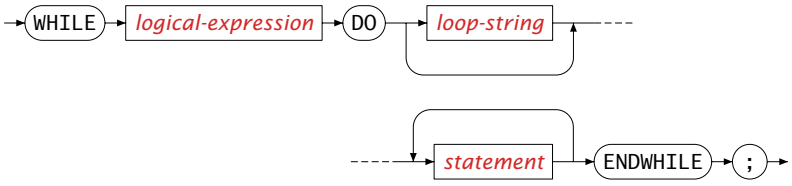
```
if ( not SupplyDepot ) then
    DialogMessage( "Select a supply depot before solving the model" );
elseif ( Exists[ p, Supply(SupplyDepot,p) < Sum( i, Demand(i,p) ) ] ) then
    DialogMessage( "The selected supply depot has insufficient capacity" );
else
    solve TransportModel ;
endif ;
```

Note that in this particular example the evaluation of the ELSEIF condition only makes sense when a SupplyDepot exists. This is automatically enforced because the IF condition is not satisfied. Similarly, successful execution of the ELSE branch apparently depends on the failure of both the IF and ELSEIF conditions.

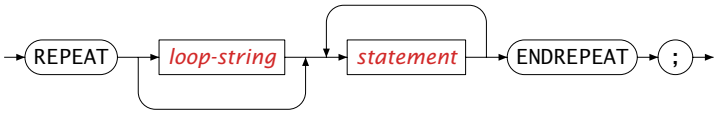
8.3.2 The WHILE and REPEAT statements

The WHILE and REPEAT statements group a series of execution statements and execute them repeatedly. The execution of the repetitive loop can be terminated by a logical condition that is part of the WHILE statement, or by means of a BREAK statement from within both the WHILE and REPEAT statements.

while-statement : *Syntax*



repeat-statement :



Loop strings are discussed in Section 8.3.3.

Termination by WHILE condition

The execution of a WHILE statement is subject to a logical condition that is verified each time the statements in the loop are executed. If the condition is false initially, the statements in the loop will never be executed. In case the WHILE loop does not contain a BREAK, HALT or RETURN statement, the statements inside the loop must in some way influence the outcome of the logical condition for the loop to terminate.

Termination by a BREAK statement

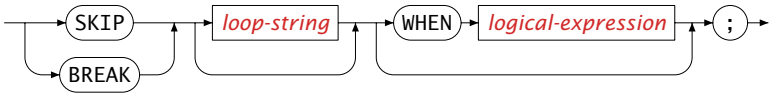
An alternative way to terminate a WHILE or REPEAT statement is the use of a BREAK statement inside the loop. BREAK statements make it possible to abort the execution at any position inside the loop. This freedom allows you to formulate more natural termination conditions than would otherwise be possible with just the logical condition in the WHILE statement. After aborting the loop, AIMMS will continue with the first statement following it.

Skipping the remainder of a loop

In addition to the BREAK statement, AIMMS also offers a SKIP statement. With it you instruct AIMMS to skip the remaining statements inside the current iteration of the loop, and immediately return to the top of the WHILE or REPEAT statement to execute the next iteration. The SKIP statement is an elegant alternative to placing the statements inside the loop following the SKIP statement in a conditional IF statement.

Syntax

skip-break-statement :



The WHEN clause

By adding a WHEN clause to either a BREAK or SKIP statement, you make its execution conditional to a logical expression. In practice, the execution of a BREAK or SKIP statement is almost always subject to some condition.

Example WHILE statement

This example computes the *machine epsilon*, which is the smallest number that, when added to 1.0, gives a value different from 1.0. It is a measure of the accuracy of the floating point arithmetic, and it is machine dependent. We assume that meps is a scalar parameter, and that the numeric comparison tolerances are set to zero (see also Section 6.2.2).

```
meps := 1.0;

while (1.0 + meps/2 > 1.0) do
    meps /= 2;
endwhile;
```

Since the parameter meps is determined iteratively, and the loop condition will eventually be satisfied, this example illustrates an appropriate use of the WHILE loop.

Example REPEAT statement

By applying a BREAK statement, the machine epsilon can be computed equivalently using the following REPEAT statement.

```
meps := 1.0;

repeat
    break when (1.0 + meps/2 = 1.0) ;
    meps /= 2;
endrepeat;
```

The BREAK statement could also have been formulated in an equivalent but less elegant manner without a WHEN clause:

```
if (1.0 + meps/2 = 1.0) then
   break;
endif;
```

8.3.3 Advanced use of WHILE and REPEAT

Advanced uses

Next to the common use of the WHILE and REPEAT statements described in the previous section, AIMMS offers some special constructs that help you

- keep track of the number executed iterations automatically, and
- control nested arrangements of WHILE and REPEAT statements.

Nonconvergent loops

There are practical examples in which the terminating condition of a repetitive statement may not be met at all or at least not within a reasonable amount of work or time. A good example of this behavior are solution algorithms for which convergence is likely but not guaranteed. In these cases, it is common practice to terminate the execution of the loop when the total number of iterations exceeds a certain limit.

The LoopCount operator

In AIMMS, such conditions can be formulated easily without the need to

- introduce an additional parameter,
- add a statement to initialize it, and
- increase the parameter every iteration of the loop.

Each repetitive statement keeps track of its iteration count automatically and makes the number of times the loop is entered available by means of the predefined operator LoopCount. Upon entering a repetitive statement AIMMS will set its value to 1, and will increase it by 1 at the end of every iteration.

Example

Whether the following sequence will converge depends on the initial value of x. In the case where there is no convergence or if convergence is too slow, the loop in the following example will terminate after 100 iterations.

```
while ( Abs(x-OldValue) >= Tolerance and LoopCount <= 100 ) do
    OldValue := x ;
    x        := x^2 - x ;
endwhile ;
```

Naming nested loops

So far, we have considered single loops. However, in practice it is quite common that repetitive statements appear in nested arrangements. To provide finer control over the flow of execution in such situations, AIMMS allows you to label a particular repetitive statement with a *loop string*.

Use of loop strings

Using a loop string in conjunction with the BREAK and SKIP statements, it is possible to break out from several nested repetitive statements with a single BREAK statement. The loop string argument can also be supplied to the LoopCount operator so the break can be conditional on the number of iterations of any loop. Without specifying a loop string, BREAK, SKIP and LoopCount refer to the current loop by default.

Example

The following example illustrates the use of loop strings and the LoopCount operator in nested repetitive statements. It outlines an algorithm in which the domain of definition of a particular problem is extended in every loop based on the current solution, after which the new problem is solved by means of a sequential solution process.

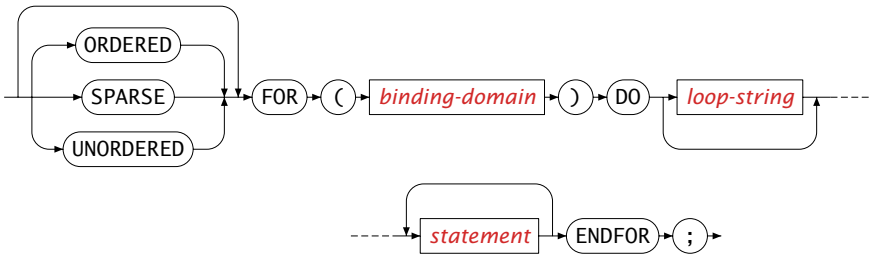
```
repeat "OuterLoop"
     ... ! Determine initial settings for sequential solution process

    while( Abs( Solution - OldSolution ) <= Tolerance ) do
        OldSolution := Solution ;

        ... ! Set up and solve next sequential step ...

        ! ... but terminate algorithm when convergence is too slow
        break "OuterLoop" when LoopCount >= LoopCount("OuterLoop")^2 ;
    endwhile;

    ... ! Extend the domain of definition based on current solution,
        ! or break from the loop when no extension is possible anymore.
endrepeat;
```

8.3.4 The FOR statement

The FOR statement is related to the use of iterative operators in expressions. An iterative operator such as SUM or MIN applies a particular operation to all expressions defined over a particular domain. Similarly, the FOR statement executes a group of execution statements for all elements in its domain. The syntax of the FOR statement is given in the following diagram.

Syntax

for-statement :



Execution is sequential

The binding domain of a FOR statement can only contain free indices, which are then bound by the statement. All statements inside a FOR statement are executed in sequence for the specific elements in the binding domain. Unless specified otherwise, the ordering of elements in the binding domain, and hence the execution order of the FOR statement, is the same as the order of the corresponding binding set(s).

Integer domains

FOR statements with an integer domain in the form of an enumerated set behave in a similar manner as the FOR statement in programming languages like C or Pascal. Like the example below, FOR statements of this type are mostly of an algorithmic nature, and the indices bound by the FOR statement typically serve as an iteration count.

Example

```
for ( n in { 1 .. MaxPriority } ) do

    x.NonVar( i | x.Priority(i) < n ) := 1;
    x.Relax ( i | x.Priority(i) = n ) := 0;
    x.Relax ( i | x.Priority(i) > n ) := 1;

    Solve IntegerModel;
endfor;
```

This example tries to solve a mixed-integer mathematical program heuristically in stages. The algorithm first only solves for those integer variables that have a particular integer priority, and then changes them to non-variables before going on to the next priority. The suffices used in this example are discussed in Section 14.1.

Non-integer domains

FOR statements with non-integer binding domains are typically used to process the data of a model for all elements in a data-related domain. The use of a FOR statement in such a situation is only necessary if the statements inside it form a unit, for which sequential execution for each element in the domain of the entire group of statements is essential. An example follows.

Example

```
for ( i in Cities ) do
    SmallestTransportCity := ArgMin( j, Transport(i,j) ) ;
    DiscardedTransports   += Transport( i, SmallestTransportCity ) ;
    Transport( i, SmallestTransportCity ) := 0 ;
endfor;
```

In this example the three assignments form an inseparable unit. For each particular value of `i`, the second and third assignment depend on the correct value of `SmallestTransport` in the first assignment.

Use FOR only when needed

If you are familiar with programming language like PASCAL and C, then the use of `FOR` statements will seem quite natural. In AIMMS, however, `FOR` statements are often not needed, especially in the context of indexed assignments. Indexed assignments bind the free indices in their domain implicitly, resulting in sequential execution of that particular assignment for all elements in its domain. In general, such an index binding assignment is executed much more efficiently than the same assignment placed inside an equivalent `FOR` statement. In general, you should use `FOR` statements only when really necessary.

AIMMS *issues a warning*

AIMMS will provide a warning when it detects unnecessary `FOR` statements in your model. Typically `FOR` statement are not required when the loop only contains assignments that do not refer to scalar identifiers (either numeric or element-valued) to which assignments have been made inside the loop as well. For instance, in the last example the `FOR` statement is essential, because the assignment and use of the element parameter `LargestTransportCity` is inside the loop.

Example

The following example shows an unnecessary use of the `FOR` statement.

```
solve OptimizationModel;

! Mark variables with large marginal values
for (i) do
    if ( Abs[x.Marginal(i)] > HighPrice ) then
        Mark(i) := x.Marginal(i);
    else
        Mark(i) := 0.0;
    endif;
endfor;
```

While this statement may seem very natural to C or Pascal programmers, in a sparse execution language like AIMMS it should preferably be written by the following simpler, and faster, assignment statement.

```
Mark(i) := x.Marginal(i) OnlyIf ( Abs[x.Marginal(i)] > HighPrice );
```

With the optional keywords SPARSE, ORDERED and UNORDERED you can indicate that AIMMS should follow one of three possible strategies to execute the FOR statement. If you do not explicitly specify a strategy, AIMMS will follow the SPARSE strategy by default, and issue a warning when this strategy leads to severe inefficiencies. You can find an explanation of each of the strategies, as well as a description of the cases in which you may want to choose a specific strategy in Section 13.2.2.

The SPARSE, ORDERED and UNORDERED keywords

Like the WHILE and the REPEAT statements, FOR is a repetitive statement. Thus, you can use the SKIP and BREAK statements and the LoopCount operator. In addition, you can identify a FOR statement with a loop string thereby controlling execution in nested arrangements as discussed in the previous section.

FOR as a repetitive statement

The SKIP statement skips the remaining statements in the FOR loop and continues to execute the loop for the next element in the binding domain. The BREAK statement will abort the execution of the FOR statement all together.

Use of SKIP and BREAK

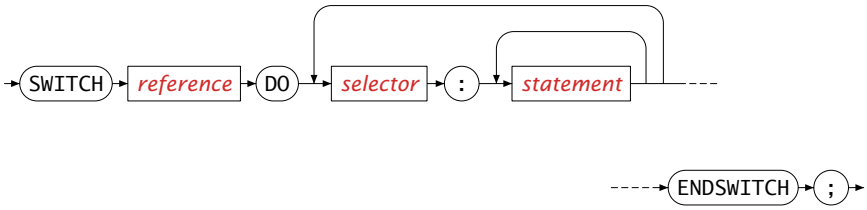
8.3.5 The SWITCH statement

The SWITCH statement is used to choose between the execution of different groups of statements depending on the value of a scalar parameter reference. The syntax of the SWITCH statement is given in the following two diagrams.

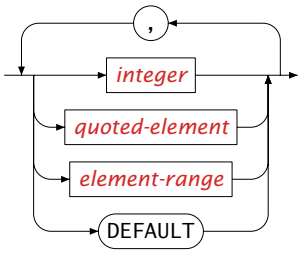
The SWITCH statement

Syntax

switch-statement :



selector :



Integers and set element

The SWITCH statement can switch on two types of scalar parameter references: set element-valued or integer-valued. When you try to switch on references to string-valued or non-integer numerical parameters, AIMMS will issue a compile time error

Switch selectors

Each selector in a SWITCH statement must be a comma-separated list of values or value ranges, matching the type of the selecting scalar parameter. Expressions and ranges used in a SWITCH statement must only contain constant integers and set elements. Set elements used in a switch selector must be known at compile time, i.e. the data initialization of the corresponding set must be a part of the model description.

The DEFAULT selector last

The optional DEFAULT selector matches every reference. Since AIMMS executes only those statements associated with the *first* selector matching the value of the scalar reference, it is clear that the DEFAULT selector should be placed last.

Example

The following SWITCH statement takes different actions based on the model status returned by a SOLVE statement.

```
solve OptimizationModel;

switch OptimizationModel.ProgramStatus do
    'Optimal', 'LocallyOptimal' :
            ObservedModelStatus := 'Solved' ;

    'Unbounded', 'Infeasible', 'IntegerInfeasible', 'LocallyInfeasible' :
            ObservedModelStatus := 'Infeasible' ;

    'IntermediateInfeasible', 'IntermediateNonInteger', 'IntermediateNonOptimal' :
            ObservedModelStatus := 'Interrupted' ;

    default :
            ObservedModelStatus := 'Not solved' ;
endswitch ;
```

8.3.6 The HALT statement

Terminating execution

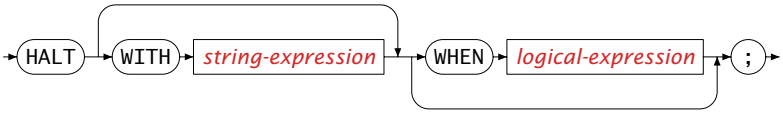
With a HALT statement you can stop the current execution. You can use it, for example, if your model has run into an unrecoverable error condition during its execution, or if you simply want to skip the remaining statements because they are no longer relevant in a particular situation.

Compare to RETURN

Instead of the HALT statement you can also use the RETURN statement (see also Section 10.1) to terminate the current execution. The HALT statement directly jumps back to the user interface, but a RETURN statement in a procedure only passes back control to the calling procedure and continues execution from there.

The syntax of the HALT statement follows. *Syntax*

halt-statement :



You can optionally specify a string in the HALT statement that will be printed in a message dialog box when execution has stopped. This is useful, for instance, to pass on an appropriate message to the user when a particular error condition has occurred. *Printing a message*

You can make the execution of the HALT statement conditional on a WHEN clause. If present, the current run will only be aborted if the condition after the WHEN clause is satisfied. *The WHEN clause*

The following example terminates the current run if the SOLVE statement does not solve to optimality. When aborting, the user will be notified with an explanatory message. *Example*

```
solve LinearOptimizationModel;

halt with "Execution aborted: model not solved to optimality"
     when OptimizationModel.ProgramStatus <> 'Optimal' ;
```

Note that the type of model termination initiated by calling the HALT statement cannot be guarded against using AIMMS' error handling facilities (see Section 8.4). An alternative to the HALT statement, which enables error handling, is the RAISE statement discussed in Section 8.4.2. When you want to let the HALT act as a RAISE statement, you can switch the option halt_acts_as_raise_error on. *Alternative*

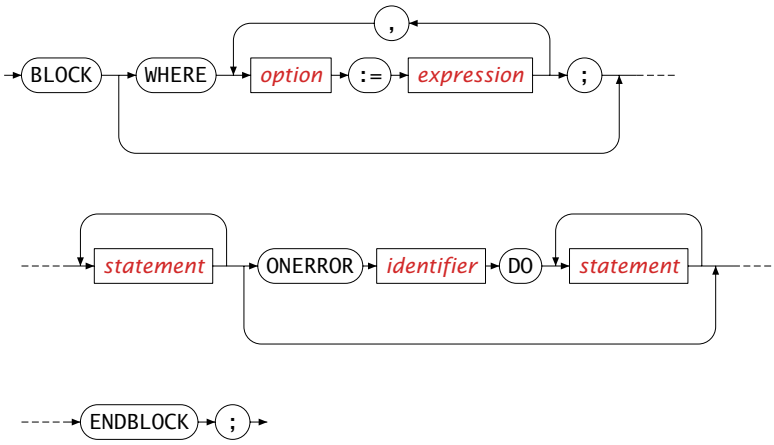
8.3.7 The BLOCK statement

A sequence of statements can be grouped together into a single statement using the BLOCK statement, possibly serving one or more of the following purposes: *The BLOCK statement*

- to emphasize the logical structure of the model,
- to execute a group of statements with different option settings, or
- to permit error handling on a group of statements (see Section 8.4).

The syntax of the BLOCK statement is as follows.

block-statement : *Syntax*

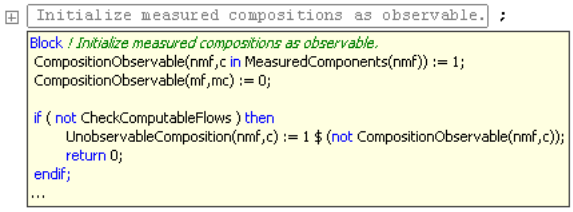


Consider the following BLOCK statement containing a group of statements. *Emphasizing logical structure in the model*

```
block ! Initialize measured compositions as observable.
    CompositionObservable(nmf,c in MeasuredComponents(nmf)) := 1;
    CompositionObservable(mf,mc) := 0;

    if ( not CheckComputableFlows ) then
        UnobservableComposition(nmf,c) := 1$(not CompositionObservable(nmf,c));
        return 0;
    endif;

    CompositionCount(pu,c) :=
        Count((f,g) | Admissable(pu,c,f,g) and CompositionObservable(g,c));
    NewCount := Card ( CompositionObservable );
endblock ;
```

In the AIMMS syntax editor, a block can be displayed in either a collapsed or an expanded state. When collapsed, the block will be displayed as follows, using a single line comment following the BLOCK keyword as its description.

When in a collapsed state, AIMMS will show the contents of the block in a tooltip if the mouse pointer is placed over the collapsed block, as illustrated in the figure below.



Executing with different option settings

During the execution of a block statement, the options in the WHERE clause will have the specified values set at the beginning of the block statement, and the old values restored at the end of the block statement. More on the format of option names and value settings can be found in Section 8.5. The example below prints various parameters using various settings of the option Listing_number_precision.

```
! The default value of the option Listing_number_precision is 3.
block ! Start printing numbers using 6 decimals.
    where Listing_number_precision := 6 ;

    display A, B ;

    block ! Start printing numbers without decimals.
        where Listing_number_precision := 0 ;
        display C, D ; ! The output looks as if C and D are integers.
    endblock ;

    display E, F ; ! Back to printing numbers using 6 decimals.

endblock ;

display G, H ; ! Back to printing numbers using 3 decimals.
```

In the above example, a nested block statement is used to set the scope of option settings; the inner block statement temporarily overrides the option setting of the outer block statement, which overrides the global option settings.

The OnError clause

The OnError clause is one of the means of handling runtime errors in AIMMS. It is discussed in detail in Section 8.4.1.

8.4 Raising and handling warnings and errors

Errors and warnings

During the development and deployment of an AIMMS application, unexpected, possibly harmful, situations can arise. These situations are divided into errors and warnings. An error is a situation that cannot be handled by the procedure encountering it. A warning is a situation that can be handled by the procedure encountering it, but might warrant further inspection by the model developer or by the model user. Note that, even when a procedure cannot handle an error itself, it should be able to recover from that error. In this section, you will find AIMMS facilities to

- handle errors; to handle an error, AIMMS will give you access to the information therein. A handler is a piece of AIMMS code that handles selected errors and warnings. Errors and warnings can be communicated to handlers higher in the execution stack.
- raise an error; not only AIMMS may detect situations warranting an error or warning message, but also the application itself. For such situations AIMMS provides a facility to raise custom errors from within your model.

- handle a legacy situation; external and intrinsic AIMMS procedures may return a status code indicating success or failure. Whenever a failure status of an external and intrinsic procedure remains unnoticed, AIMMS can automatically raise an error in such situations.
- extensively check the code; AIMMS can check your application for many different kinds of situations that occassionally warrant a warning. It is usually worthwhile to apply all these checks to your application.

8.4.1 Handling errors

Subsection overview

In this subsection you will find an introduction to both the global and local error handling mechanisms available in AIMMS. Global error handling, by means of specifying a single handler procedure, is used to treat runtime errors occuring anywhere inside the entire model that are not handled elsewhere. Local error handling, by means of the `OnError` clause in a `BLOCK` statement, allows error handling of runtime errors occuring in a specific block of code. Global and local error handling are the blocks on which the error handling framework in AIMMS is built. At the end of this subsection, you will find a description of all the intrinsic functions available for accessing and manipulating information regarding errors.

Global error handling

To activate global error handling, the name of a handling procedure in your model must be assigned to the option `Global_error_handler`. Such a procedure must have a single element parameter argument `err` in the predeclared set `errh::PendingErrors`. The global error handling procedure will be executed for each pending error whenever an execution run has been terminated because of errors that have not been handled elsewhere in the model. The global error handler will also be called at the end of a finished execution run if there are unhandled warnings. In this context, an execution run is any call to an AIMMS procedure initiated either through the AIMMS GUI or through the AIMMS API.

Example

Below a global error handling procedure `MyErrorHandler` is illustrated. The lines in the body of the procedure are numbered to facilitate the explanation of the example.

```
Procedure MyErrorHandler {
    Arguments  :  err;
    ElementParameter  err {
        Range      :  errh::PendingErrors;
        Property   :  Input;
    }
    Body: {
        1  if errh::Node(err) = 'DefP' then
        2      DialogMessage(errh::Message(err) + "; resetting P to its default.");
        3      Empty P ;
        4      errh::MarkAsHandled(err);
        5  elseif errh::InsideCategory(err,'IO') then
        6      errh::Adapt(err,message:"IO error: please consult ...; "
```

```
        7         + errh::Message(err)  ); ! Pass adapted message on to next handler.
        8  else
        9      ! Errors not handled will be passed on to the error/warning window.
        10  endif
    }
}
```

Example explanation

The procedure starts with declaring the argument `err` as an element parameter with the predeclared set `errh::PendingErrors`, with a subset of the predeclared set `Integers` as its range. During an execution run, this set is filled with the numbers of the errors and warnings raised. Each number refers to an error or warning with various pieces of information therein, such as its error description, the node in which the error or warning occurred and its severity. In addition, each error belongs to a category. All this information can be accessed using intrinsic functions. The body of the procedure is now explained line by line:

- **line 1:** The intrinsic function `errh::Node` is used to determine whether or not the error occurred inside the procedure `DefP`. This intrinsic function returns the identifier or node in which the error occured as an element of the predeclared set `AllSymbols`.
- **lines 2, 3:** If the error did happen inside the procedure `DefP`, the application user is notified and `P` is reset to its default. The notification uses the original error description obtained using the intrinsic function `errh::Message(err)`.
- **line 4:** Each handled error will be marked as such. When an error handler finishes, it will delete the errors that have been marked as handled from the predeclared set `errh::PendingErrors`.
- **line 5:** To discern the type of an error, errors are divided into categories. For each error, the category to which it belongs can be obtained using the function `errh::Category(err)`. The error categories form a nested structure. For instance, both `IO` and `Generation` errors are `Execution` errors. The intrinsic function `errh::InsideCategory(err)` can be used to determine whether or not an error is within a particular category.
- **lines 6, 7:** Translate the error by adapting information. In this example, only the message is actually adapted, but most parts of an error can be adapted. Note that in this `else` branch, the function `errh::MarkAsHandled` is not called, the result being that the adapted error message will appear in the messages/errors window.
- **line 8:** In this branch, the error is not handled. An error that has not been handled when the error handler finishes will not be deleted. Instead, it is being displayed in the messages/errors window.

Local error handling by means of the OnError clause

The following template of a BLOCK statement illustrates local error handling by means of the OnError clause.

```
1    BLOCK
2       statement_1 ;
3       ...
4       statement_n ;
5    ONERROR err DO
6            ...
7            ...
8    ENDBLOCK ;
```

All errors occuring inside statement_1 ... statement_n on lines 2 ... 4 are handled by the error handler on lines 6 and 7, where err is an element parameter of the set errh::PendingErrors. Block statements can be nested, either directly in a single body, or in other procedures called from within block statements. This gives rise to a stack of error handlers as illustrated below. A detailed example of a local error handler is given in Section 35.6.

Error flow architecture

The global error handlers and the OnError error handlers are essential building blocks of the error handling framework of AIMMS. This error handling framework is illustrated in Figure 8.1.

Construction of the error handler stack

At the start of each execution run, a new stack of error handlers is created. At the bottom of this stack is the standard handler To Global Collector. When the option Global_error_handler is set, the specified procedure is placed on top of this new stack. Additional handlers are placed on the stack by each OnError clause in a nested BLOCK statement.

Errors flowing through a handler stack

When raised, each error is set aside for handling by the topmost error handler. When the number of errors set aside reaches the limit specified by the option Errors_until_execution_interrupt, the execution is interrupted and resumes by executing the code in the topmost error handler. When the execution is not interrupted, but there are pending errors or warnings, the error handling code is executed after the completion of the last statement prior to the BLOCK statement.

Multiple errors may require handling

A single statement may result in multiple error messages, for instance a solve statement or a data assignment statement with several duplicate entries. Thus, even if the option Errors_until_execution_interrupt is 1 (its default), multiple errors may need to be handled. If multiple errors caused by a single statement are handled inside the OnError clause of a BLOCK statement, the code within the OnError clause will be executed unconditionally *for every single error*, unless you explicitly break away from theOnError clause.

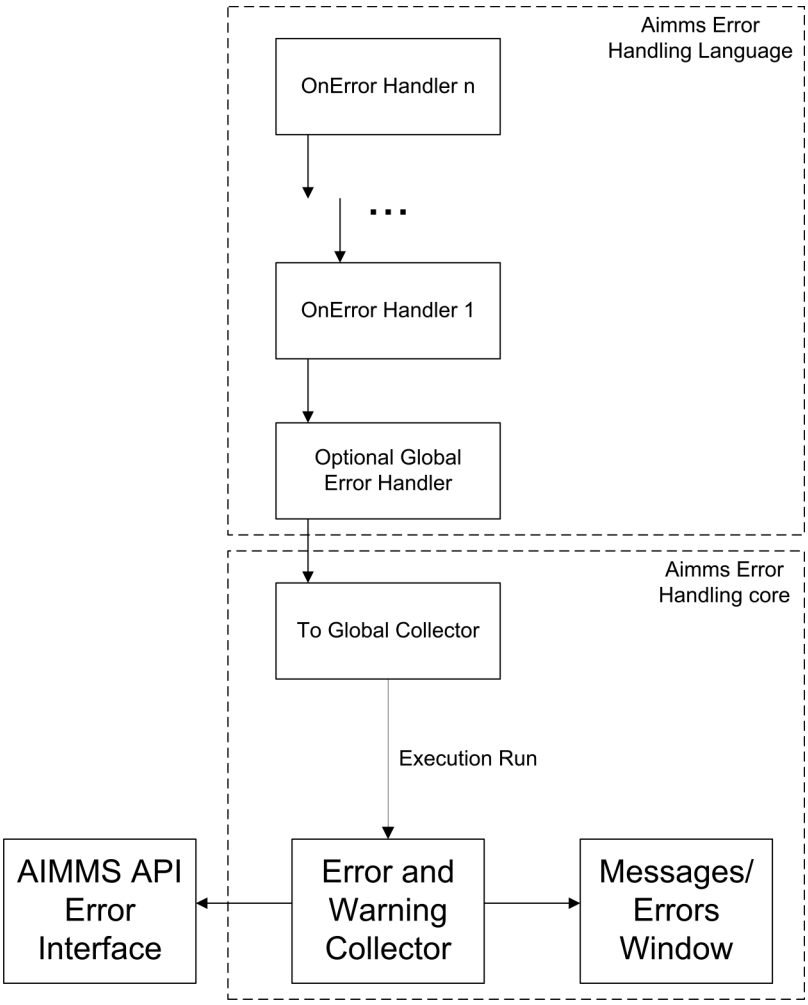


Figure 8.1: Error flow through handlers

Break away from handling

If you use a RETURN, HALT, BREAK or RAISE ERROR statement inside the OnError clause, the handling of any subsequent errors or warnings will be stopped. You are actually indicating that these further errors and warnings are no longer of interest and thus they will be automatically set as handled. A plain BREAK statement just breaks the error handling loop. If the Block statement is inside an outer loop statement like FOR or WHILE and you want to break from that loop, you need to use a *loop string* (see Section 8.3.3).

SKIP in OnError

A plain Skip statement in the OnError clause simply skips the remaining statements and continues with the next error that needs to be handled. You can use a SKIP with a *loop string* to skip the statements of an outer loop statement. This will break away from the OnError clause as described above.

What to do with an error

For each error, the error handling code will decide whether to handle that error itself, let another handler handle the error, or ignore the error (as was already illustrated in the example above).

Handling an error inside a handler

Errors may also occur during the execution of the OnError clause or of a BLOCK statement or the global error handling procedure. These errors are handled by the next error handler in the stack of error handlers.

Error collector

When an error reaches the handler To Global Collector, it is sent to the **Error and Warning Collector** object which collects all errors that have fallen through the various handlers (if any). Errors in the **Error and Warning Collector** can be queried from within the AIMMS API or viewed from within the messages/errors window of the AIMMS GUI.

The predeclared module ErrorHandling

Errors to be handled can be queried using the following predeclared identifiers and intrinsic functions from the module ErrorHandling with prefix errh:

- **errh::PendingErrors:** A predeclared set filled with the numbers of the errors that can be handled at this point.
- **errh::IndexPendingErrors:** An index of the above predeclared set.
- **error parts:** An error is made up of several parts; each of which can be obtained separately using the intrinsic functions below. Each of the functions below will raise an error of their own if err is not a valid error that can be handled at that point.
  - **errh::Severity(err):** An element in errh::AllErrorSeverities is returned indicating the severity of the error.
  - **errh::Message(err):** A string containing the error description is returned. This string is not empty.
  - **errh::Category(err):** An element in errh::AllErrorCategories is returned indicating the category of the error.
  - **errh::Code(err):** The element in errh::ErrorCodes that is returned by this function identifies the message code of the error. This element name may be cryptic; as it is primarily used for identification of the error within the AIMMS system.
  - **errh::NumberOfLocations(err):** The number of locations relevant to this error. For compilation errors, there is typically only one relevant location. For an AIMMS initialization error there are no relevant locations. For an execution error the positions in all the active procedures are recorded. For an error during file read, at least the positions in the data file and the read statement are recorded. Similarly, for an error during the generation of a constraint, at least the constraint and the SOLVE statement are recorded as relevant positions.
  - **errh::Node(err,loc):** An element in AllSymbols is returned for an error location inside the model. The optional argument loc defaults to 1 and should be in the range { 1 .. NumberOfLocations } . The

element returned by this function is non-empty except for the first location when reading data from a file.

- **errh::Attribute(err,loc):** An element in AllAttributeNames.
- **errh::Line(err,loc):** An integer indicating the line number of the error in the attribute or file, or 0 if not known.
- **errh::Column(err):** An integer indicating the column position in an erroneous line being read from a data file. All errors when reading a data file are reported separately, such that the loc argument is not applicable.
- **errh::Filename(err):** A non-empty string is returned when reading from a data file. All errors when reading a data file are reported separately, and so the loc argument is not applicable.
- **errh::Multiplicity(err):** An integer indicating the number of occurrences of this error. Two errors are considered equal if they are equal in all of the following parts: Severity, Message, Category, Code and the first location (if available). The first location is the location in the file being read when the error occurs during a read statement, otherwise it is the statement being executed.
- **errh::CreationTime(err,fmt):** A string representing the creation time of the first occurrence of the error, formatted according to time format fmt.

- **errh::InsideCategory(err,cat):** Returns 1 if the error code of err falls inside the category cat.
- **errh::IsMarkedAsHandled(err):** Returns 1 if the error is marked as handled.
- **errh::Adapt(err, severity, message, category, code) :** The error err is adapted with the components specified. Besides the mandatory argument err, there should be at least one other argument.
- **errh::MarkAsHandled(err,actually):** The error err is marked as handled if the argument actually is non-zero. Marked errors will not be passed to the next error handler. The default of the optional argument actually is 1. Using 0 will remove the mark from the error.

The log file aimms.err

AIMMS logs all errors and warnings to the file aimms.err as they are raised. The folder in which this file resides is controlled by the option Listing_and_temporary_files. The number of backups retained of this file is controlled by the option Number_of_log_file_backups.

8.4.2 Raising errors and warnings

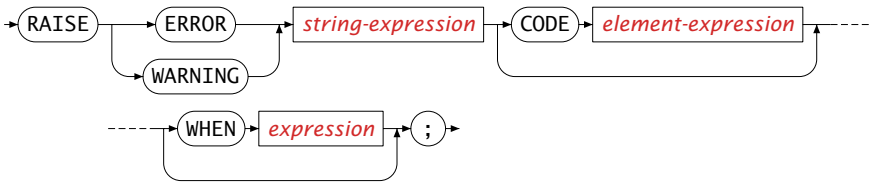
Raising errors

The RAISE statement is used to

- raise an error regarding a situation that cannot be handled, or to
- raise a warning regarding a situation that can be handled but might warrant further investigation.

The syntax of the RAISE statement is straightforward.

raise-statement : *Syntax*



In the following example an error is raised when the inflow of a node exceeds its capacity. *Example*

```
if inflow > stockCap then
    RAISE ERROR "Inflow exceeds stock capacity" CODE 'TooMuchInflow' ;
endif ;
```

In order to enable an error handler to recognize the type of error being raised by a RAISE statement, that statement allows an optional error code to be specified. This is an element in the set errh::ErrorCodes. If the specified element does not yet exist, it is created and added to that set. The category of an error raised by the RAISE statement is fixed to 'User'. *Error code and category*

AIMMS uses the line/procedure in which the RAISE statement is specified as the position information associated with the error. This permits the messages/errors window to open the attribute window of the procedure and place the cursor on the statement where the problematic situation is detected. *Position information*

Not only AIMMS itself but also procedures written in AIMMS may recognize situations that can be handled but might warrant closer inspection by the application user. For this purpose, the RAISE statement can raise a warning, for example: *Raising warnings*

```
if card( RawMaterialTraders ) = 0 then
   RAISE WARNING "There are no raw material traders, this may lead to " +
                 "infeasibilities in the case of too many accepted deliveries." ;
endif ;
```

The handling of warnings generated by a RAISE statement is controlled by the option Warning_user, with default common_warning_default. The control of warning handling is further explained in Subsection 8.4.4.

8.4.3 Legacy: intrinsics with a return status

Legacy situation

AIMMS external procedures and intrinsic procedures can both return a status code indicating whether or not they were successful. A return value ≤ 0.0 is interpreted as not successful, wheareas a return value > 0.0 is successful. In addition, when they are not successful, the error message is often left in CurrentErrorMessage, although this is only a guideline. The return value of a call to an intrinsic procedure is either

- **checked:** As illustrated in the example:

```
retval := PageOpen(...) ;
if retval <= 0 then
   ... use CurrentErrorMessage ...
endif ;
```

- **not checked:** As illustrated in the example:

```
PageOpen(...) ;
```

Available error handling methods

In the context of the error handling facility available in AIMMS, how should one handle the "checked" and "not checked" procedure calls when the return value is 0 and these procedures have not raised an error themselves? There are five error handling methods available to choose from:

- **ignore:** An error is never raised for an error occurring inside such a procedure, whether or not the return status is checked.
- **raise_warning_when_not_checked:** A warning is only raised if the return status of an intrinsic procedure is not checked.
- **raise_when_not_checked:** An error is only raised if the return status of an intrinsic procedure is not checked.
- **raise_always_warning:** A warning is raised whether or not the return status is checked.
- **raise_always:** An error is raised whether or not the return status is checked.

Which choice of error handling method is best depends on the application and can be controlled using the options:

- **Intrinsic_procedure_error_handling:** for procedures with a return status supplied by AIMMS and
- **External_procedure_error_handling:** for externally supplied procedures.

The values of these options are the names of the error handling methods described above. The default of both these options is raise_when_not_checked. For projects created prior to the introduction of the error handling facilities in AIMMS (i.e. created in AIMMS 3.9 or lower), these options generate the non-default value raise_warning_when_not_checked in order to notify the model developer but do not change the existing behavior of such projects significantly.

8.4.4 Warnings

Warnings

AIMMS recognizes and warns about several types of possibly problematic situations. These situations might warrant further investigation. As with most other languages, AIMMS warns against the use of identifiers before initializing them. But unlike other languages, AIMMS also warns against the inconsistent use of units of measurement (such as a comparison of a volume against a weight), or of model formulations for which AIMMS can detect either compiletime or runtime issues that lead to sub-optimal performance or ambiguous results. A selection of performance-related warnings is discussed in Section 13.2.8.

Complete flexibility

The desired handling of each of these situations depends on the developer and the application; varying from treating it as an error to fully ignoring it. To permit complete flexibility, there is separate option to control the reporting of each type of problematic situation recognized.

Grouping Warning options

Although all warnings can be controlled individually, this is not the most convenient way to employ the diagnostics provided by these warnings. When entertaining a new idea (quick prototyping), most modelers understandably do not want to be bothered by various warnings and want to be able to turn them all off. To facilitate this, all the warnings have been grouped into either common or strict warnings, and the associated options assume default value for common and strict warnings. Thus, all diagnostic warnings can be switched off by just changing the options that control these defaults. For normal development work it is advisable to at least turn the common warnings on. In addition, we would encourage to turn on the strict warnings during application tests.

Choosing the option setting

In order to implement the above scheme and still permit full flexibility, each option controlling the detection of a type of problematic situation can take on one of the following values:

- **error:** The situation is marked as an error and treated as an error.
- **warning_handle:** The warning is raised in the current error handler, but does not count toward the interruption of normal execution.
- **common_warning_default:** The value of the option Common_warning_default is used.
- **warning_collect:** The warning is raised in the Global_error_collector, bypassing the stack of error handlers.
- **strict_warning_default:** The value of the option Strict_warning_default is used.
- **off:** The warning is ignored.

The default of these options is either common_warning_default or strict_warning_default, thereby effectively dividing these options into common and strict groups. The range of options for common_warning_default and strict_warning_default is {off, warning_collect, warning_handle, error}. The default of the option common_warning_default is warning_handle and the default of the option strict_warning_default is off.

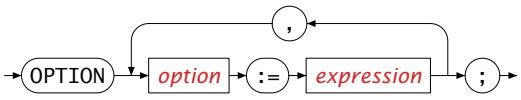
8.5 The OPTION and PROPERTY statements

Options

Options are directives to AIMMS or to the solvers to execute a task in a particular manner. Options have a name and can assume a value that is either numeric or string-valued. You can modify the value of an option from within the graphical interface. The assigned value is stored along with the project. All global options are set to their stored values at the beginning of each session. During execution you can change option settings using the OPTION statement.

Syntax

option-statement :



You can find a complete list of global options for AIMMS and its solvers in the help system.

Option values

The right-hand side of an OPTION statement must be a scalar expression of the proper type. If the option expects a string value, AIMMS will accept both string- or element-valued expressions. An example follows.

```
option  Bound_Tolerance  := 1.0e-6,
        Iteration_Limit  := UserSettings('IterationLimit');
```

Solver options

Some solver options are available for more than one solver. If you modify such a solver option per se, AIMMS will modify the option for all solver that support it. If you want to restrict the change to only a single solver, you can prefix the option name by the name of the solver followed by a dot ".", as illustrated in the example below.

```
option  'Cplex 12.9'.lp_method := 'dual simplex';
```

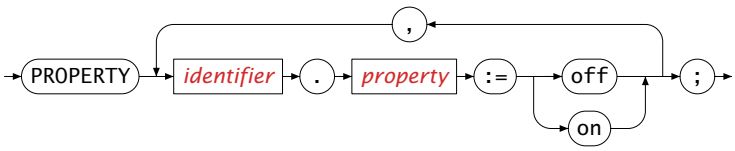
This statement will set the option lp_method of the solver that is known to the system as 'Cplex 12.9' equal to 'dual simplex'. The solver name can be either a quoted solver name, or an element parameter into the predefined set AllSolvers.

Identifier properties

Identifier properties can be turned on or off. All properties default to off, unless they are turned on—either in the declaration of the identifier or in a PROPERTY statement. During the execution of your model you can dynamically change the default values of properties through the PROPERTY execution statements.

Syntax

property-statement :



Resetting properties

The properties of all identifier types can be found in the identifier declaration sections. Not all property settings can be changed, e.g. you cannot dynamically change the Input or Output property of arguments of functions and procedures. In such cases, AIMMS will produce a runtime error. An example of the PROPERTY statement follows.

```
if ( Card(Cities) > 100 ) then
   property IntermediateTransport.NoSave := on;
endif;
```

Once the set of Cities contains more than 100 elements, the identifier IntermediateTransport is no longer saved as part of a case file.

Multiple identifiers

When the PROPERTY statement is applied to an index into a subset of the predefined set AllIdentifiers, AIMMS will change the corresponding property for all identifiers in that subset.

Example

The following example illustrates how the PROPERTY statement can be used to obtain additional sensitivity data for a set SensitivityVariables of (symbolic) variables that has been previously determined.

```
for ( var in SensitivityVariables ) do
    property var.CoefficientRanges := on;
endfor;
```

Here, you request AIMMS to determine the smallest and largest values for the objective coefficient of each variable in SensitivityVariables during the execution of a SOLVE statement such that the optimal basis remains constant (see also Section 14.1.2).

Chapter 9

Index Binding

This chapter

This chapter presents the *index binding* rules implemented in AIMMS. These rules play an essential role during most repetitive set operations. For standard situations AIMMS behaves as expected. You should read this chapter if you are interested in a formal discussion of the rules of the underlying semantics.

9.1 Binding rules

Repetitive operations

During execution, indices are used to traverse a set to repeatedly apply a specific operation on all elements of a set. These operations concern

- indexed assignment statements,
- FOR statements,
- iterative operations like summation over a domain,
- constraint generation,
- arc generation, and
- constructed set expression.

Index binding

Index binding is the process by which AIMMS repeatedly couples the value of an index to elements of a specific set to execute repetitive operations.

Different types of binding

There are three ways in which index binding takes place:

- *local* binding,
- *default* binding, and
- *context* binding.

Local binding

Local binding takes place through the use of an IN modifier at the index binding position as illustrated in the following example.

```
NettoTransport(i in SupplyCities, j in DestinationCitiesFromSupply(i)) :=
    Transport(i,j) - Transport(j,i);
```

Instead of executing the assignment for all cities i and j, it is only executed for those combinations for which city i is in SupplyCities and city j is in DestinationCitiesFromSupply(i).

Default binding

Indices can have a *default binding*. This is the binding specified in a declaration. You can specify a default binding either via the Index attribute of a set, or via the Range attribute of an Index declaration. Whenever you use an index with a default binding and do not specify a local binding, AIMMS will couple this index to its default set automatically. The following example illustrates default binding.

```
IntermediateTransportCitiesInBetween(i,j) :=
       DestinationCitiesFromSupply(i) * SupplyCitiesToDestination(j);
```

Assuming that i and j have a default binding to the set Cities, the assignment takes place for all tuples of cities (i,j).

Context binding

Whenever you use an index that has no default binding and for which you do not provide a local binding, AIMMS will try to determine a *context binding* from the context. Assume that k is an index without a default binding. Further assume that LargestTransport is an element parameter into Cities and indexed over Cities. Then the following example is an illustration of context binding.

```
LargestTransport(k) := ArgMax( j, Transport(k,j) );
```

In this assignment AIMMS will automatically bind the index k to Cities, because the identifier LargestTransport has been declared with the index domain Cities. Note that context binding will only work in indexed assignments.

Nested index binding

Index binding can be nested through the use of indexed element-valued parameters on the left-hand side of an assignment. The binding takes place in the way that you would expect, applying the same rules as for non-nested index binding. For example, given the declarations

```
ElementParameter NextCity {
   IndexDomain  : i;
   Range        : Cities;
}
ElementParameter PreviousCity {
   IndexDomain  : i;
   Range        : Cities;
}
```

the following assignment, which computes the value of PreviousCity given the contents of NextCity, will bind the nested reference to the index i.

```
   PreviousCity( NextCity(i) ) := i;
```

This binding is sparse, in the sense that the statement is only executed for those i for which NextCity(i) assumes a nonempty value.

Compatible index binding only

In general, AIMMS will never accept the use of an index in references to indexed identifiers when the binding set does not have the same root set as the index domain of the identifier. This is even the case when the elements, referenced in the particular statement, have identical names in both the binding set and the index domain. Internally, AIMMS stores a set elements as a unique (integer) number with respect to its root set, and uses this number for storing data for that element in indexed identifiers. Thus, when the root sets of the binding set and the index domain are not identical, the set element numbers will be incompatible, preventing AIMMS from referencing the correct data.

Use indirect referencing

When you want to use a binding set which is incompatible with the index domain of identifier on the left-hand side of an assignment, you should manually create an element parameter which maps elements in one root to the corresponding elements the other root set. Such a mapping can be easily created using the function `ElementCast` (discussed in Section 5.2.1), as exemplified below.

```
ElementMap(i) := ElementCast( IncompatibleRootSet, i );
```

Subsequently, you can use a nested binding through the element parameter `ElementMap` to reference elements in the index domain of the identifier on the left-hand side of an assignment, while still using the index `i` as a binding index, as illustrated in the following statement.

```
IncompatibleParameter( ElementMap(i) ) := CompatibleParameter(i);
```

Use the `ElementCast` function

Conversely, when you want to use an incompatible set element in a parameter reference on the right-hand side of an assignment, there is no direct need to create a mapping parameter. In an expression on the right of an assignment, you can use the function `ElementCast` directly at any index position, as illustrated below.

```
CompatibleParameter(i) := IncompatibleParameter( ElementCast(IncompatibleRootSet, i) );
```

Universal set

Note that you could have accomplished the same effect by creating a universal set of which all other sets are subsets. As a result, all set elements are represented as unique integer numbers with respect to the same root set, allowing the index domains of all identifiers to be referenced in a compatible manner. However, often it is not very natural to do so, and the usage of a universal set is likely to slow down the performance of AIMMS.

Index binding rules

For most situations the result of index binding is self-evident and the behavior of the system is as you would expect. Following are the precise rules for index binding.

- **Dominance rule:** Whenever index binding takes place, local binding precedes default binding, which in turn precedes context binding. If no method is applicable, a compile time error will result.

- **Intersection rule:** In indexed assignments the binding set(s) should be compatible with the index domain. The assignment will be performed for all tuples on the left-hand side that lie in the intersection of the binding set(s) and the index domain of the corresponding identifier.
- **Ordering rule:** Lag and lead operators, as well as the `Ord` and `Element` functions operate according to the order of elements in the corresponding binding set.

Chapter 10

Procedures and Functions

Functions and procedures

Functions and procedures are pieces of execution code dedicated to a specific task that can be called either from within the graphical end-user interface or from within the model text. Both functions and procedures in AIMMS can have arguments. A function returns either a scalar value or an indexed set of values, and can be used inside expressions. Procedures are more general than functions in that they can have both multiple inputs and outputs. A procedure invocation is a single statement in AIMMS, and can be used to modify the values of global identifiers.

Procedures for initiating execution

Any computation that is part of your application must be started from within a procedure. For simple applications, execution from within the predefined procedure `MainExecution` is usually sufficient to perform all tasks. However, in more complicated applications there are often many entry points, and these can best be implemented as separate procedures.

This chapter

This chapter describes how to construct and use procedures and functions in the AIMMS language. Such procedures and functions are called *internal*. In Chapter 11 you will find additional material on how to link *external* functions and procedures written in FORTRAN and C to your application.

10.1 Internal procedures

AIMMS and internal procedures

Internal procedures are pieces of execution code to perform a dedicated task. For most tasks, and particularly large ones, it is strongly recommended that you use procedures to break your task into smaller, purpose-specific tasks. This provides code structure which is easier to maintain and run. Often it is appropriate to write procedures to obtain input data from users, databases and files, to execute data consistency checks, to perform side computations, to solve a mathematical program, and to create selected reports. Procedures can be called both inside the model text and inside the graphical user interface.

Declaration and attributes

Procedures are added by inserting a special type of node in the model tree. The attributes of a Procedure specify its arguments and execution code. All possible attributes of a Procedure node are given in Table 10.1.

Attribute	Value-type	See also page
Arguments	*argument-list*	
Property	UndoSafe	
Body	*statements*	102
Comment	*comment string*	

Table 10.1: Procedure attributes

Formal arguments

The arguments of a procedure are given as a parenthesized, comma-separated list of formal argument names. These argument names are only the formal identifier names without reference to their index domains. AIMMS allows formal arguments of the following types:

- simple sets and relations, and
- scalar and indexed parameters (either element-valued, string-valued or numerical).

The type and dimension of every formal argument is not part of the argument list, and must be specified as part of the argument's (mandatory) local declaration in a declaration subnode of the procedure.

Interactive support

When you add new formal arguments to a procedure in the AIMMS Model Explorer, AIMMS provides support to automatically add these arguments as local identifiers to the procedure. For all formal arguments which have not yet been declared as local identifiers, AIMMS will pop up a dialog box to let you choose from all supported identifier types. After finishing the dialog box, all new arguments will be added as (scalar) local identifiers of the indicated type. When an argument is indexed, you still need to add the proper IndexDomain manually in the attribute form of the argument declaration.

Range checking

If the declaration of a formal argument of a procedure contains a numerical range, AIMMS will automatically perform a range check on the actual arguments based on the specified range of the formal argument.

Input or output

In the declaration of each argument you can specify its type by setting one of the properties

- Input,
- Output,

- InOut (default), or
- Optional.

AIMMS passes the values of any Input and InOut arguments when entering the procedure, and passes back the values of Output and InOut arguments. For this reason an actual Input argument can be any expression, but actual Output and InOut arguments must be parameter references or set references.

Optional arguments

An argument can be made optional by setting the property Optional in its declaration. Optional arguments are always input, and must be scalar. When an optional argument is not provided in a procedure call, AIMMS will pass its default value as specified in its declaration.

The Body attribute

In the Body attribute you can specify the sequence of AIMMS execution statements that you want to be executed when the procedure is run. All statements in the body of a procedure are executed in their order of appearance.

Example

The following example illustrates the declaration of a simple procedure in AIMMS. The body of the procedure has only been outlined.

```
Procedure ComputeShortestDistance {
    Arguments  : (City, DistanceMatrix, Distance);
    Comment    : {
        "This procedure computes the distance along the shortest path
        from City to any other city j, given DistanceMatrix."
    }
    Body: {
        Distance(j) := DistanceMatrix(City,j);

        for ( j | not Distance(j) ) do
            /*
             *  Compute the shortest path and the corresponding distance
             *  for cities j without a direct connection to City.
             */
        endfor
    }
}
```

The procedure ComputeShortestDistance has three formal arguments, which must be declared in a declaration subnode of the procedure. Their declarations within this subnode could be as follows.

```
ElementParameter City {
    Range        : Cities;
    Property     : Input;
}
Parameter DistanceMatrix {
    IndexDomain  : (i,j);
    Property     : Input;
}
Parameter Distance {
    IndexDomain  : j;
    Property     : Output;
}
```

From these declarations (and not from the argument list itself) AIMMS can deduce that

- the first actual (input) argument in a call to ComputeShortestDistance must be an element of the (global) set Cities,
- the second (input) argument must be a two-dimensional parameter over Cities × Cities, and
- the third (output) arguments must be a one-dimensional parameter over Cities.

Arguments declared over global sets

As in the example above, arguments of procedures can be indexed identifiers declared over global sets. An advantage is that no local sets need to be defined. A disadvantage is that the corresponding procedure is not generic. Procedures with arguments declared over global sets are preferred when the procedure is uniquely designed for the application at hand, and direct references to global sets add to the overall understandability and maintainability.

Arguments declared over local sets

The index domain or range of a procedure argument need not always be defined in terms of global sets. Also sets that are declared locally within the procedure can be used as index domain or range of that procedure. When a procedure with such arguments is called, AIMMS will examine the actual arguments, and pass the global domain set to the local set identifier *by reference*. This allows you to implement procedures performing generic functionality for which a priori knowledge of the index domain or range of the arguments is not relevant.

Local sets are read-only

When you pass arguments defined over local sets, AIMMS does not allow you to modify the contents of these local sets during the execution of the procedure. Because such local sets are passed by reference, this will prevent you from inadvertently modifying the contents of the global domain sets. When you do want to modify the contents of the global domain sets, you should pass these sets as explicit arguments as well.

Unit analysis of arguments

Whenever your model contains one or more Quantity declarations (see Section 32.2), AIMMS allows you to associate units of measurements with every argument. Similarly as the index domains of multidimensional arguments can be expressed either in terms of global sets, or in terms of local sets that are determined at runtime, the units of measurements of function and procedure arguments can also be expressed either in terms of globally defined units, or in terms of local unit parameters that are determined runtime by AIMMS. The unit analysis of procedure arguments is discussed in full detail in Section 32.4.1.

Local identifiers

Besides the arguments, you can also declare other local scalar or indexed identifiers in a declaration subnode of a procedure or function in AIMMS. Local identifiers cannot have a definition, and their scope is limited to the procedure or function itself.

The property RetainsValue

For each local identifier of a procedure or function that is not a formal argument, you can specify the option RetainsValue. With it you can indicate that such a local identifier must retain its last assigned value between successive calls to that procedure or function. You can use this feature, for instance, to retain local data that must be initialized once and can be used during every subsequent call to the procedure, or to keep track of the number of calls to a procedure.

Execution subnodes

In addition to AIMMS execution statements, you can include references to (named) execution subnodes to the body of a procedure. AIMMS supports several types of execution subnodes. They can either contain just execution statements or provide a graphical input form for complicated statements like the READ, WRITE and SOLVE statement. The contents of the execution subnodes will be expanded by AIMMS into the body of the procedure at the position of their references.

Top-down implementation

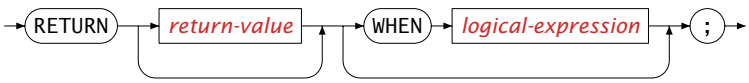
By partitioning the body of a long procedure into several execution subnodes, you can effectively implement the procedure in a self-documenting top-down approach. While the body can just contain the outermost structure of the procedure's execution, the implementation details can be hidden behind subnode references with meaningful names.

The RETURN statement

In some situations, you may want to return from a procedure or function before the end of its execution has been reached. You use the RETURN statement for this purpose. It can be subject to a conditional WHEN clause similar to the SKIP and BREAK statements in loops. The syntax follows.

Syntax

return-statement :



Return value

Procedures in AIMMS can have an (integer) return value, which you can pass by means of the RETURN statement. You can use the return value only in a limited sense: you can assign it to a scalar parameter, or use it in a logical condition in, for instance, an IF statement. You cannot use the return value in a compound numerical expression. For more details, refer to Section 10.3.

The Property attribute

In the Property attribute of internal procedures you can specify a single property, UndoSafe. With the UndoSafe property you can indicate that the procedure, when called from a page within the graphical end-user interface of a model, should leave the stack of end-user undo actions intact. Normally, procedure calls made from within the end-user interface will clear the undo stack, because such calls usually make additional modifications to (global) data based on end-user edits.

Procedures summarized

The following list summarizes the main characteristics of AIMMS procedures.

- The arguments of a procedure can be sets, set elements and parameters.
- The arguments, together with their attributes, must be declared in a local declaration subnode.
- The domain and range of indexed arguments can be in terms of either global or local sets.
- Each argument is of type Input, Output, Optional or InOut (default).
- Optional arguments must be scalar, and you must specify a default value. Optional arguments are always of type Input.
- AIMMS performs range checking on the actual arguments at runtime, based on the specified range of the formal arguments.

10.2 Internal functions

Similar to procedures

The specification of a function is very similar to that of a procedure. The following items provide a summary of their similarities.

- Arguments, together with their attributes, must be declared in a local declaration subnode.
- The domain and range of indexed arguments can be in terms of either global or local sets.
- The units of arguments can be expressed in terms of globally defined units of measurement, or in locally defined unit parameters.
- Optional arguments must be scalar, and you must specify a default value.
- AIMMS performs range checking on the actual arguments at runtime.
- Both functions and procedures can have a RETURN statement.

There are differences

There are also differences between a function and a procedure, as summarized below:

- Functions return a result that can be used in numerical expressions. The result can be either scalar-valued or indexed, and can have an associated unit of measurement.
- Functions cannot have side effects either on global identifiers or on their arguments, i.e. every function argument is of type Input by definition.

Not allowed in constraints

AIMMS only allows the (possibly multi-dimensional) result of a function to be used in constraints if none of the function arguments are variables. Allowing function arguments to be variables, would require AIMMS to compute the Jacobian of the function with respect to its variable arguments, which is not a straightforward task. External functions in AIMMS do support variables as arguments (see also Section 11.4).

Example: the Cobb-Douglas function

The Cobb-Douglas (CD) function is a scalar-valued function that is often used in economical models. It has the following form:

$$q = CD_{(a_1,\ldots,a_k)}(c_1,\ldots,c_k) = \prod_f c_f^{a_f},$$

where

q	is the quantity produced,
c_f	is the factor input f,
a_f	is the share parameter satisfying $a_f \geq 0$ and $\sum_f a_f = 1$.

In its simplest form, the declaration of the Cobb-Douglas function could look as follows.

```
Function CobbDouglas {
    Arguments  : (a,c);
    Range      : nonnegative;
    Body       : {
        CobbDouglas := prod[f, c(f)^a(f)]
    }
}
```

The arguments of the `CobbDouglas` function must be declared in a local declaration subnode. The following declarations describe the arguments.

```
Set InputFactors {
    Index        : f;
}
Parameter a {
    IndexDomain  : f;
}
Parameter c {
    IndexDomain : f;
}
```

Function attributes

The attributes of functions are listed in Table 10.2. Most of them are the same as those of procedures.

Returning the result

By providing an index domain to the function, you indicate that the result of the function is multidimensional. Inside the function you can use the function name with its indices as if it were a locally defined parameter. The result of the function must be assigned to this 'parameter'. As a consequence, the body of any function should contain at least one assignment to itself to be useful.

Attribute	Value-type	See also page
Arguments	*argument-list*	
IndexDomain	*index-domain*	42
Range	*range*	43
Unit	*unit-expression*	45
Property	RetainsValue	
Body	*statements*	102
Comment	*comment string*	

Table 10.2: Function attributes

Note that the RETURN statement cannot have a return value in the context of a function body.

The Range attribute

Through the Range attribute you can specify in which numerical, set, element or string range the function should assume its result. If the result of the function is numeric and multidimensional, you can specify a range using multidimensional parameters which depend on all or only a subset of the indices specified in the IndexDomain of the function. This is similar as for parameters (see also page 43). Upon return from the function, AIMMS will verify that the function result lies within the specified range.

The Unit attribute

Through the Unit attribute of a function you can associate a unit with the function result. AIMMS will use the unit specified here during the unit consistency check of each assignment to the result parameter within the function body, based on the units of the global identifiers and function arguments that are referenced in the assigned expression. In addition, AIMMS will use the value of the Unit attribute during unit consistency checks of all expressions that contain calls to the function at hand. You can find general information on the use of units in Chapter 32. Section 32.4.1 focusses on unit consistency checking for functions and procedures.

Example: computing the shortest distance

The procedure ComputeShortestDistance discussed in the previous section can also be implemented as a function ShortestDistance, returning an indexed result. In this case, the declaration looks as follows.

```
Function ShortestDistance {
    Arguments    : (City, DistanceMatrix);
    IndexDomain  : j;
    Range        : nonnegative;
    Comment      : {
        "This procedure computes the distance along the shortest path
         from City to any other city j, given DistanceMatrix."
    }
    Body         : {
        ShortestDistance(j) := DistanceMatrix(City,j);

        for ( j | not ShortestDistance(j) ) do
            /*
             *  Compute the shortest path and the corresponding distance
             *  for cities j without a direct connection to City.
             */
        endfor
    }
```

10.3 Calls to procedures and functions

Consistency with prototype

Functions and procedures must be called from within AIMMS in accordance with the prototype as specified in their declaration. For every call to a function or procedure, AIMMS will verify not only the number of arguments, but also whether the arguments and result are consistent with the specified domains and ranges.

Example procedure call

Consider the procedure ComputeShortestDistance defined in Section 10.1. Further assume that DistanceMatrix and ShortestDistanceMatrix are two-dimensional identifiers defined over Cities × Cities. Then the following assignment illustrates a valid procedure call.

```
for ( i ) do
    ComputeShortestDistance(i, DistanceMatrix, ShortestDistanceMatrix(i,.)) ;
endfor;
```

As you will see later on, the "." notation used in the third argument is a shorthand for the corresponding domain set. In this instance, the corresponding domain set of ShortestDistanceMatrix(i,.) is the set Cities.

Domain checking of arguments

In analyzing the resulting domains of the arguments, AIMMS takes into account the following considerations.

- Due to the surrounding FOR statement the index i is bound, so that the first argument is indeed an element in the set Cities.

- The second argument DistanceMatrix is provided without an explicit domain. AIMMS will interpret this as offering the complete two-dimensional identifier DistanceMatrix. As expected, the argument is defined over Cities × Cities.
- Because of the binding of index i, the third argument ShortestDistanceMatrix(i,.) results into the (expected) one-dimensional slice over the set Cities in which the result of the computation will be stored.

Thus, the domains of the actual arguments coincide with the domains of the formal arguments, and AIMMS can correctly compute the result.

Example function call

Now consider the function ShortestDistance defined in Section 10.1. The following statement is equivalent to the FOR statement of the previous example.

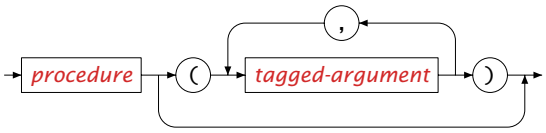
```
ShortestDistanceMatrix(i,j) := ShortestDistance(i, DistanceMatrix)(j) ;
```

In this example index binding takes place through the indexed assignment. Per city i AIMMS will call the function ShortestDistance once, and assign the one-dimensional result (indexed by j) to the one-dimensional slice ShortestDistanceMatrix(i,j).

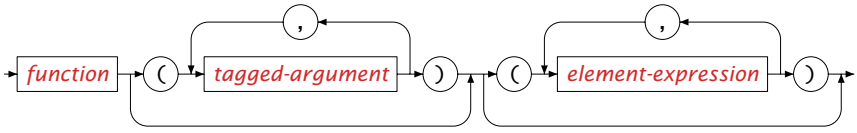
Call syntax

The general forms of procedure and function calls are identical, except that a function reference can have additional indexing.

procedure-call :



function-call :



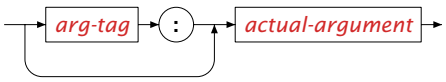
Actual arguments

Each actual argument can be

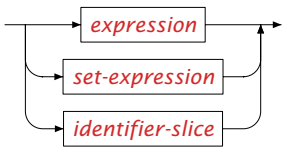
- any type of scalar expression for *scalar* arguments, and
- a reference to an identifier slice of the proper dimensions for *non-scalar* arguments.

Actual arguments can be tagged with their formal argument name used inside the declaration of the function or procedure. The syntax follows.

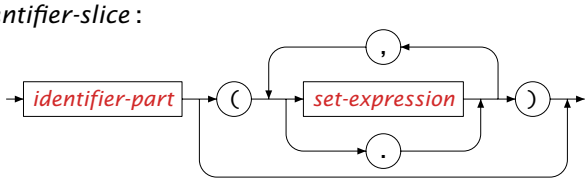
tagged-argument :



actual-argument :



identifier-slice :



Scalar and set arguments

For scalar and set arguments that are of type `Input` you can enter any scalar or set expression, respectively. Scalar and set arguments that are of type `InOut` or `Output` must contain a reference to a scalar parameter or set, or to a scalar slice of an indexed parameter or set. The latter is necessary so that AIMMS knows where to store the output value.

No slices of indexed sets

Note that AIMMS does not allow you to pass slices of an indexed set as a set arguments to functions and procedures. If you want to pass the contents of a slice of an indexed set as an argument to a procedure or function, you should assign the contents to a simple (sub)set instead, and pass that set as an argument.

Multi-dimensional arguments

For multidimensional actual arguments AIMMS only allows references to identifiers or slices thereof. Such arguments can be indicated in two manners.

- If you just enter the name of a multidimensional identifier, AIMMS assumes that you want to pass the fully dimensioned data block associated with the identifier.
- If you enter an identifier name plus
  - a ".",
  - a set element, or
  - a set expression

  at each position in the index domain of the identifier, AIMMS will pass the corresponding identifier *slice* or *subdomain*.

The "." notation

When passing slices or subdomains of a multidimensional identifier argument, you can use the "." shorthand notation at a particular position in the index domain. With it you indicate that AIMMS should use the corresponding domain set of the identifier at hand at that index position. Recall the argument `ShortestDistanceMatrix(i,.)` in the call to the procedure `ComputeShortestDistance` discussed at the beginning of this section. As the index do-

main of ShortestDistanceMatrix is the set Cities × Cities, the "." reference stands for a reference to the set Cities.

Slicing

By specifying an explicit set element or an element expression at a certain index position of an actual argument, you will decrease the dimension of the resulting slice by one. The call to the procedure ComputeShortestDistance discussed earlier in this section illustrates an example of an actual argument containing a one-dimensional slice of a two-dimensional parameter.

Dimensions must match

Note that AIMMS requires that the dimensions of the formal and actual arguments match exactly.

Subdomains

By specifying a subset expression at a particular index position of an indexed argument, you indicate to AIMMS that the procedure or function should only consider the argument as defined over this subdomain.

Example

Consider the Cobb-Douglas function discussed in the previous section, and assume the existence of a parameter a(f) and a parameter c(f), both defined over a set Factors. Then the statement

```
Result := CobbDouglas(a,c) ;
```

will compute the result by taking the product of exponents over all factors f. If SubFactors is a subset of Factors, satisfying the condition on the share parameter a(f), then the following call will compute the result by only taking the product over factors f in the subset SubFactors.

```
Result := CobbDouglas( a(SubFactors), c(SubFactors) );
```

Global subdomains

Whenever a formal argument refers to an indexed identifier defined over global sets, it could be that an actual argument in a function or procedure call refers to an identifier defined over a superset of one or more of these global sets. In this case, AIMMS will automatically restrict the domain of the actual argument to the domain of the formal argument. Likewise, if an index set of an actual argument is a real subset of the corresponding global index set of a formal argument, the values of the formal argument, when referred to from within the body of the procedure, will assume the default value of the formal argument in the complement of the index (sub)set of actual argument.

Local subdomains

Whenever a formal argument refers to an indexed identifier defined over local sets, the domain of the actual argument can be further restricted to a subdomain as in the example above. In any case, the (sub)domain of the actual argument determines the contents of the local set(s) used in the formal arguments. Note that consistency in the specified domains of the actual arguments is required when a local set is used in the index domain of several formal arguments.

Tagging arguments

In order to improve the understandability of calls to procedures and functions the actual arguments in a reference may be tagged with the formal argument names used in the declaration. In a procedure reference, it is mandatory to tag all *optional* arguments which do not occur in their natural order.

Permuting tagged arguments

Tagged arguments may be inserted at any position in the argument list, because AIMMS can determine their actual position based on the tag. The non-tagged arguments must keep their relative position, and will be intertwined with the (permuted) tagged arguments to form the complete argument list.

Example

The following permuted call to the procedure ComputeShortestDistance illustrates the use of tags.

```
for ( i ) do
    ComputeShortestDistance( Distance       : ShortestDistanceMatrix(i,.),
                             DistanceMatrix : DistanceMatrix,
                             City           : i );
endfor;
```

Using the return value

As indicated in Section 10.1 procedures in AIMMS can return with an integer return value. Its use is limited to two situations.

- You can assign the return value of a procedure to a scalar parameter in the calling procedure. However, a procedure call can never be part of a numerical expression.
- You can use the return value in a logical condition in, for instance, an IF statement to terminate the execution when a procedure returns with an error condition.

You can use a procedure just as a single statement and ignore the return value, or use the return value as described above. In the latter case, AIMMS will first execute the procedure, and subsequently use the return value as indicated.

Example

Assume the existence of a procedure AskForUserInputs(Inputs,Outputs) which presents a dialog box to the user, passes the results to the Outputs argument, and returns with a nonzero value when the user has pressed the OK button in the dialog box. Then the following IF statement illustrates a valid use of the return value.

```
if ( AskForUserInputs( Inputs, Outputs ) )
then
   ... /* Take appropriate action to process user inputs */
else
   ... /* Take actions to process invalid user input */
endif ;
```

10.3.1 The APPLY operator

Data-driven procedures

In many real-life applications the exact nature of a specific type of computation may heavily depend on particular characteristics of its input data. To accommodate such data-driven computations, AIMMS offers the APPLY operator which can be used to dynamically select a procedure or function of a given prototype to perform a particular computation. The following two examples give you some feeling of the possible uses.

Example: processing events

In event-based applications many different types of events may exist, each of which may require an event-type specific sequence of actions to process it. For instance, a ship arrival event should be treated differently from an event representing a pipeline batch, or an event representing a batch feeding a crude distiller unit. Ideally, such event-specific actions should be modeled as a separate procedure for each event type.

Example: product blending

A common action in the oil-processing industry is the blending of crudes and intermediate products. During this process certain material properties are monitored, and their computation for a blend require a property-specific *blending rule*. For instance, the sulphur content of a mixture may blend linearly in weight, while for density the reciprocal density values blend linear in weight. Ideally, each blending rule should be implemented as a separate procedure or function.

The APPLY operator

With the APPLY operator you can dynamically select a procedure or function to be called. The first argument of the APPLY operator must be the name of the procedure or function that you want to call. If the called procedure or function has arguments itself, these must be added as the second and further arguments to the APPLY operator. In case of an indexed-valued function, you can add indexing to the APPLY operator as if it were a function call.

Requirements

In order to allow AIMMS to perform the necessary dynamic type checking for the APPLY operator, certain requirements must be met:

- the first argument of the APPLY operator must be a reference to a string parameter or to an element parameter into the set AllIdentifiers,
- this element parameter must have a Default value, which is the name of an existing procedure or function in your model, and
- all other values that this string or element parameter assumes must be existing procedures or functions with the same prototype as its Default value.

Example: processing events elaborated

Consider a set of Events with an index e and an element parameter named CurrentEvent. Assume that each event e has been assigned an event type from a set EventTypes, and that an event handler is defined for each event type. It is further assumed that the event handler of a particular event type takes the appropriate actions for that type. The following declarations illustrates this set up.

```
ElementParameter EventType {
    IndexDomain    : e;
    Range          : EventTypes;
}
ElementParameter EventHandler {
    IndexDomain    : et in EventTypes;
    Range          : AllIdentifiers;
    Default        : NoEventHandlerSelected;
    InitialData    : {
          DATA { ShipArrivalEvent    : DischargeShip,
                 PipelineEvent       : PumpoverPipelineBatch,
                 CrudeDistillerEvent : CrudeDistillerBatch }
    }
}
```

The Default value of the parameter EventHandler(et), as well as all of the values assigned in the InitialData attribute, must be valid procedure names in the model, each having the same prototype. In this example, it is assumed that the procedures NoEventHandlerSelected, DischargeShip, PumpoverPipelineBatch, and CrudeDistillerBatch all have two arguments, the first being an element of a set Events, and the second being the time at which the event has to commence. Then the following call to the APPLY statement implements the call to an event type specific event handler for a particular event CurrentEvent at time NewEventTime.

```
Apply( EventHandler(EventType(CurrentEvent)), CurrentEvent, NewEventTime );
```

When no event handler for a particular event type has been provided, the default procedure NoEventHandlerSelected is run which can abort with an appropriate error message.

Use in constraints

When applied to functions, you can also use the APPLY operator inside constraints. This allows you, for instance, to provide a generic constraint where the individual terms depend on the value of set elements in the domain of the constraint. Note, that such use of the APPLY operator will only work in conjunction with external functions, which allow the use of variable arguments (see Section 11.4).

Example: product blending

Consider a set of Products with index p, and a set of monitored Properties with index q. With each property q a blend rule function can be associated such that the resulting values blend linear in weight. These property-dependent functions can be expressed by the element parameter BlendRule(q) given by

```
ElementParameter BlendRule {
    IndexDomain    : q;
    Range          : AllIdentifiers;
    Default        : BlendLinear;
    InitialData    : {
          DATA { Sulphur    : BlendLinear,
                 Density    : BlendReciprocal,
                 Viscosity  : BlendViscosity   }
    }
}
```

Thus, the computation of the property values of a product blend can be expressed by the following single constraint, which takes into account the differing blend rules for all properties.

```
Constraint ComputeBlendProperty {
    IndexDomain    : q;
    Definition     : {
        Sum[p, ProductAmount(p)  * Apply(BlendRule(q), ProductProperty(p,q))] =
        Sum[p, ProductAmount(p)] * Apply(BlendRule(q), BlendProperty(q))
    }
}
```

Depending on the precise computation in the blend rules functions for every property `q`, the `APPLY` operator may result in linear or nonlinear terms being added to the constraint.

Chapter 11

External Procedures and Functions

Why call external procedures

Even though AIMMS offers easy-to-use multidimensional data structures combined with a powerful programming language, there are often good reasons to relay parts of the execution of your model to external procedures and functions written in e.g. C/C++ or FORTRAN. The capability to call external procedures and functions in your AIMMS application allows you

- to re-use existing software (e.g. a library of financial functions, or a collection of accurate, nonlinear process models),
- to speed up selected computations by making use of dedicated data structures which are difficult to implement in AIMMS itself, and
- to provide links to external data sources (e.g. on-line data feeds or proprietary databases).

This chapter

This chapter describes the steps you have to follow for linking libraries of external procedures and functions to AIMMS. Such procedures and functions can be used to manipulate AIMMS data during the execution of a model. In addition, external libraries may contain functions that can be used inside the constraints of a nonlinear mathematical program.

11.1 Introduction

Getting started

The aim of this section is to give you a quick feel for the effort required to make a link to an external function or procedure through a short illustrative example linking a C implementation of the Cobb-Douglas function (discussed in Section 10.2) into an AIMMS application. Section 34.1 contains a more elaborate example of an external procedure which uses AIMMS API functions to obtain additional information about the passed arguments.

External procedures and functions

The interface to external procedures and functions is arranged through special `ExternalProcedure` and `ExternalFunction` declarations which behave just like internal procedures and functions. Instead of specifying a body to initiate internal AIMMS computations, the execution of external procedures and functions is relayed to the indicated procedures and functions inside one or more DLL's.

The Cobb-Douglas function

Consider the Cobb-Douglas function discussed in Section 10.2. Given the cardinality n of the set InputFactors and two arrays a and c of doubles representing the one-dimensional input arguments of the Cobb-Douglas function (both defined over InputFactors), the following simple C function computes its value.

```
double Cobb_Douglas( int n, double *a, double *c ) {
  int i;
  double CD = 1.0 ;

  for ( i = 0; i < n; i++ )
    CD = CD * pow(c[i],a[i]) ;

  return CD;
}
```

In the sequel it is assumed that this function is contained in a DLL named "Userfunc.dll".

Linking to AIMMS

In order to make the function available in AIMMS you have to declare an ExternalFunction CobbDouglasExternal, which just relays its execution to the C implementation of the Cobb-Douglas function discussed above. The declaration of CobbDouglasExternal looks as follows.

```
ExternalFunction CobbDouglasExternal {
    Arguments     : (a,c);
    Range         : nonnegative;
    DLLName       : "Userfunc.dll";
    ReturnType    : double;
    BodyCall      : Cobb_Douglas( card : InputFactors, array: a, array: c );
}
```

The arguments a and c must be declared in the same way as for the internal CobbDouglas function discussed on page 141, with the exception that for the external implementation we will also compute the Jacobian with respect to the argument c(f). For this reason, the argument c(f) is declared as a Variable.

```
Set InputFactors {
    Index        : f;
}
Parameter a {
    IndexDomain  : f;
}
Variable c {
    IndexDomain  : f;
}
```

Explanation

The translation type "card" of the set argument InputFactors causes AIMMS to pass the cardinality of the set as an integer value to the external function Cobb_Douglas. The translation type "array" of the arguments a and c are instructions to AIMMS to pass these arguments as full arrays of double precision values. As function arguments are always of type Input, AIMMS will disregard any changes made to the arguments by the external function. The double return value of the C function Cobb_Douglas will become the result of the function CobbDouglasExternal.

Calling external functions

After the declaration of an external function or procedure you can use it as if it were an internal function or procedure. Thus, to call the external function CobbDouglasExternal in the body of a procedure the following statement suffices.

```
CobbDouglasValue := CobbDouglasExternal(a,c) ;
```

Of course, any two (possibly sliced) identifiers with single common index domain could have been used as arguments. AIMMS will determine this common index domain, and pass its cardinality to the external function.

Use in constraints

Unlike internal functions, external functions can be called inside constraints. To accomplish this, the declaration has to be extended with a DerivativeCall attribute. For this attribute you specify the external call that has to be made when AIMMS also needs the partial derivatives of all variable arguments inside constraints of mathematical programs. In the absence of a DerivativeCall attribute, AIMMS will use a differencing scheme to estimate these derivatives. The details of using external functions in constraints, as well as the obvious extension to compute the derivative of the Cobb-Douglas function directly, are given in Section 11.4.

Setting up external libraries

Once you have developed a collection of external functions and procedures, it may be a good idea to make this available in the form of a library for use in AIMMS applications. In this way, the users of your library do not have to spend any time translating their AIMMS arguments into external arguments of the appropriate type in the external procedure and function declarations.

Save library as include file

To provide a library as an entity on its own, you can store all the external procedures and functions in a separate model section, and save this section as a source file. The functions and procedures in the library can then be made available by simply including this source file into a model.

Hiding the interface

When you want to protect the interface to your external library, you can accomplish this by encrypting the include file containing the function library (see also the AIMMS User's Guide). Thus, the interface to the external library becomes invisible, effectively preventing misuse of the library outside AIMMS.

11.2 Declaration of external procedures and functions

External procedures and functions

External procedures and functions are special types of nodes in the model tree. They have the same attributes as internal procedures and functions with the exception of the Body and Derivative attributes, which are replaced by the attributes in Table 11.1.

Attribute	Value-type	See also page
DllName	*string, file-identifier*	
ReturnType	integer, double	
Property	FortranConventions, UndoSafe	140
BodyCall	*external-call*	
DerivativeCall	*external-call*	

Table 11.1: Additional attributes of external procedures and functions

The DllName attribute

With the mandatory DllName attribute you can specify the name of the DLL which contains the external procedure or function to which you want to make a link in your AIMMS application. The value of the attribute must be a string, a string parameter, or a File identifier, representing the path to the external DLL.

Search path

If you only specify a DLL name, AIMMS will search for the DLL in all directories in the AIMMSUSERDLL environment variable, and the PATH environment variable on Windows, or the LD_LIBRARY_PATH environment variables on Linux, respectively. In addition, on Windows, AIMMS will also search for the DLL in the project folder. If you specify a relative path including a folder (possibly ./), AIMMS will take this path relative to the project folder. If you specify an absolute path, AIMMS will try to open the DLL at the specified location.

File identifier and unit conventions

When you use a File identifier to specify an external DLL name, AIMMS will use the Convention attribute of that File identifier (if specified) to pass numeric values to any procedure or function in that DLL according to the specified unit convention (see also Section 32.8). When the DLL name has not been specified through a File identifier, or when its Convention attribute is left empty, AIMMS will use the unit convention specified for the main model.

Default argument scaling

Without any such convention, AIMMS will use the default convention, i.e. arguments will be scaled according to the unit specified for each argument, and AIMMS will assume that the result of an external function is scaled according to the unit specified in its Unit attribute. Unit analysis for functions and procedures is discussed in full detail in Section 32.4.1.

The ReturnType attribute

The ReturnType indicates the type of any *scalar* numerical value returned by the DLL function. The possible values are integer and double. AIMMS will use the value returned by the DLL function either as the return value of the ExternalProcedure, or as the (numerical) function value of the ExternalFunction, whichever is applicable. If you do not specify the ReturnType attribute, AIMMS will discard any value returned by the function.

Restricted use

You cannot directly use the returned value of a DLL function as the function value of an ExternalFunction when its return value is either an indexed parameter, a set, a set element or a string. In such cases you must pass the function name as an additional external argument to the DLL function, and specify how the function value must be dealt with.

Example

Consider a C function Cobb_Douglas_Arg with prototype

```
void Cobb_Douglas_Arg( int n, double *a, double *c, double *CDValue );
```

which passes the Cobb-Douglas function value through the argument CDValue instead of as the return value. In this example CDValue is a scalar, which could have been passed as the result of the DLL function as well. The following ExternalFunction declaration provides a link with Cobb_Douglas_Arg and obtains its function value via the argument list.

```
ExternalFunction CobbDouglasArgument {
    Arguments     : (a,c);
    Range         : nonnegative;
    DllName       : "Userfunc.dll";
    BodyCall      : {
        Cobb_Douglas_Arg( card : InputFactors, array: a, array: c,
            scalar: CobbDouglasArgument );
    }
}
```

The Property attribute

With the Property attribute you can specify through the FortranConventions property whether the external function is based on FORTRAN calling conventions. By default, AIMMS will assume that the DLL function is written in a C-like languages such as C, C++ or PASCAL. The precise differences between both calling conventions are explained in full detail in Section 11.5. In addition, for external procedures, you can specify the UndoSafe property. The semantics of the UndoSafe property is discussed in Section 10.1.

Formal argument types

As with internal procedures and functions, all formal arguments of an external procedure or function must be declared as local identifiers. AIMMS supports the following identifier types for formal arguments of external procedures and functions:

- simple sets and relations,
- scalar and indexed Parameters,
- scalar and indexed Variables (external functions only), and
- Handles (external procedures only).

Argument handling

Many details regarding the handling of arguments of internal procedures and functions also apply to external procedures and functions. Thus, arguments of external procedures and functions can be defined over global and local sets, and their associated units of measurement can be specified in terms of either global units or locally defined unit parameters, completely similar to internal procedures and functions (see Section 10.1).

Handle arguments

The Handle identifier type is only supported for formal arguments of external procedures, i.e. it is not possible to declare global identifiers of type Handle. The following rules apply:

- Handle arguments are always declared as scalar local identifiers,
- Handle arguments can only be passed to the DLL function as an integer Handle (see below), and
- the actual argument in a call to the external procedure corresponding to a formal Handle argument can be a (sliced) reference to an identifier in your model *of any type and of any dimension*.

Handle arguments allow you to completely circumvent any type checking on actual arguments with respect to the dimension and the respective index domains of the corresponding formal arguments in the call to an external procedure. As a result of this, however, the actual data transfer of Handle arguments to the DLL function must completely take place via the AIMMS API (see also Chapter 34).

The BodyCall attribute

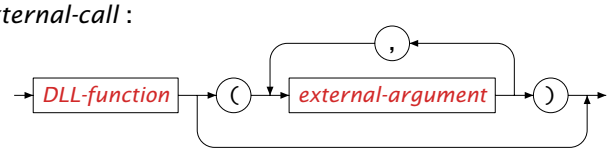
In the mandatory BodyCall attribute you must specify the call to the DLL procedure or function, to which the execution of the ExternalProcedure or Function must be relayed. Such an external call specifies:

- the name of the DLL function or procedure that must be called, and
- how the actual AIMMS arguments must be translated into arguments suitable for the DLL function or procedure.

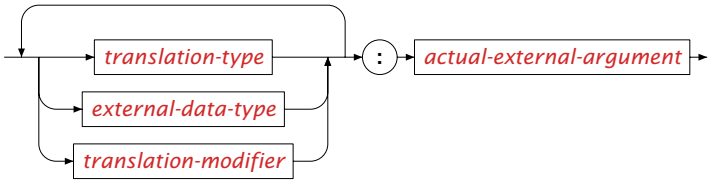
Any external call must be specified according to the syntax below. In the Model Explorer, you can specify all components of the BodyCall attribute using a wizard which will guide you through most of the necessary detail.

Syntax

external-call :



external-argument :



Mandatory translation type

The mandatory translation type indicates the type of the external argument into which the actual argument must be translated before being passed to the external procedure. The following translation types are supported.

- **scalar**: the actual scalar AIMMS argument is passed on as a scalar of the indicated external data type.
- **literal**: the literal specified in the external call is passed on as a scalar of the indicated external data type, i.e. a `literal` argument does *never* correspond to an actual AIMMS argument, but is specified directly in the `BodyCall` attribute.
- **array**: the AIMMS argument is passed on as an array of values according to the indicated translation type and external data type. The precise manner in which the translation takes place is discussed below.
- **card**: the cardinality of a set argument is passed on as an integer value. The set argument can be either a set passed as an actual AIMMS argument or the domain set of a multi-dimensional parameter passed as an actual argument.
- **handle**: an integer handle to a (sliced) set or parameter argument is passed on. Within the external procedure you must use functions from the AIMMS API (see also Chapter 34) to obtain the dimension, domain and range associated with the handle, or to retrieve or change its data values.
- **work**: an array of the indicated type is passed as a temporary workspace to the external procedure. The actual argument must be an integer expression and is interpreted as the size of the array to be passed on. This translation type is useful for programmers of languages such as standard F77 FORTRAN which lack facilities for dynamic memory allocation.

Actual external argument

The actual external argument specified in an external argument of the `BodyCall` attribute can be

- a reference to a formal argument of the `ExternalProcedure` at hand (for the `scalar`, `array`, `card`, `handle` and `work` translation types),
- a reference to a domain set of a formal multi-dimensional argument of the `ExternalProcedure` at hand (for the `card` translation type), or
- an integer, double or string literal (such as `12345`, `123.45` or `"This is a string"`) directly specified within the `BodyCall` attribute (for the `literal` translation type).

Input-output type

For every formal argument of an ExternalProcedure, you can specify its associated *input-output* type through the Input, InOut (default) or Output properties in the Propert attribute of the local argument declaration. With it, you indicate whether or not AIMMS should consider any changes made to the argument by the DLL function. For each input-output type, AIMMS performs the following actions:

- **Input**: AIMMS initializes the external argument, but discards all changes made to it by the DLL function,
- **InOut**: AIMMS initializes the external argument, and passes back to the model the values returned by the DLL function, or
- **Output**: AIMMS allocates memory for the external argument, but does not initialize it; the values returned by the DLL function are passed back to the model.

As with internal functions, all ExternalFunction arguments are Input by definition. The return value of an ExternalProcedure and the function value of an ExternalFunction are considered as an (implicit) Output argument when passed to the DLL function as an external argument.

External data type

In translating AIMMS arguments into values (or arrays of values) suitable as arguments for an external procedure or function, AIMMS supports the external data types listed in Table 11.2.

External data type	Passed as
integer	4-byte (signed) integer
double	8-byte double precision floating number
string	C-style string
integer8	1-byte (signed) integer
integer16	2-byte (signed) integer
integer32	4-byte (signed) integer

Table 11.2: External data types

Allowed combinations

Not all combinations of input-output types, translation types and external data types are supported (or even useful). Table 11.3 describes all allowed combinations, as well as the resulting argument type that is passed on to the external procedure. The external data types printed in bold are the default, and can be omitted if appropriate. Throughout the table, the data type integer can be replaced by any of the other integer types integer8, integer16 or integer32.

Allowed types			AIMMS argument	Passed as
translation	input-output	data		
scalar	input	integer **double** string	scalar expression	integer double string
	inout output	integer **double** string	scalar reference	integer pointer double pointer string
literal	—	integer **double** string	—	integer double string
card	—	—	set, parameter	integer
array	input **inout** output	integer **double**	parameter	integer array double array
		integer string	element parameter set	integer array string array
		string	string/unit parameter	string array
handle	input **inout** output	—	set, parameter, handle	integer
work	—	integer **double**	integer expression	integer array double array

Table 11.3: Allowed combinations of translation, input-output and data types

Passing array arguments

When you are passing a multidimensional AIMMS identifier to an external procedure or function as a `array` argument, AIMMS passes a one-dimensional buffer in which all values are stored in a manner that is compatible with the storage of multidimensional arrays in the language which you have specified through the `Property` attribute. The precise array numbering conventions for both C-like and FORTRAN arrays are explained in Section 11.5.

Encoding of string arguments

The strings communicated with your DLL have an encoding. This encoding is set by the option `external_string_character_encoding`, which has a default of `UTF8`. This option can be overridden by using the `Encoding` attribute of string parameters, similar to the `Encoding` attribute of a `File`, see Page 497. On Windows, using the encoding `UTF-16LE` and on Linux, using the encoding `UTF-32LE`, the strings are passed as a `wchar_t*` array, otherwise the strings are passed as a `char *` array.

Output string arguments

When you pass a scalar or multidimensional output string argument, AIMMS will pass a single `char` buffer of fixed length, or an array of such buffers. The

length is determined by the option external function string buf size. The default of this option is 2048. You must use the C function strcpy or a similar function to copy the string data in your DLL to the appropriate char buffer associated with the output string argument.

Full versus sparse data transfer

When considering your options on how to pass a high-dimensional parameter to an external procedure, you will find that passing it as an array is often not the best solution. Not only will the memory requirements grow rapidly for increasing dimension, but also running over all elements in the array inside your DLL function may turn out to be a very time-consuming process. In such a case, it is much better practice to pass the argument as an integer handle, and use the AIMMS API functions discussed in Section 34.4 to retrieve only the nondefault values associated with the handle. You can then set up your own sparse data structures to deal with high-dimensional parameters efficiently.

Translation modifiers ...

In addition to the translation types, input-output types and external data types you can specify one or more translation modifiers for each external argument. Translation modifiers allow you to slightly modify the manner in which AIMMS will pass the arguments to the DLL function. AIMMS supports translation modifiers for specifying the precise manner in which

- special values,
- the data associated with handles, and
- set elements,

are passed.

... for special values

When a parameter or variable that you want to pass to an external DLL contains special values like ZERO or INF, AIMMS will, by default, pass ZERO as 0.0, INF and -INF as ±1.0e150, and will not pass any of the values NA and UNDF. When you specify the translation modifier retainspecials, AIMMS will pass all special numbers by their internal representation as a double precision floating point number. You can use the AIMMS API functions discussed in Section 34.4 to obtain the MapVal value (see also Table 6.1) associated with each number. The translation modifier retainspecials can be specified for numeric arguments that are passed either as a full array or as an integer handle.

... for handles

When passing a multidimensional identifier handle to an external DLL, AIMMS can provide several methods of access to the data associated with the handle by specifying one of the following translation modifiers:

- ordered: the data retrieval functions will pass the data values according to the particular ordering imposed any of the domain sets of the identifier associated with the handle. By default, AIMMS will use the natural ordering determined by the data entry order of all domain sets.

- raw: the data retrieval functions will also pass inactive data (see also Section 25.3). By default, AIMMS will not pass inactive data.

The details of ordered versus unordered and raw data transfer are discussed in full detail in Section 34.4.

... for set elements

AIMMS can pass set elements (in the context of element parameters and sets) to external procedures in various manners. More specifically, set elements can be translated into:

- an integer external data type, or
- a string external data type.

When the external data type is string, AIMMS will pass the element name for each set element. Transfer of element names is always input only. In general, when the external data type is integer, AIMMS can pass either

- the ordinal number with respect to its associated subset domain (ordinalnumber modifier), or
- the element number with respect to its associated root set (elementnumber modifier).

Alternatively, when set elements are passed in the context of a set you can specify the indicator modifier in combination with the integer external data type. This will result in the transfer of a multidimensional binary parameter which indicates whether a particular tuple is or is not contained in the set.

Passing element parameters

When you pass an element parameter as an integer scalar or array argument, AIMMS will assume the ordinalnumber modifier by default. When passed as integer, element parameters can be input, output or inout arguments. When element parameters are passed as string arguments, they can be input only.

When to use

Element numbers and ordinal numbers each can have their use within an DLL function. Element numbers remain identical throughout a modeling session using a single data set, regardless of addition and deletion of set elements, or any change in set ordering. For this reason, it is best to use element numbers when the set elements need to be used in multiple calls of the DLL function. Ordinal numbers, on the other hand, are the most convenient means for passing permutations that are used within the current external call only. With it, you can directly access a permuted reference in other array arguments.

Passing set arguments

Sets can be passed as array arguments to an external DLL function. When passing set arguments, you have to make a distinction between one-dimensional root sets, one-dimensional subsets (both either simple or relation), and multi-dimensional subsets and indexed sets. The following rules apply.

Pass as one-dimensional array

One-dimensional root sets and subsets can be passed as a one-dimensional array of length equal to the cardinality of the set. To accomplish this, you can must pass such a set as

- an array of `integer` numbers, representing either the ordinal or element numbers of each element in the set (using the `ordinalnumber` or `elementnumber` modifier), or
- a `string` array, representing the names of all elements in the set.

One-dimensional set arguments passed in this manner can only be input arguments. As a specific consequence, you cannot modify the contents of root sets passed as array arguments.

Pass as indicator parameter

You can pass any subset (whether it is simple, relation or indexed) as a multidimensional `integer indicator` array defined over its respective domain sets, indicating whether a particular tuple of domain set elements is contained in the subset (value equals 1) or not (value equals 0). The dimension of such indicator parameters is given by the following set of rules:

- the dimension for a *simple subset* is 1,
- the dimension for a multidimensional relation is the dimension of the Cartesian product of which the set is a subset,
- the dimension of an *indexed set* is the dimension of the index domain of the set plus 1.

Set arguments passed as an `indicator` argument can be of input, output, or in-out type. In the latter two cases modifications to the 0-1 values of the indicator parameter are translated back into the corresponding element memberships of the subset.

Set argument defaults

When you pass set arguments to an external DLL, AIMMS will assume no default translation methods when the set is passed as an `integer` array, as each type of set does not allow every translation method. For integer set arguments you should therefore always specify one of the translation modifiers `ordinalnumber`, `elementnumber` or `indicator`.

Passing set handles

Sets can also be passed by an integer handle. AIMMS offers various API functions (see also Section 34.2) to obtain information about the domain of the set, its cardinality and elements, and to add or remove elements to the set.

11.3 WIN32 calling conventions

WIN32 *calling conventions*

The 32-bit Windows environment (WIN32) supports several calling conventions that influence the precise manner in which arguments are passed to a function, and how the return value must be retrieved. When calling an external function

or procedure in this environment, AIMMS will *always* assume the WINAPI calling convention. The following macro in C makes sure that the WINAPI calling convention is used. That same macro also makes sure that the function or procedure is automatically exported from the DLL.

```
#include <windows.h>
#define DLL_EXPORT(type) __declspec(dllexport) type WINAPI
```

You can add this macro to the implementation of any function that you want to call from within AIMMS, as illustrated below.

```
DLL_EXPORT(double) Cobb_Douglas( int n, double *a, double *c )
{
    /* Implementation of Cobb_Douglas goes here */
}
```

Prevent C++ *name mangling*

By default, C++ compilers will perform a process referred to as *name mangling*, modifying each function name in your source code according to its prototype. By doing this, C++ is able to deal with the same function name defined for different argument types. If you want to export a DLL function to AIMMS, however, you must prevent name mangling to take place, ensuring that AIMMS can find the exported function name within the DLL. You can do this by declaring the prototype of the function using the following macro, which accounts for both C and C++.

```
#ifdef __cplusplus
#define DLL_EXPORT_PROTO(type) extern "C" __declspec(dllexport) type WINAPI
#else
#define DLL_EXPORT_PROTO(type) extern __declspec(dllexport) type WINAPI
#endif
```

Thus, to make sure that a C++ implementation of Cobb_Douglas is exported without name mangling, declare its prototype as follows before providing the function implementation.

```
DLL_EXPORT_PROTO(double) Cobb_Douglas( int n, double *a, double *c );
```

Function declarations like this are usually stored in a separate header file. Note that along with this prototype declaration, you must still use the DLL_EXPORT macro in the implementation of Cobb_Douglas.

DLL initialization

When your external DLL requires initialization statements to be executed when the DLL is loaded, or requires the execution of some cleanup statements when the DLL is closed, you can accomplish this by adding a function DllMain to your DLL. When the linker finds a function named DllMain in your DLL, it will execute this function when opening and closing the DLL. The following example provides a skeleton DllMain implementation which you can directly copy into your DLL source code.

```
#include <windows.h>

BOOL WINAPI DllMain(HINSTANCE hdll, DWORD reason, LPVOID reserved)
{
    switch( reason ) {
       case DLL_THREAD_ATTACH:
           break;
       case DLL_PROCESS_ATTACH:
           /* Your DLL initialization code goes here */
           break;
       case DLL_THREAD_DETACH:
           break;
       case DLL_PROCESS_DETACH:
           /* Your DLL exit code goes here */
           break;
    }
    return 1; /* Return 0 in case of an error */
}
```

To prevent name mangling to take place, you can best declare the function DllMain as follows.

```
#ifdef __cplusplus
extern "C" BOOL WINAPI DllMain(HINSTANCE hdll, DWORD reason, LPVOID reserved);
#else
BOOL WINAPI DllMain(HINSTANCE hdll, DWORD reason, LPVOID reserved);
#endif
```

11.4 External functions in constraints

Variable arguments

AIMMS allows you to use external functions in the constraints of a mathematical program. To accommodate this, AIMMS makes a distinction between function arguments of type Parameter and arguments of type Variable. When a function is executed as part of an expression in an ordinary assignment, AIMMS makes no distinction between both types of arguments. In the context of a mathematical program, however, AIMMS will provide the solver with the derivative information for all variable arguments of the function, while it will not do so for parameter arguments. The actual computation of the derivatives is explained in the next section.

11.4.1 Derivative computation

Functions in constraints

Whenever you use external functions with variable arguments in constraints of a mathematical program, the following rules apply.

- AIMMS requires that the mathematical program dependent on these constraints be declared as nonlinear.
- All the actual variable arguments must correspond to formal arguments which have been locally declared as Variables.

If you fail to comply with these rules, a compiler error will result.

Providing derivatives

During the solution process of a mathematical program containing such functions, partial derivative information of the function with respect to all the variable arguments must be passed to the solver. AIMMS supports three methods to compute the derivatives of a function:

- you provide the actual statements for computing the derivatives as a part of the function declaration,
- AIMMS estimates the derivatives using a simple differencing scheme.

The `DerivativeCall` attribute

In the `DerivativeCall` attribute of an external function you can specify the call to the DLL procedure or function, to which the derivative computation must be relayed. The syntax of the `DerivativeCall` attribute is the same as that of the `BodyCall`, and is most conveniently completed using the wizard in the Model Explorer.

Function value and derivative

If the nonlinear solver only needs a function value, AIMMS will simply call the function specified in the `BodyCall` attribute. If the nonlinear solver requests derivative information as well, AIMMS will only call the function specified in the `DerivativeCall` attribute, and require that this function compute the function value as well. By combining these two computations in a single call, AIMMS allows you to take advantage of any possible optimization that can be obtained in your code from computing the function value and derivative at the same time.

The `.Derivative` suffix

For every function argument which is a variable, you must assign the partial derivative value(s) to the `.Derivative` suffix of that variable. Note that this will have an impact on the number of indices. If the result of a block-valued function is m-dimensional, the derivative information with respect to an n-dimensional variable argument will result in an $(m+n)$-dimensional identifier holding the derivative.

Abstract example

Consider a function `f` with an index domain $(i_1, \ldots, i_m)$ and a variable argument `x` with index domain $(j_1, \ldots, j_n)$. Then the matrix with partial derivatives of `f` with respect to the argument `x` must be provided as assignments to the suffix `x.Derivative`$(i_1, \ldots, i_m, j_1, \ldots, j_n)$. Each element of this identifier represents the partial derivative

$$\frac{\partial \mathtt{f}(i_1, \ldots, i_m)}{\partial \mathtt{x}(j_1, \ldots, j_n)}$$

Cobb-Douglas function revisited

Consider the Cobb-Douglas function discussed above. Although AIMMS is capable of computing its partial derivatives automatically, you may verify that the derivative with respect to argument c_i can also be written more compactly as follows:

$$\frac{\partial q}{\partial c_i} = \frac{a_i}{c_i} CD(c_1, \ldots, c_k)$$

Implementation in C

Consider the following C function Cobb_Douglas_Der which computes the Cobb-Douglas function and, if required, also the partial derivatives with respect to the input argument c. The function Cobb_Douglas_No_Der is added to support computation of the Cobb-Douglas function without derivatives.

```
double Cobb_Douglas_Der( int n, double *a, double *c, double *c_der ) {
  int i;
  double CD = 1.0 ;

  for ( i = 0; i < n; i++ )
    CD = CD * pow(c[i],a[i]) ;

  /* Check if derivatives are needed */
  if ( c_der )
    for ( i = 0; i < n; i++ )
      c_der[i] = CD * a[i] / c[i] ;

  return CD;
}

double Cobb_Douglas_No_Der( int n, double *a, double *c ) {
  return Cobb_Douglas_Der( n, a, c, NULL );
}
```

Always skip unwanted derivatives

Note that in the above example the derivative computation is skipped whenever the pointer c_der is null. You should *always* check for this condition when implementing a derivative computation, because AIMMS will pass a null pointer (and hence reserve no memory for storing the derivative) whenever the corresponding actual argument is not a variable but a parameter.

. . . in FORTRAN *code*

When an internal function makes a call to a FORTRAN procedure to compute derivative values, then it is not so easy to discover the presence of null pointer argument. To overcome this, you can call your FORTRAN procedure from within a wrapper function written in C, and provide your FORTRAN code with the information whether or not derivatives need to be computed for a particular variable argument via an additional argument to your FORTRAN routine.

Passing derivative arguments

To pass the partial derivatives computed in the external procedure back to AIMMS, the argument list of the external procedure called in the Derivative attribute of the internal function should contain arguments for the .Derivative suffices of all variable arguments. AIMMS will implicitly consider such derivative arguments as Output arguments. They can be passed either as a full array or as an integer handle. In the latter case AIMMS API functions have to be used to pass back the relevant partial derivatives (see also Chapter 34).

Example continued

The following external function declaration provides an interface to the above Cobb-Douglas function with derivative computations, which is ready to be used both inside and outside the context of constraints.

```
ExternalFunction CobbDouglasPlusDerivative {
    Arguments       : (a,c);
    Range           : nonnegative;
    DLLName         : "Userfunc.dll";
    ReturnValue     : double;
    BodyCall        : Cobb_Douglas_No_Der( card : InputFactors, array: a, array: c );
    DerivativeCall  : {
        Cobb_Douglas_Der( card : InputFactors, array: a,
            array: c, array: c.Derivative );
    }
}
```

Numerical differencing

When the DerivativeCall attribute to compute the derivatives of an external function has not been specified, AIMMS employs a simple differencing scheme to estimate the derivatives. For example, if AIMMS requires the derivative of a function $f(x_1, x_2, \ldots, x_k)$ at the point $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k)$, then AIMMS will approximate each partial derivative as follows:

$$\frac{\partial}{\partial x_i} f(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k) \approx \frac{f(\bar{x}_1, \ldots, \bar{x}_i + \varepsilon, \ldots, \bar{x}_k) - f(\bar{x}_1, \ldots, \bar{x}_k)}{\varepsilon}$$

where ε is the current value of the global option Differencing_Delta.

Disadvantages of numerical differencing

While the numerical differencing scheme does not require any action from the user, there are two distinct disadvantages.

- First of all, numerical differencing is not always a stable process, and the results may not be accurate enough. As a result, a nonlinear solver may have trouble converging to a solution.
- Secondly, the process can be computationally very expensive.

In general, it is recommended that you do not rely on numerical differencing. This is especially the case when the function body is quite extensive, or when the function, at the individual level, has a lot of variable arguments or contains conditional loops.

11.5 C versus FORTRAN conventions

Language conventions

For any external procedure or function you can specify whether the DLL procedure or function to which the execution is relayed, is written in C-like languages (such as C and C++) or FORTRAN (see also Section 11.2). For FORTRAN code AIMMS will make sure that

- scalar values are always passed by reference (i.e. as a pointer), and
- multidimensional arrays are ordered in a FORTRAN-compatible manner.

By default, AIMMS will use C conventions when passing arguments to the DLL procedure or function.

Strings excluded

AIMMS will not directly translate strings into FORTRAN format, because most FORTRAN compilers use their own particular string representation. Thus, if you want to pass strings to a `fortran` subroutine, you should write your own C interface which converts C strings into the format appropriate for your FORTRAN compiler.

Array dimensions and ordering

When a multidimensional parameter (or parameter slice) is specified as a `array` argument to an external procedure, AIMMS passes an array of the specified type which is constructed as follows. If the actual argument has n remaining (i.e. non-sliced) dimensions of cardinality $N_1, \ldots, N_n$, respectively, then the associated values are passed as a (one-dimensional) array of length $N_1 \cdots N_n$. The value associated with the tuple $(i_1, \ldots, i_n)$ is mapped onto the element

$$i_n + N_n(i_{n-1} + N_{n-1}(\cdots(i_2 + N_2 i_1)\cdots))$$

for running indices $i_j = 0, \ldots, N_j - 1$ (C-style programming). For PASCAL-like languages (with indices running from $1, \ldots, N$) all running indices in this formula must be decreased by 1, and the final result increased by 1. This ordering is compatible with the C declaration of e.g. the multidimensional array

```
double arr[N1][N2]...[Nn];
```

Multidimensional example in C

The C function `ComputeAverage` defined below computes the average of a 2-dimensional parameter `a(i,j)` passed as an argument in AIMMS.

```
DLL_EXPORT(void) ComputeAverage( double *a, int card_i, int card_j, double *average )
{ int i, j;
  double sum_a = 0.0;

#define __A(i,j)    a[j + i*card_j]

  for ( i = 0; i < card_i; i++ )
    for ( j = 0; j < card_j; j++ )
      sum_a += __A(i,j);

  *average = sum_a / (card_i*card_j);
}
```

Within your AIMMS model, you can call this procedure via an external procedure declaration `ExternalAverage` defined as follows.

```
ExternalProcedure ExternalAverage {
    Arguments     : (x,res);
    DLLName       : "Userfunc.dll";
    BodyCall      : ComputeAverage(double array: x, card: i, card: j, double scalar: res);
}
```

where the argument `x` and `res` are declared as

```
Parameter x {
    IndexDomain   : (i,j);
    Property      : Input;
}
Parameter res {
    Property      : Output;
}
```

Fortran array ordering

When you specify the FORTRAN language convention for an external procedure, AIMMS will order the array passed to the external procedure such that the tuple $(i_1, \ldots, i_n)$ is mapped onto the element

$$i_1 + N_1(i_2 - 1 + N_2(\cdots(i_{n-1} - 1 + N_{n-1}(i_n - 1))\cdots))$$

for running indices $i_j = 1, \ldots, N_j$. This is compatible with the default storage of multidimensional arrays in FORTRAN, and allows you to access such array arguments using the ordinary multidimensional notation.

Example

Consider a parameter a(i,j), where the index i is associated with the set {1, 2} and j with the set {1, 2, 3}. When this parameter is passed as a array argument to an external procedure, the resulting array (as a one-dimensional array with 6 elements) is ordered as follows in the C convention (default).

Element #	0	1	2	3	4	5
Value	a(1,1)	a(1,2)	a(1,3)	a(2,1)	a(2,2)	a(2,3)

With the FORTRAN language convention, the ordering is changed as follows.

Element #	1	2	3	4	5	6
Value	a(1,1)	a(2,1)	a(1,2)	a(2,2)	a(1,3)	a(2,3)

Part IV

Sparse Execution

Chapter 12

The AIMMS Sparse Execution Engine

Learning about sparse execution

In this chapter, we look under the hood of the AIMMS sparse execution engine. It is not only interesting to know what AIMMS can do, but also, to some extent, how it is done. An understanding of the inner workings of the AIMMS execution engine may also give you a framework for understanding why some formulations of AIMMS statements are more efficient than others, leading to more efficient applications. Increasing the efficiency of your application will help make it a success.

Sparse matrix technology

The AIMMS execution system borrows and extends two simple but powerful concepts from sparse matrix technology. These concepts are:

- only store the non-zero values, and
- do not compute `0+0` and `0*`x (x any number), because these computations always result in 0.0 and these results are consequently not stored.

AIMMS extensions

The AIMMS extensions to these borrowed concepts are that:

- only non-default values are stored, where the default is a selectable value, and
- many operations such as `OR` and `AND` have similar behaviors as + and * respectively.

Note, however, that other operators, such as the / and = operators, will have to consider zeros:

- the computation `0.0 / 0.0` results in `UNDF`, and
- the computation `0.0 = 0.0` results in `1.0`

The results of these computations are not equal to 0.0 and need to be stored; and therefore 'sparse execution' is not applicable to these operators.

12.1 Storage and basic operations of the execution engine

In this section we present, in a step-by-step manner, the operations that, when combined, build up the AIMMS sparse execution engine. The data storage method with which these operations work is called an *ordered view*.

Ordered view

The AIMMS execution engine stores the data according to the concept of an ordered view. An ordered view is an ordered, sparse collection of the non-default elements of an identifier. The order is the lexicographical order of the indices of that identifier. Because of this order:

- the non-default elements of the identifier can be visited in a lexicographic order one at a time, and
- a particular tuple can be found efficiently using values for the indices.

Running example

The running example, used in this section and presented below, contains the two parameters A(i,j) and B(i,j), where i and j are indices in a set S containing the elements {a1..a5}. The default values of these parameters are 0.0, and they contain the following data:

```
A(i,j) := data table
     a1 a2 a3 a4 a5
  !  -- -- -- -- --
  a1     2        5
  a2  2     3  2
  a3
  a4  4
  a5                  ;
```

```
B(i,j) := data table
     a1 a2 a3 a4 a5
  !  -- -- -- -- --
  a1     3        2
  a2
  a3  5     1  2
  a4  4
  a5                  ;
```

The ordered views of A and B are presented in the composite tables below:

```
Composite table:
     i  j A
  ! -- -- -
    a1 a2 2
    a1 a5 5
    a2 a1 2
    a2 a3 3
    a2 a4 2
    a4 a1 4 ;
```

```
Composite table:
     i  j B
  ! -- -- -
    a1 a2 3
    a1 a5 2
    a3 a1 5
    a3 a3 1
    a3 a4 2
    a4 a1 4 ;
```

Like an index in relational databases

There is nothing really new here; an ordered view corresponds to an relational table in database terminology, with a (database) index on the primary keys i and j. A characteristic of both representations is that they can be easily searched given explicit values for i and j.

Basic operations

In the following sections, we will classify the algebraic operations in AIMMS according to their behavior in the AIMMS sparse execution engine, and discuss the effects of combining multiple operations or changing the natural index order.

12.1.1 The + operator: union behavior

First statement

The first statement in the running example is the simple addition of the matching elements resulting in parameter C(i,j):

```
C(i,j) := A(i,j) + B(i,j);
```

Merging rows

As illustrated in Figure 12.1, this statement can be executed in a sparse manner by merging the ordered views of A and B and adding the values as one progresses.

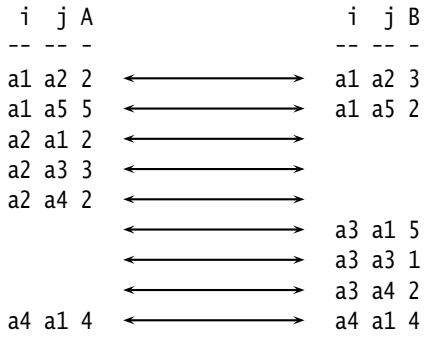


Figure 12.1: Sparse execution of the + operator

Union behavior

In this figure, each arrow represents a computed result. The behavior of the + operator is referred to as *sparse union* behavior: the union of rows from A and B is taken to form the rows of C and it is sparse because we do not need to consider those tuples (i,j) for which A(i,j) and B(i,j) are both 0.0.

Similar operators

Other operators, such as OR, XOR, <, > and <> have a similar behavior. They can also be implemented using the union of rows and performing the appropriate operation.

12.1.2 The * operator: intersection behavior

Second statement

The second statement in the running example is the simple multiplication of the matching elements resulting in parameter D(i,j):

```
D(i,j) := A(i,j) * B(i,j);
```

Matching rows

This statement can be executed in a sparse manner by intersecting the ordered views of A and B and multiplying the corresponding values. Intersection is sufficient because only for those tuples (i,j) for which both A(i,j) and B(i,j) are non-zero, will a non-zero be computed. This is illustrated in the Figure 12.2

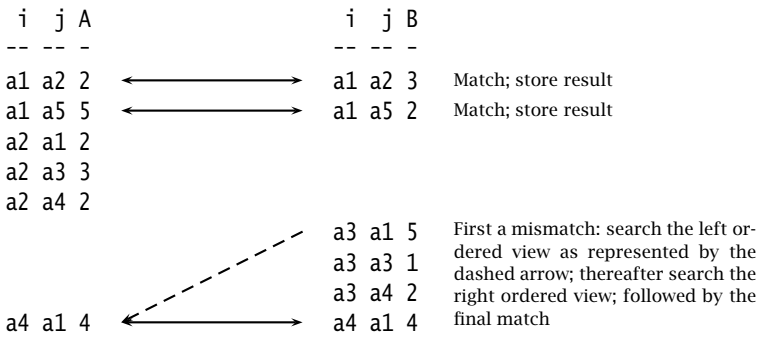


Figure 12.2: Sparse execution of the * operator

Intersection behavior

Note that the ordered views of both A and B are searchable and, thus, finding the matching elements can be efficiently implemented. We call this behavior *sparse intersection* behavior. Because only matching rows need to be considered, sparse intersection operators are much more efficient than sparse union operators.

Similar operators

Other operators, such as the AND and $ operators, exhibit similar behavior. They can also be implemented using the intersection of the rows and performing the appropriate operation.

12.1.3 The = operator: dense behavior

Third statement

The third statement in the running example checks whether corresponding values are equal.

```
E(i,j) := (A(i,j) = B(i,j));
```

Comparing values

This statement is admittedly somewhat artificial. However, such conditions are frequently part of larger expressions and must be considered. The key observation is that the comparison 0.0 = 0.0 evaluates to true. In AIMMS the value 'true' is represented by the numerical value 1.0. Therefore, the result of E(i,j) is:

```
E(i,j) := data table
     a1 a2 a3 a4 a5
  !  -- -- -- -- --
  a1  1     1  1
  a2     1        1
  a3     1        1
  a4  1  1  1  1  1
  a5  1  1  1  1  1   ;
```

Dense behavior

Given that the comparison of two zeros also results in a non-zero, all possible combinations of (i,j) have to be considered. Therefore, this operation exhibits *dense* behavior, i.e. the operation cannot be performed in a sparse manner. Dense operators have the worst possible efficiency.

Similar operators

Other operators, such as /, **, <= and => demonstrate similar behavior. They also need to be implemented by considering all the possibilities and evaluating as one progresses.

Beware!

Increasing the number of indices, or increasing the size of the sets will make the number of rows to be considered in such operations grow rapidly. Large-dimensional dense operations are a potential cause of performance glitches in an application.

12.1.4 Behavior of combined operations

Fourth statement

The fourth statement is a variation of the third statement:

```
EP(i,j) := ( A(i,j) = B(i,j) ) $ A(i,j);
```

Speeding up

Although the operation = remains dense, the entire right hand side of the assignment statement is limited to only those tuples (i,j) for which A(i,j) is non-zero. This is known as a domain condition on the expression. The net effect on the expression is that this condition speeds up efficient behavior by moving from dense to sparse behavior. The result of this fourth assignment is:

```
EP(i,j) := data table
     a1 a2 a3 a4 a5
  !  -- -- -- -- --
  a1
  a2
  a3
  a4  1
  a5                ;
```

Preventing dense behavior

If your model contains a statement that performs badly due to a dense operation, using a domain condition can remedy the problem. Often, it is possible to formulate a domain condition that does not alter the result of the computation, but which does allow AIMMS to execute the statement in a sparse manner.

12.1.5 Summation

Fifth statement

The fifth statement, as detailed below, is a step towards the sixth statement and illustrates a language construct where sparse evaluation is straightforward. This fifth statement is a simple aggregation of the parameter A(i,j) in a parameter AI(i):

```
AI(i) := Sum( j, A(i,j) );
```

This operation is illustrated in Figure 12.3.

```
 i  j A
-- -- -
a1 a2 2 } 7
a1 a5 5 }
a2 a1 2 }
a2 a3 3 } 7
a2 a4 2 }
a4 a1 4 } 4
```

Figure 12.3: Sparse execution of the Sum operator

Running indices and identifier indices match

Each pairing represents a group of values corresponding to a particular value of i. As the elements in a group are adjacent in this ordered view, the result of AI can be computed in a single pass over the ordered view of A. The order of the running indices in the statement is [i,j]. The first running index i is already part of the left hand side of the assignment, and j is added to this list as part of the sum.

Single pass is sufficient

Because the order of the running indices matches the order of the indices in the identifier A(i,j), the results of the sum can be computed in a single pass over the ordered view of A(i,j).

12.1.6 Reordered views

Sixth statement

The sixth statement is a small variation to the fifth statement above. This sixth statement is an aggregation of the parameter A in a parameter AJ(j):

```
AJ(j) := Sum( i, A(i,j) );
```

Non-matching index order

This time, the elements that belong to the same group `j` are not adjacent in the ordered view of `A` as the order of the indices in this statement is `[j,i]` which does not match the order of the indices in `A(i,j)`.

Reordered views

In order to regain adjacency of the elements in the same group, AIMMS maintains other views of the parameter `A` known as *reordered views*. A reordered view of an ordered view is a lexicographic order of the elements such that the order of the indices in the identifier matches the order of the running indices. A reordered view, and the grouping according to this view, are illustrated in Figure 12.4.

```
 j  i A
-- -- -
a1 a2 2  }
a1 a4 4  } 6
a2 a1 2  } 2
a3 a2 3  } 3
a4 a2 2  } 2
a5 a1 5  } 4
```

Figure 12.4: Sparse execution of the reordered `Sum` operator

Single pass is sufficient

Again, each pairing represents a group of values corresponding to a particular value of `j`. As the elements in a group are adjacent in this reordered view, the results of `AJ` can be computed by a single pass over this reordered view of `A`. AIMMS generates and maintains reordered views on an as needs basis. They do, however, take up memory.

12.2 Modifying the sparsity

Questions

Now that we've glanced at the execution engine's inner workings, you may be wondering about the following questions.

- Does sparse execution influence the results of a model?
- Does AIMMS have sparse versions of operators that are dense by nature?

Sparse execution is correct

Sparse execution never changes the results of your model. AIMMS only applies sparse intersection or sparse union when it is applicable. It does not in any way influence the results of your model compared to simply considering all the possible combinations of the running indices, but only the efficiency with which these results are obtained.

Sparsity modifiers

AIMMS does support sparse versions of some dense operators, but this time the sparse versions will in general lead to different results. Adding $ characters to dense operators modify these operators to sparse ones. That is why we call the $ characters added to these operators *sparsity modifiers*.

Left and right operands

Sparsity modifiers may be added to the left-hand side of a dense operator, to the right-hand side, or to both. It causes the operator only to return a non-zero result if the associated operand(s) are non-zero. Such a change to an operator may, however, change its results in a way you may, or may not, want.

A first example: the /$ operator

Let us now consider a few examples where such a modification is applicable. The first example of using a sparsity modifier is in the efficient guarding against division by zero errors. Without the use of sparsity modifiers, we can accomplish this as follows.

```
! Leave A(i,j) zero when C(i,j)+D(i,j) is zero in order to
! avoid division by zero errors.
! This is accomplished by repeating the denominator in the condition.
A(i,j) := ( B(i,j) / (C(i,j)+D(i,j)) ) $ (C(i,j)+D(i,j)) ;
```

In the example, we only divide by C(i)+D(i) if this sum is non-zero. Note that this subexpression is actually computed twice. AIMMS provides a notational convenience in the form of $ sparsity modifiers as follows.

```
! Leave A(i,j) zero when C(i,j)+D(i,j) is zero in order to
! avoid division by zero errors.
! This is accomplished by using the /$ division operator
! which sparsely skips 0.0's.
A(i,j) := B(i,j) /$ (C(i,j)+D(i,j)) ;
```

The /$ operator is defined as the / operator except when the right hand side is 0.0. In that case, the $ sparsity modifier defines it as 0.0. An added advantage is that the sub-expression C(i)+D(i) is only computed once.

The merge operator :=$

A second example is in the merging of new results in a set of existing results. Without the use of a sparsity modifier you can accomplish this as follows.

```
! Only overwrite elements of E(i,j) when the result
! F(i,j) + G(i,j) is non-zero.
! This is accomplished by repeating the RHS of the
! assignment as a domain condition.
E((i,j) | F(i,j)+G(i,j)) := F(i,j)+G(i,j) ;
```

Using the $ sparsity modifier this can be equivalently obtained as follows.

```
! Only overwrite elements of E(i,j) when the result
! F(i,j) + G(i,j) is non-zero.
! This is accomplished by using the $ sparsity
! modifier on the assignment operator:
E(i,j) :=$ F(i,j)+G(i,j) ;
```

Where allowed?

Table 12.1 summarizes the operators to which the $ sparsity modifier can be applied, and whether it can be applied to the left-hand side operand, to the right-hand side operand, or to both.

Operator	Sparsity modifier allowed	
	$ left	$ right
^	yes	yes
*	no	no
/	no	yes
+, -	no	no
=, <>, <, <=, >, >=	yes	yes
:=	yes	yes
+=, -=	yes	no
*=, /=, ^=	yes	yes
$, ONLYIF AND, OR, XOR	no	no

Table 12.1: Sparsity modifiers of binary operators

Modifying iterative operators

In addition to modifying the behavior of binary operators, the $ sparsity modifier can also be applied to iterative operators. The effect in this case is that the iterative operator in the presence of a $ modifier will only be applied to tuples for which the expression yields a non-zero value.

Example: the Min$ operator

The third and final example of the $ sparsity modifier provided here is on the Min operator. Suppose you want to find the smallest non-zero distance between a particular node and other nodes. This can be modeled as follows:

```
! Find the smallest non-zero distance:
MinimalDistance(i) := Min(j | Distance(i,j), Distance(i,j));
```

The 'non-zero' restriction is taken care of by repeating the argument of the Min operator in its domain condition. By using the $ sparsity modifier we can shorten the above as follows:

```
! Find the smallest non-zero distance:
MinimalDistance(i) := Min$(j, Distance(i,j));
```

Where allowed?

Table 12.2 summarizes the iterative operators to which the $ sparsity modifier can be applied.

Iterative operator	Sparsity modifier allowed
	$ added
Sort, NBest	yes
Intersection,	yes
First, Last, Nth	no
ArgMin, ArgMax	yes
Sum, Union	no
Prod	yes
Min, Max	yes
Statistical operators (see also page 82)	yes
ForAll	no
Other logical operators (see also page 90)	no

Table 12.2: Sparsity modifiers of iterative operators

Usage of sparsity modifiers

To conclude, we can say that the $ sparsity modifier is notationally a convenience which you may or may not like. In the end it is up to you whether you use it or not. You decide this by weighing its advantage and disadvantages. Our view on this is discussed briefly below.

Advantages

Using sparsity modifiers has the following advantages.

- It enables a more compact notation. In the examples above, the domain condition is replaced by a strategically placed $ sparsity modifier thereby reducing the overall expression. Many models have with multiple line subexpressions and with these the reduction is not insignificant.
- It is more efficient. There are usually abundant zeros in a model. You want them ignored so that the corresponding entries do not appear in the results. In addition, you want them to be ignored as quickly as possible: so as not to waste any computation time on them.

Disadvantages

As with any new notation it takes time to get used to it. This holds both for you as a modeler and also for the people you want to communicate your model to. In order to alleviate this disadvantage you may want to add a few brief comments on the modified operators you use such as "`:=$ operator used here to merge the result into the existing data`".

12.3 Overview of operator efficiency

Operator efficiency

In this section you will find an overview of the efficiency of all unary, binary and iterative operators in AIMMS.

Unary operators and functions

The unary operators and functions presented in Table 12.3 are divided in two groups: sparse and dense.

- *sparse*: Here, when the argument is 0.0, the result is 0.0. The result needs to be computed only for those tuples for which the argument has a non-zero value.
- *dense*: Here, when the argument is 0.0, the result is not equal to 0.0. The results of all possible tuples need to be computed.

sparse		dense	
-	Sinh	NOT	Cos
Sin	Tanh	Cos	Cosh
Tan	ArcSin	Exp	ArcCos
Round	ArcTan	Log	ArcCosh
Floor	ArcSinh	Log10	Factorial
Ceil	ArcTanh		
Trunc	Sqr		
Sqrt			

Table 12.3: Sparsen and dense unary operators and functions

Binary operators

The binary operators presented in Table 12.4 can be divided in three groups:

- *intersection sparse*: Here, when either of the arguments is 0.0, the result is 0.0. The result of only those tuples need to be computed where both arguments are not equal to 0.0. This corresponds to taking the intersection of the set of tuples for which the arguments are defined.
- *union sparse*: Here, when both arguments are 0.0, the result is 0.0. The result of only those tuples need to be computed where at least one of the arguments is not equal to 0.0. This corresponds to taking the union of the set of tuples for which the arguments are defined.
- *dense*: Here, when both arguments are 0.0, the result is not equal to 0.0. In this case, the expression needs to be evaluated for all possible combinations of values of the indices, unless these combinations are limited by a sparse operator elsewhere in the same expression. This corresponds to taking the Cartesian product of the ranges of the indices.

Iterative operators

The iterative operators presented in Table 12.5 are divided in three groups as follows:

- *sparse* A value 0.0 of an argument does not influence the result and can safely be ignored. The iterative operator only considers existing entries of its argument.

intersection	union	dense
*	+	^
$	-	/
ONLYIF	<>	=
AND	<	<=
	>	>=
	OR	Permutation
	XOR	Combination

Table 12.4: Sparseness of binary operators

- *almost sparse* A second 0.0 in the argument does not influence the result. The execution starts in a dense meaning that the iterative operator considers all possible tuples. However, after a first 0.0 has been encountered, execution continues in a sparse manner.
- *dense* A value 0.0 in the argument influences the result. The iterative operator considers all possible combinations.

sparse	almost sparse	dense	
Sum	Max	Mean	SampleDeviation
Prod	Min	GeometricMean	PopulationDeviation
Exists	ArgMax	HarmonicMean	Skewness
Forall	ArgMin	RootMeanSquare	Kurtosis
Count		Median	RankCorrelation

Table 12.5: Sparseness of iterative operators

Chapter 13

Execution Efficiency Cookbook

This chapter

Typically, when you start running your model with realistic, large-scale data sets, execution performance becomes an important issue. In this chapter, we discuss various techniques that you can use to improve the execution efficiency of your model.

Dividing the time spent

The running time of AIMMS applications can be divided in the time spent by AIMMS itself and the time spent by the solution algorithms (i.e. solvers) used by AIMMS.

Time spent by solvers

The time used by the solvers mostly depends, apart from the quality of the solver, on the specific formulation of the mathematical program to be solved. Finding a formulation that can be efficiently solved is often a challenging task and is beyond the scope of this chapter. For a detailed discussion, you are referred to the extensive literature that exists on this subject.

Time spent by AIMMS

AIMMS itself typically spends most of its time on the execution of assignment statements and the generation of constraints. This time depends on several factors. A few of these factors are:

- the size of the sets and the data set size used in your model,
- the efficiency of the AIMMS execution engine, and
- the language constructs used to formulate the execution statements and constraints.

Understanding AIMMS *execution*

At AIMMS we are committed to continuously improving the efficiency of the AIMMS execution engine and the AIMMS matrix generator. The efficiency of your application, however, does not only depend on the efficiency of AIMMS, but also on the specific formulation of your model and the language constructs that you have used. A global understanding of the AIMMS execution engine, as presented in Chapter 12, may provide a good background on which to start reconsidering particular formulations that lead to bottlenecks in execution performance in your application.

Analysis tools

In addition, AIMMS provides you with two tools for analyzing execution bottlenecks, namely the **Identifier Cardinalities** and **Profiler** tools. The use of both tools is described in Chapter 8 of the AIMMS User's Guide.

Analyzing cardinalities

The **Identifier Cardinalities** tool can help you to discover identifiers with a large number of elements. Such identifiers, when used in statements and constraints, may lead to efficiency bottlenecks throughout your model. Whenever you are able to reduce the number of elements associated with such identifiers, by leaving out irrelevant elements, the execution efficiency of your model will improve at several places. Naturally, such reductions are not possible when all the elements are relevant to the computation of the solution. In Section 13.1, we discuss two frequently observed and effective approaches to reducing the number of elements in both one-dimensional sets and multidimensional identifiers.

Analyzing statements

With the AIMMS **Profiler** tool you can identify the individual statements and constraints on which the AIMMS execution engine spends most of its time. Even if the inefficiencies are not the result of superfluous identifier cardinalities, it may still be possible to review and rewrite such statements and constraints in order to improve the execution efficiency of your application. In Section 13.2 we discuss potential bottlenecks and alternative formulations for particular statements and constraints.

Simple precautions

Before you begin tuning your application, you may want to set aside a copy of the application and inputs with known results. You can then set up a script that executes each of these tests using the AIMMS command line option `--run-only` (see also Chapter 18 of AIMMS The User's guide). In addition, you may wish to regularly commit your sources to a version control system in order to track the changes you make over time.

13.1 Reducing the number of elements

Application phases

In general, one can divide an application in three phases:

1. reading input data, often referred to as reading and preprocessing,
2. processing data, often referred to as the core model, and
3. writing output results, often referred to as reporting.

Interactive applications add the on/off switching of various application features, the setting of tuning parameters, the consideration of various scenarios, the output to screen, and so on. This does not change the basic concept, however. It only means that the inputs come from various sources and the outputs go to various destinations. An important observation is that, usually, most of the computation time is spent in the core model as this involves:

- the execution of assignments,

- the evaluation of definitions,
- the generation of constraints, and
- the execution of one or more SOLVE statements.

Reducing the core model

Obviously, the fewer data we have in the core model, the sooner we're finished. Often, a considerable percentage of the data read in during the data input phase is irrelevant to the final result. We could, therefore, consider spending more time in the data input phase and try to remove such irrelevant data with the primary objective of reducing the amount of data used in the core model. Experience shows that this effort is usually, but not always, worthwhile.

Reducing the number of elements

In this section, two complementary methods of reducing the model size are considered, namely reducing the number of elements in

- one-dimensional sets, and
- multidimensional identifiers.

13.1.1 Size reduction of one-dimensional sets

Two approaches

If, after the data input phase, a one-dimensional set contains a large number of elements that are irrelevant to the core model, there are two possible approaches to removing them from computations in the core model. These are:

- adding a condition to all identifiers indexed over that set, or
- introducing a subset of active elements, and using an index to that active subset.

These two approaches are illustrated below.

Tanks example

As a running example, consider a collection of tanks. Let us introduce a few identifiers related to tanks:

```
Set Periods {
    Index        :  t;
}
Set Tanks {
    Index        :  Tnks;
}
Set BrokenTanks {
    SubsetOf     :  Tanks;
}
Parameter StrategicReserve {
    IndexDomain  :  Tnks;
}


Parameter SizeOfTank {
    IndexDomain  :  Tnks;
}
Parameter TankIsRelevant {
```

```
    IndexDomain  :  Tnks;
    Range        :  binary;
    Definition   :  {
        1 $ [ ( not Tnks in BrokenTanks                      ) AND
              ( SizeOfTank( Tnks ) > StrategicReserve( Tnks ) )     ]
    }
}
Variable TankLevel {
    IndexDomain  :  (t,Tnks) | TanksIsRelevant( Tnks );
}
Constraint TankLimit {
    IndexDomain  :  (t,Tnks) | TanksIsRelevant( Tnks );
    Definition   :  TankLevel( t,Tnks ) <= SizeOfTank( Tnks );
}
```

The example above illustrates the first approach, in which the restriction on the tanks is embodied by the parameter `TankIsRelevant`.

Introducing active subsets

To illustrate the second approach, we change the above model section by introducing the *active subset* `ActiveTanks` and modifying the declaration of the variable `TankLevel` and the constraint `TankLimit` as presented below.

```
Set ActiveTanks {
    SubsetOf     :  Tanks;
    Index        :  tnk, tnk2;
    Definition   :  { Tnks | TankIsRelevant(Tnks) };
}
Variable TankLevel {
    IndexDomain  :  (t,tnk);
}
Constraint TankLimit {
    IndexDomain  :  (t,tnk);
    Definition   :  TankLevel( t,tnk ) <= SizeOfTank( tnk );
}
```

The core model still consists of the variable `TankLevel` and the constraint `TankLimit` but their index domain has been changed. These identifiers are now declared over active tanks only. Because of this change in the index domain, the parameter `TankIsRelevant` is no longer needed in their index domain condition.

Speedup by active subsets

One may argue that nothing is gained because the selection through TankIsRelevant is now replaced by the index tnk of the active subset ActiveTanks. However, the AIMMS execution engine has been tuned to select relevant elements of parameters and variables through indices in subsets. The selection via a condition such as TankIsRelevant(Tnks) will force AIMMS to retrieve the values for:

- the parameter or variable at hand,
- the parameter TanksIsRelevant, and then
- combine these values using the 'such that' operator |.

Both approaches produce identical results and limit the core model execution to relevant elements only. The first approach using the TankIsRelevant condition takes more execution time than the second approach using an index in the active subset ActiveTanks because this latter approach selects the relevant elements more directly.

Multiple active subsets

Intuitively you might expect the improvement to be minor because probably only a few tanks, if any, are removed from the collection of all tanks. However, for other indices of the model the gain may be significant. More significant gains may be observed, for example, when

- you study a few periods from a large model calendar,
- you study a few scenarios from a large database of scenarios,
- you study a rather limited region,
- there are only a few crudes available from a large collection of available crudes, or
- there are only a few products ordered from a large catalog.

A large dimensional identifier, indexed over multiple active subsets, will have the effect.

Starting with a core model

What if your model does not limit the number of elements in one-dimensional sets at all? Following the active subset approach, as illustrated above, you will have to modify the core model wherever you use the root set or an index in the root set. In such a situation, you can also implement "active subsets" by introducing a superset of the root set, and letting the original root set take on the role of an active subset.

Example

We continue the running example by presenting a core model version of it.

```
Set Periods {
    Index        :  t;
}
Set Tanks {
    Index        :  tnk;
}
```

```
Parameter SizeOfTank {
    IndexDomain  :  tnk;
}
Variable TankLevel {
    IndexDomain  :  (t,tnk);
}
Constraint TankLimit {
    IndexDomain  :  (t,tnk);
    Definition   :  TankLevel( t,tnk ) <= SizeOfTank( tnk );
}
```

In implementing the active subset approach, we introduce a new superset AllTanks and redefine the original set Tanks as an active subset of the superset AllTanks as follows.

```
Set AllTanks {
    Index        :  Tnks;
}
Set BrokenTanks {
    SubsetOf     :  AllTanks;
}
Parameter StrategicReserve {
    IndexDomain  :  Tnks;
}
Parameter TankIsRelevant {
    IndexDomain  :  Tnks;
    Range        :  binary;
    Definition   :  {
        1 $ [ ( not Tnks in BrokenTanks                      ) AND
              ( SizeOfTank( Tnks ) > StrategicReserve( Tnks ) )     ]
    }
}
Set Tanks {
    SubsetOf     :  AllTanks;
    Index        :  tnk;
    Definition   :  { Tnks | TankIsRelevant(Tnks) };
}
Parameter SizeOfTank {
    IndexDomain  :  Tnks;
    Comment      :  Now Wrt AllTanks instead of Tanks;
}
```

Note that the variable and constraint declarations in the core model above have not been altered, but their size has been reduced by the size reduction in the set Tanks.

13.1.2 Size reduction of multidimensional identifiers

Limiting multi-dimensional identifiers

Having illustrated limiting the number of elements in one-dimensional sets, we want to consider limiting the number of elements in multidimensional parameters, variables, and constraints. The AIMMS language facilitates this through the IndexDomain attribute.

Index domain conditions

Domain conditions can be specified in the IndexDomain attribute of multidimensional parameters, variables, and constraints. Whenever such an identifier is assigned, generated, or referenced in an expression, AIMMS will automatically add the domain condition so keeping your assignments and constraints more concise and efficient.

Continued example

We illustrate this by extending the above example as follows.

```
Variable Flow {
    IndexDomain  : (t,tnk,tnk2);
}
Constraint TankLevelBalance {
    IndexDomain  : (t,tnk) | t <> first(Periods);
    Definition   : {
        TankLevel(t-1,tnk)              ! Level of previous period
                   - Sum( tnk2, Flow(t,tnk,tnk2) ) ! Flow out of the tank
                   + Sum( tnk2, Flow(t,tnk2,tnk) ) ! Flow in to the tank
                   = TankLevel(t,tnk)              ! Current level
    }
    Comment      : {
           "Level at end of previous period
            minus outflow
            plus  inflow is
            level at end of current period"
    }
}
```

Note that, using this formulation, AIMMS generates matrix columns for every possible pair of tanks, whereas in practice only a small selection can have an actual flow. If this selection of possible connections between tanks is represented by a relation TankConnections, the constraint TankLevelBalance could be written more efficiently as:

```
Set TankConnections {
    SubsetOf     : (AllTanks, AllTanks);
}
Variable Flow {
    IndexDomain  : (t,tnk,tnk2);
}
Constraint TankLevelBalance {
    IndexDomain  : (t,tnk) | t <> first(Periods);
    Definition   : {
        TankLevel(t-1,tnk)
            - Sum( tnk2 | (tnk,tnk2) in TankConnections, Flow(t,tnk,tnk2) )
            + Sum( tnk2 | (tnk2,tnk) in TankConnections, Flow(t,tnk2,tnk) )
            = TankLevel(t,tnk)
    }
}
```

Note the repetition of the condition in the above formulation. This is because the condition is actually a restriction on the Flow variable, and should therefor be a part of its declaration. This leads to a much more concise formulation, as presented below.

```
Variable Flow {
    IndexDomain  : (t,tnk,tnk2) | (tnk,tnk2) in TankConnections;
}
Constraint TankLevelBalance {
    IndexDomain  : (t,tnk) | t <> first(Periods);
    Definition   : {
        TankLevel(t-1,tnk)
            - Sum( tnk2, Flow(t,tnk,tnk2) )
            + Sum( tnk2, Flow(t,tnk2,tnk) )
            = TankLevel(t,tnk)
    }
}
```

Using binary parameters

A frequently observed alternative to using relations is the use of binary parameters. The above example could then be written as follows:

```
Parameter TankIsConnected {
    IndexDomain  : (tnk,tnk2);
    Range        : {0, 1};
}
Variable Flow {
    IndexDomain  : (t,tnk,tnk2) | TankIsConnected(tnk,tnk2);
}
```

The outflow term of `TankLevelBalance` will then be generated as if it were written:

```
    Sum( tnk2, Flow(t,tnk,tnk2) $ TankIsConnected(tnk,tnk2) )
```

The notation using binary parameters is equivalent to that with relations. Which option you use is only a matter of taste and style.

Why use index domain conditions?

We would encourage you to employ index domain conditions, as using them has the following advantages:

1. Index domain conditions speed up the execution because:
   - They exclude irrelevant elements in assignments to parameters with an index domain condition,
   - Having index domain conditions on variables effectively makes the referencing of such variables sparse, as only relevant columns are generated, and
   - Index domain conditions on a constraint avoid the generation of irrelevant rows of that constraint.
2. Index domain conditions permits concise formulations. As illustrated above, you do not need to include the domain condition of the `Flow` variables while constructing the `TankLevelBalance` constraint. Moreover, you do not need to worry that you mightforget such a condition at a particular place in the model.
3. Whenever you determine a more restrictive condition on an identifier `A`, you only need to change your model at one place, namely in the index domain condition of that identifier `A`. You don't need to go through the entire model changing every reference to the identifier `A`.

Tight conditions

To make index domain conditions as effective as possible, they should remove all, or almost all, irrelevant combinations. Constructing such "tight" index domain conditions, can be far from straightforward. However, the time spent on constructing tight index domain conditions often pays off with a significant reduction in the total execution time of your model.

13.2 Analyzing and tuning statements

Analyzing and tuning statements

As illustrated in the previous section, carefully reviewing the number of elements in active subsets and the index domain conditions may lead to significant reductions in execution time. Additional reductions can be obtained by analyzing and rewriting specific time-consuming statements and constraints. In this section we will discuss a procedure which you can follow to identify and resolve potential inefficiencies in your model.

Suggested approach

You can use the AIMMS profiler to identify computational bottlenecks in your model. If you have found a particular bottleneck, you may want to use the checklist below to quickly find relevant information for the problem at hand. For each question that you answer with a yes you may want to follow the suggested option.

- Is the bottleneck a repeated expression where the combined execution of all instances takes up a lot of time? If so, you can either
  - manually replace the expression by a new parameter containing the repeated expression as a definition. Do not forget to check the `NoSave` property if you do not want that newly defined parameter to be stored in cases.
  - or let AIMMS do it for you, by setting the option `subst low dim expr class` to an appropriate value for your application. See also the help associated with that option.

  For a worked example, see also Subsection 13.2.4
- Is the bottleneck due to debugging/obsolete code? If so, delete it, move it to the `Comment` attribute, or enclose the time-consuming debugging code in something like an `IF ( DebugMode ) THEN` and `ENDIF` pair.
- Are you using dense operators such `/`, `=`, `^`, or dense functions such as `Log`, `Exp`, `Cos` in which a zero argument has a non-zero result? An overview of the efficiency of such functions and operators can be found in Section 12.3. Could you add index domain conditions to make the execution of the time-consuming expressions more sparse, without changing the final result?
- Is the bottleneck part of a `FOR` statement? If so, is that `FOR` statement really necessary? For a detailed discussion about the need for and alternatives to `FOR` statements, see Section 13.2.1.
- Is the bottleneck the condition of the `FOR` statement that takes up most of the time? This is shown in the profiler by a large net time for the `FOR`

statement. Section 13.2.2 discusses why the conditions of FOR statements may absorb a lot of computational time and discusses alternatives.

- Does the body of a FOR, WHILE, or REPEAT statement contain a SOLVE statement, and is AIMMS spending a lot of time regenerating constraints (as shown in the profiling times of the constraints)? If so, consider modifying the generated mathematical programs directly using the GMP library as discussed in Chapter 16.
- Does your model contain a defined parameter over an index, say t, and do you use this parameter inside a FOR loop that runs over that same index t? Inefficient use of this construct is indicated by the AIMMS profiler through a high hitcount for that defined parameter. See Section 13.2.3 for an example and an alternative formulation.
- Is the bottleneck an expression with several running indices? Contains this expression non-trivial sub-expressions with fewer running indices? If the answer is yes, consult Section 13.2.4 for a detailed analysis of two examples.
- Does the expression involve a parameter or a variable that is bound with a non-zero default? Section 13.2.5 discusses the possible adverse timing effects of using non-zero defaults in expressions, and how to overcome these.
- Would you expect a time-consuming assignment to take less time given the sparsity of the identifiers involved? This may be one of those rare occasions in which the specific order of running indices has an effect on the execution speed. Although tackling this type of bottleneck may be very challenging, Section 13.2.6 hopefully offers sufficient clues through an illustrative example.
- Are you using ordered sets? Reordering the elements in a set can slow execution significantly as detailed in Section 13.2.7.

13.2.1 Consider the use of FOR statements

Why avoid the FOR statement?

The AIMMS execution system is designed for efficient bulk execution of assignment statements, plus set and parameter definitions and constraints. A consequence of this design choice is that computation time is spent, just before the execution of such an executable object, analyzing and initializing that object. This is usually worthwhile except when only one element is computed at a time. Consider the following two fragments of AIMMS code that have the same final result. The first fragment uses a FOR statement:

```
for ( (i,j) | B(i,j) ) do ! Only when B(i,j) exists we want to
    A(i,j) := B(i,j);     ! overwrite A(i,j) with it.
endfor ;
```

The second fragment avoids the FOR statement:

```
A((i,j) | B(i,j)) := B(i,j); ! Overwrite A(i,j) only when B(i,j) exists
```

In the first fragment, the initialization and analysis is performed for every iteration of the FOR loop. In the second fragment the initialization and analysis is performed only once. Using the $ sparsity modifier on the assignment operator := (see also Section 12.2), the statement can be formulated even more compactly and efficiently as:

```
A(i,j) :=$ B(i,j); ! Merge B(i,j) in A(i,j)
```

In the above example, the FOR statement is used only to restrict the domain of execution of a single assignment. While using the FOR statement in this manner may seem normal to programmers, the execution engine of AIMMS can deal with conditions on assignment statements much more efficiently. As such, the use of the FOR statement is superfluous and time consuming.

When not *to remove the FOR statement*

Now that the FOR statement has been made to look inefficient, you are probably wondering why has it been introduced in the AIMMS language in the first place? Well, simply because sometimes it is needed. And it is only inefficient if used unnecessarilly. So when is the FOR statement applicable? Two typical examples are:

- generating a text report file, and
- in algorithmic code inside the core model.

We will discuss these examples in the next two paragraphs.

Generating text reports

The AIMMS DISPLAY statement is a high level command that outputs an identifier in tabular, list, or composite list format with a limited amount of control. In addition, the output of the DISPLAY statement can always be read back by AIMMS, and, to enable that requirement, the name of the identifier is always included in the output. Thus, the AIMMS DISPLAY statement usually fails to meet the specific formatting requirements of your application domain, and you end up needing control over the position of the output on an element-by-element basis. This requires the use of FOR statements. However, depending on the purpose of your text report file, there might be very good alternatives available:

- When this reporting is for printing purposes only, you may want to consider the AIMMS print pages as explained in AIMMS The User's Guide Chapter 14. These print pages look far better than text reports.
- When the report file is for communication with other programs, you may want to consider whether communication using relational databases (see Chapter 27), or through XML (see Chapter 30) form better alternatives. For communication with EXCEL or OpenOffice Calc, a library of dedicated functions is built in AIMMS (see Chapter 29).

Algorithmic code inside the core model

A FOR statement is needed whenever your model contains two statements where:

- the computation of the last statement depends on the computation of the first statement, and
- the computation of the first statement depends on the results of the last statement obtained during a previous iteration.

Iterating unneccesarily

FOR statements may be especially inefficient, if the condition of a FOR statement allows elements for which none of the statements inside the FOR loop modify the data in your model or generate output. This is illustrated in the following example.

Transposing a distance matrix

Consider a distance matrix, D(i,j), with only a few entries per row in its lower left half containing the distances to near neighbors. You also want it to contain the reverse distances. One, inefficient, but valid, way to formulate that in AIMMS is as follows:

```
for ( (i,j) | i > j ) do ! The condition 'i > j' ensures we only
    D(i,j) := D(j,i) ;   ! write to the upper right of D.
endfor ;
```

Why inefficient?

There are two reasons why the above is inefficient:

- Although there is a condition on the FOR loop, this condition permits many combinations of (i,j) that do not invoke execution as D(i,j) was sparse to begin with. A tempting improvement would be to add D(j,i) to the condition on the FOR loop. However, this will lead to other problems, however, as will be explained in the next section.
- As explained in Section 12.1.6, AIMMS maintains reordered views. For each non-zero value computed and assigned to the identifier D(i,j), AIMMS will need to adapt the reordered view for D(j,i), and re-initialize searching in that reordered view.

Suggested modification

In the example at hand we can move the condition on the FOR loop to the assignment itself and simply remove the FOR statement altogether (but not its contents). The example then reads:

```
D((i,j) | i > j) := D(j,i) ; ! The condition 'i > j' ensures we only
                             ! write to the upper right of D.
```

Using application domain knowledge

We can improve the assignment further by noting that we are actually merging the transposed lower half in the identifier itself, and that there is no conflict in the elements. This can be achieved by a $ sparsity modifier on the assignment operator. The $ sparsity modifier and the opportunity it offers are introduced in Section 12.2. The example can then be written as:

```
D(i,j) :=$ D(j,i); ! Merge the transpose of the lower half in the identifier itself.
```

13.2.2 Ordered sets and the condition of a FOR statement

Modifying the FOR condition

The condition placed on a FOR statement is like any other expression evaluated one element at a time. However, during that evaluation, the identifiers referenced in the condition may have been modified by the statements inside the FOR loop. In general, this is not a problem, except when the range of the running index of the FOR statement is an *ordered* set. In that situation, the evaluation of the condition itself becomes time consuming as the tuples satisfying the condition have to be repeatedly computed and sorted, as illustrated below.

Continued example

Let us again consider the example of the previous section with the parameter D now added to the FOR loop condition, and the set S ordered lexicographically. As an efficient formulation has already been presented in the previous section, it looks somewhat artificial, but similar structures may appear in real-life models.

```
Set S {
    Index      : i,j;
    OrderBy    : i ! lexicographic ordering.;
    Body       : {
        for ( (i,j) | ( i > j ) AND D(j,i) ) do ! Only execute the statements in the
            D(i,j) := D(j,i) ;                  ! loop when this is essential.
        endfor
    }
}
```

What does AIMMS *do in this example?*

First note that the FOR statement respects the ordering of the set S. Because of this ordering, AIMMS will first evaluate the entire collection of tuples satisfying the condition (i > j) AND D(j,i), and subsequently order this collection according to the ordering of the set S. Next, the body of the FOR statement is executed for every tuple in the ordered tuple collection. However, when an identifier, such as D in this example, is modified inside the body of the FOR loop AIMMS will need to recompute the ordered tuple collection, and continue where it left off. This not only sounds time consuming, it is.

FOR as a bottleneck

If the following three conditions are met, the condition of a FOR statement becomes time consuming:

- the indices of a FOR statement have a specified element order,
- the condition of the FOR statement is changed by the statements inside the loop, and
- the product of the cardinality of the sets associated with the running indices of the FOR statement is very large.

if these three conditions are met, AIMMS will issue a warning when the number of re-evaluations reaches a certain threshold.

Improving efficiency

There are several ways to improve the efficiency of inefficient FOR statements. To understand this, it is necessary to explain a little more about the execution strategies available to AIMMS when evaluating FOR statements, as each strategy has its own merits and drawbacks. Therefore, consider the FOR statement:

```
for ( (i,j,k) | Expression(i,j,k) ) do
   ! statements ...
endfor;
```

where i, j and k are indices of some sets, each with a specified ordering, and Expression(i,j,k) is some expression over the indices i, j and k.

The sparse strategy

The first strategy, called the *sparse* strategy, fully evaluates Expression(i,j,k), and stores the result in temporary storage before executing the FOR statement. Subsequently, for each tuple (i,j,k) for which a non-zero value is stored, the statements within the FOR loop are executed. If an identifier is modified during the execution of these statements, then the condition Expression(i,j,k) has to be fully re-evaluated.

The dense strategy

The second strategy, called the *dense* strategy, evaluates Expression(i,j,k) for all possible combinations of indices (i,j,k). As soon as a non-zero result is found the statements are executed. Re-evaluation is avoided, but at the price of considering every (i,j,k) combination.

The unordered strategy

The third strategy, called the *unordered* strategy, uses the normal sparse execution engine of AIMMS but ignores the specified order of the indices. This may, however, give different results, especially when the FOR loop contains one or more DISPLAY/PUT statements or uses lag and lead operators in conjunction with one or more of the ordered indices.

Selecting a strategy

By prefixing the FOR statement with one of the keywords SPARSE, ORDERED, or UNORDERED (as explained in Section 8.3.4), you can force AIMMS to adopt a particular strategy. If you do not explicitly specify a strategy, AIMMS uses the sparse strategy by default, and only issues a warning if an identifier referenced inside the FOR loop is modified and the second evaluation of Expression(i,j,k) gives a non-empty result.

Improving efficiency

Given the above, you have the following options for improving the efficiency of the FOR statement.

- Rewrite the FOR statement such that the condition does not change during each iteration.
- Prefix the FOR statement with the keyword UNORDERED such that the unordered strategy will be set. You can safely choose this strategy if the element order is not relevant for the FOR statement. In all other cases, the semantics of the FOR statement will be changed.

- Prefix the FOR statement with the keyword ORDERED such that the dense strategy is selected. You can safely choose this strategy if the condition on the running indices evaluates to true for a significant number of all possible combinations of the tuples (i,j,k).
- Prefix the FOR statement with the keyword SPARSE to adopt the sparse strategy. However, all warnings will be suppressed relating to the condition on the running indices needing to be evaluated multiple times. You can choose this strategy if the condition needs to be re-evaluated in only a few iterations.

13.2.3 Combining definitions and FOR loops

Dependency is symbolic

As explained in Section 7.1, the dependency structure between set and parameter definitions is based only on symbol references. AIMMS' evaluation scheme recomputes a defined parameter *in its entirety* even if only a single element in its inputs has changed. This negatively affects performance when such a defined parameter is used inside a FOR loop and its input is changed inside that same FOR loop.

A simulation example

A typical example occurs when using definitions in simulations over time. In simulations, computations are often performed period by period, referring back to data from previous period(s). The relation used to computate the stock of a particular product p in period t can easily be expressed by the following definition and then used inside the body of a procedure.

```
Parameter ProductStock {
    IndexDomain  :  (p,t);
    Definition   :  ProductStock(p,t-1) + Production(p,t) - Sales(p,t);
}
Procedure ComputeProduction {
    Body         : {
        for ( t ) do
            ! Compute Production(p,t) partly based on the stock for period (t-1)
            Production(p,t) := Max( ProductionCapacity(p),
                                    MaxStock(p) - ProductStock(p,t-1) + Sales(p,t) );
        endfor ;
    }
}
```

During every iteration, the production in period t is computed on the basis of the stock in the previous period and the maximum production capacity. However, because of the dependency of ProductStock with respect to Production, AIMMS will re-evaluate the definition of ProductStock in its entirety for *each* period before executing the assignment for the next period. Although the FOR loop is not really necessary here, it is used for illustrative purposes.

Improved formulation

In this example, execution times can be reduced by moving the definition of ProductStock to an explicit assignment in the FOR loop.

```
Parameter ProductStock {
    IndexDomain  :  (p,t);
    ! Definition attribute is empty.
}
Procedure ComputeProduction {
    Body         : {
        for ( t ) do
            ! Compute Production(p,t) partly based on the stock for period (t-1)
            Production(p,t) := Max( ProductionCapacity(p),
                                    MaxStock(p) - ProductStock(p,t-1) + Sales(p,t) );

            ! Then compute stocks for current period t
            ProductStock(p,t) := ProductStock(p,t-1) + Production(p,t) - Sales(p,t);
        endfor ;
    }
}
```

In this formulation, only one slice of the ProductStock parameter is computed per period. A drawback of this formulation is that it will have to be restated at various places in your model if the inputs of the definition are assigned at several places in your model.

Use of macros

As an alternative, you might consider the use of a Macro (see also Section 6.4) to localize the defining expression of ProductStock at a single node in the model tree. The disadvantage of macros is that they cannot be used in DISPLAY statements, or saved to cases.

When to avoid definitions

As illustrated above, it is best to avoid definitions, if, within a FOR loop, you only need a slice of that definition to modify the inputs for another slice of that same definition. AIMMS is not designed to recognize this situation and will repeatly evaluate the entire definition. The AIMMS profiler will expose such definitions by their high hitcount.

13.2.4 Identifying lower-dimensional subexpressions

Lower-dimensional subexpressions

Repeatedly performing the same computation is obviously a waste of time. In this section, we will discuss a special, but not uncommon, instance of such behavior, namely lower-dimensional sub-expressions. A lengthy expression, that runs over several indices, can have distinct subexpressions that depend on fewer indices. Let us illustrate this with two examples, the first being an artificial example to explain the principle, and the second a larger example that has actually been encountered in practice and permits the discussion of related issues.

Artificial example

Consider the following artificial example:

```
F(i,k) := G(i,k) * Sum[j | A(i,j) = B(i,j), H(j)] ;
```

For every value of i, the sub-expression Sum[j | A(i,j) = B(i,j), H(j)] results in the same value for each k. Currently, the AIMMS execution engine will repeatedly compute this value. It is more efficient to rewrite the example as follows.

```
FP(i) := Sum[j | A(i,j) = B(i,j), H(j)] ;
F(i,k) := G(i,k) * FP(i) ;
```

The principle of introducing an identifier for a specific sub-expression often leads to dramatic performance improvements, as illustrated in the following real-life example.

A complicated assignment

Consider the following 4-dimensional assignment involving region-terminal-terminal-region transports. Here, sr and dr (source region and destination region) are indices in a set of Regions with m elements and st and dt (source terminal and destination terminal) are indices in a set of Terminals with n elements.

```
Transport( (sr,st,dt,dr) |          TRDistance(sr,st) <= MaxTRDistance(st) AND
                                    TRDistance(dr,dt) <= MaxTRDistance(dt) AND
   sr <> dr AND MinTransDistance <= RRDistance(sr,dr) <= MaxTransDistance  AND
   st <> dt AND MinTransDistance <= TTDistance(st,dt) <= MaxTransDistance
         ) := Demand(sr,dr);
```

The domain condition states that region-terminal-terminal-region transport should only be assigned if the various distances between regions and/or terminals satisfy the given bounds.

Efficiency analysis

The <= operator is dense and be evaluated for all possible values of the indices. The subexpression TRDistance(sr,st) <= MaxTRDistance(st), for example, will be evaluated for every possible value of dr and dt, even though it only depends on sr and st. In other words, we're computing the same thing over and over again.

Effect of the AND operator

There are multiple AND operators in this example. The AND operator is sparse, and oten, sparse operators make execution quick. However, they fail to do just that in this particular example. Bear with us. Although the AND operator is a sparse binary operator, its effectiveness depends on how effectively the intersection is taken. What are we taking the intersection of? If we consider a particular argument of the AND operators: TRDistance(sr,st) <= MaxTRDistance(st), as the operator <= is dense and this argument will be computed for all tuples {(sr,st,dt,dr)} even though the results will be mostly 0.0's. The domain of evaluation for this argument is thus the full Cartesian product of four sets. The evaluation domain of the other arguments of the AND operators will be the

same. So, in this example, we are repeatedly taking the intersection of a Cartesian product with itself, resulting in that same Cartesian product. Thus, the AND operator will be evaluated for all tuples in {(sr,st,dt,dr)} even though this operator is sparse.

Example reformulation

In the formulation below, we've named the following sub-expressions.

```
ConnectableRegionalTerminal( (sr,st) | TRDistance(sr,st) <= MaxTRDistance(st) ) := 1;

ConnectableRegions( (sr,dr) | sr <> dr AND
        MinTransDistance <= RRDistance(sr,dr) <= MaxTransDistance )  := 1;

ConnectableTerminals( (st,dt) | st <> dt AND
        MinTransDistance <= TTDistance(st,dt) <= MaxTransDistance )  := 1;
```

In each of these three assignments, each condition depends fully on the running indices and thus its evaluation is not unnecessarily repeated. By substituting the three newly introduced identifiers in the condition the original assignment becomes:

```
Transport( (sr,st,dt,dr) |
           ConnectableRegionalTerminal(sr,st)     AND
           ConnectableRegionalTerminal(dr,dt)     AND
           ConnectableRegions(sr,dr)              AND
           ConnectableTerminals(st,dt) )
         := Demand(sr,dr);
```

The newly created identifiers are all sparse, and the sparse operator AND can effectively use this created sparsity in its arguments.

Effect of reformulation

A modified version of the above example was sent to us by a customer. While the original formulation took several minutes to execute for a given large dataset, the reformulation only took a few seconds.

A bit of general advice

Perhaps a modeling style which avoids the need for substitutions is best. The easy way is to let AIMMS identify the places in which such substitutions can be made by switching the options in the option category Aimms - Tuning - Substitute Lower Dimension Expressions to appropriate settings. The disadvantage of this easy method is that some opportunities are missed as AIMMS cannot guarantee the equivalence of the formulations, and some replacements are missed. For instance, in the above example, AIMMS will create an identifier for both TRDistance(sr,st) <= MaxTRDistance(st) and TRDistance(dr,dt) <= MaxTRDistance(dt), even though only one suffices. You can avoid substitutions by keeping your expressions brief relating only a few identifiers at a time. This will also help to keep your model readable.

13.2.5 Parameters with non-zero defaults

Sparse execution expects 0.0's

Sparse execution is based on the effect of the number 0.0 on addition and multiplication. When other numbers are used as a default, all possible elements of these parameters need to be considered rather than only the stored ones. The advice is not to use such parameters in intensive computations. In the example below, the summation operator will need to consider every possible element of P rather than only its non-zeros.

```
Parameter P {
    IndexDomain  :  (i,j);
    Default      :  1;
    Body         : {
        CountP := Sum( (i,j), P(i,j) )
    }
}
```

Appropriate use of default

Identifiers with a non-zero default, may be convenient, however, in the interface of your application as the GUI of AIMMS can display non-default elements only.

The NonDefault function

For parameters with a non-zero default, you still may want to execute only for its non-default values. For this purpose, the function NonDefault has been introduced. This function allows one to limit execution to the data actually stored in such a parameter. Consider the following example where the non defaults of P are summed:

```
CountP := Sum( (i,j)| NonDefault(P(i,j)), P(i,j) );
```

In the above example the summation is limited to only those entries in P(i,j) that are stored. If you would rather use familiar algebraic notation, instead of the dedicated function NonDefault, you can change the above example to:

```
CountP := Sum( (i,j) | P(i,j) <> 1, P(i,j) );
```

This statement also sums only the non-default values of P. AIMMS recognizes this special use of the <> operator as actually using the NonDefault function; the summation operator will only consider the tuples (i,j) that are actually stored for P.

Suffices of variables

Note that the suffices .Lower and .Upper of variables are like parameters with a non-zero default. For example in a free variable the default of the .lower suffix is -inf and the default of the suffix .upper is inf.

13.2.6 Index ordering

Index ordering

In rare cases, the particular order of indices in a statement may have an adverse effect on its performance. The efficiency aspects of index ordering, when they occur, are inarguably the most difficult to understand and recognize. Again, this inefficiency is best explained using an example.

An artificial example

Consider the following assignment statement:

```
FS(i,k) := Sum( j, A(i,j) * B(j,k) );
```

If A(i,j) and B(j,k) are binary parameters, where

- for any given i, the parameter A(i,j) maps to a single j, and,
- for any given j, the parameter B(j,k) maps to a single k,

one would intuitively expect that the assignment could be executed rather efficiently. When actually executing the statement, it may therefore come as an unpleasant surprise that it takes a seemingly unexplainable amount of time.

An analysis

In the qualitative analysis above, implicitly the index order i selects j, and j selects a few k's, or, in AIMMS terminology, a running index order [i,j,k]. The actual running index order of AIMMS is, however, first the indices [i,k] from the assignment operator, followed by the index [j] from the summation operator: [i,k,j]. The effect of the actual index order is that, for a given value of index i, the relevant values of index k cannot be restricted using the parameter chain A(i,j)-B(j,k) without the aid of an intermediate running index j. Consequently, AIMMS has to try every combination of (i,k).

Reformulation

Given the above analysis, the preferred index ordering [i,j,k] can be accomplished by introducing an intermediate identifier FSH(i,j,k), and replacing the original assignment by the following statements.

```
FSH(i,j,k) := A(i,j) * B(j,k);
FS(i,k) := Sum( j, FSH(i,j,k) );
```

With a real-life example, where the range of the indices i, j and k contained over 10000 elements, the observed improvement was more than a factor 50.

Not for +

A similar improvement could be obtained for the following example:

```
FSP(i,k) := Sum( j, A(i,j) + B(j,k) );
```

Here a value is computed for each (i,k) of FSP, because, for every i, there is a non-zero A(i,j), and for every k, there is a non-zero B(j,k).

13.2.7 Set element ordering

Data entry order

By default, all elements in a root set are numbered internally by AIMMS in a consecutive manner according to their *data entry order*, i.e. the order in which the elements have been added to the set. Such additions can be either explicit or implicit, and may take place, for example when the model text contains references to explicit elements in the root set, or by reading the set from files, databases, or cases.

Multi-dimensional storage

The storage of multidimensional data defined over a root set is always based on this internal and consecutive numbering of root set elements. More explicitly, all tuple-value pairs associated with a multidimensional identifier are stored according to a strict right-to-left ordering based on the respective root set numberings.

Indexed execution

By default, all indexed execution taking place in AIMMS, either through implied loops induced by indexed assignments or through explicit FOR loops, employs the same strict right-to-left ordering of root set elements. Thus, there is a perfect match between the execution order and the order in which identifiers referenced in such loops are stored internally. As a consequence, it is very easy for AIMMS to synchronize the tuple for which execution is currently due with an ordered route through all the non-zero tuples in the identifiers involved in the statement. This principle is the basis of the sparse execution engine underlying AIMMS.

Execution over ordered sets

Inefficiency is introduced if the elements in a set over which a loop takes place have been ordered differently from the data entry order, either because of an ordering principle specified in the OrderBy attribute of the set declaration or through an explicit Sort operation. Consequently, there will no lomger be a direct match between the execution order of the loop and the storage order of the non-zero identifier values. Depending on the precise type of statement, this may result in no, slight or serious increase in the execution time of the statement, as AIMMS may have to perform randomly-placed lookups for particular tuples. These random lookups are much more time consuming than running over the data only once in an ordered fashion.

Effect on FOR loops

In particular, you should avoid using FOR statements in which the running index is an index in a set with a nondefault ordering whenever possible. If not, AIMMS is forced to execute such FOR statements using the imposed nondefault ordering and, as a result, *all* identifier lookups within the FOR loop are random. In such a situation, you should carefully consider whether ordered execution is really essential. If not, it is advisable to leave the original set unordered, and

create an ordered *subset* (containing all the elements of the original set) for use when the nondefault element ordering is required.

Effect on assignments

In most situations, the efficiency of indexed assignments is not affected by the use of indices in sets with a nondefault ordering. AIMMS has only to rely on the nondefault ordering if an assignment contains special order-dependent constructs such as lag and lead operators. In all other cases, AIMMS can use the default data entry order.

Complete reordering

If a nondefault ordering of some sets in your model causes a serious increase in execution times, you may want to apply the CLEANDEPENDENTS statement (see also Section 25.3) to those roots sets that are the cause of the increase of execution times. The CLEANDEPENDENTS statement will fully renumber the elements in the root set according to their current ordering, and rebuild all data defined over it according to this new numbering.

Use sparingly

As all identifiers defined over the root set have to be completely rebuilt, CLEANDEPENDENTS is an inherently expensive operation. You should, therefore, only use it when really necessary.

13.2.8 Using AIMMS' advanced diagnostics

Using diagnostic warnings

In order to help you create correct and efficient applications, AIMMS is regularly extended with diagnostics that incorporate the recognition of new types of problematic situations. These diagnostics may help you detect model formulations that lead to sub-optimal performance and/or ambiguous results. These diagnostics can be controled through various options in the **Warning** category.

Apply diagnostics regularly

As the list of diagnostic options is regularly extended, and some of the formulation problems depend on the model data and, thus, can only be detected at runtime, you are advised to apply the diagnostics provided by AIMMS on a regular basis during your application tests. Section 8.4.4 describes a way in which you can switch on all the diagnostic options by just changing the value of the two options strict_warning_default and common_warning_default.

Diagnostic options

Below we provide a list of performance-related diagnostics that may help you tune the performance of your model:

- **Warning_repeated_iterative_evaluation:** If the arguments of an iterative operator do not depend on some of the indices, the iterative operator is repeatedly evaluated with the same result. Consider the assignment `a(i) := sum(j,b(j));` in which the sum does not depend on the index `i` and so the same value is computed for every value of `i`. See also Subsection 13.2.4.
- **Warning_unused_index:** If an index is not used inside the argument(s) or index domain condition of an iterative iterator, this leads to inefficient execution. In the assignment `a(i) := sum((j,k),b(i,j));`, the index `k` is not used in the summation. Further, when an index in the index domain of a constraint is not used inside the definition of that constraint this is likely to lead to the generation of duplicate rows.
- **Warning_duplicate_row:** At the end of generating a mathematical program it is verified that there are no duplicate rows inside that mathematical program. This might be caused by two constraints having the same definition. Besides consuming more memory, duplicate rows cause the problem to become degenerate and may cause the problem to become more difficult to solve. This warning is not supported for mathematical programs of type MCP or MPCC because, for these types the row col mapping is also relevant and duplicate rows cannot be simply eliminated.
- **Warning_duplicate_column:** At the end of generating a mathematical progrram it is verified that there are no duplicate columns inside that mathematical program. Besides consuming more memory, duplicate columns result in the generated mathematical program having non-unique solutions.
- **Warning_trivial_row:** Generating and eliminating trivial rows such as 0 <= 1 takes time.

The help for the option category `AIMMS - Progress, errors & warnings - warnings` provides more information on these options.

13.3 Summary

The recipe boils down to three steps

This chapter consists of a recipe for fine tuning an existing AIMMS application such that AIMMS more efficiently executes the definitions and statements and efficiently generates the constraints. The recipe consists of the following three steps:

- First, construct active subsets by removing all elements for which the variables are fixed in advance. These active subsets should then be used throughout your core model. This reduces the work each time, even in the evaluation of the index domain conditions to be constructed next.

- Second, construct index domain conditions for the parameters, variables, and constraints of the core model. This will make several dense expressions seemingly execute more sparse, because only a limited number of elements are evaluated. Especially with variables and constraints this avoids the generation of columns for fixed variables and empty constraints. Thus the number of bottlenecks in your application is further reduced.
- Finally, use the AIMMS profiler to pinpoint those assignment statements, FOR loops, and constraints that still absorb a considerable amount of computation time, and analyze and possible modify them. A checklist that can be used for this analysis has been presented in Section 13.2.

Part V

Optimization Modeling Components

Chapter 14

Variable and Constraint Declaration

Terminology

The word variable does not have a uniform meaning. In general, programmers view a variable as a known but varying quantity that receives its value through direct assignments. *However, in the context of constraints in* AIMMS*, the word variable denotes an unknown quantity.* Constraints can be grouped together to form a system of simultaneous equations and/or inequalities, which is referred to as a *mathematical program*. Variables in a mathematical program are assigned values when a solver (a solution algorithm) finds a solution for the unknowns in the system.

Similarity of parameters and variables

When used outside the scope of constraints and the solution of mathematical programs, variables in AIMMS behave essentially the same as parameters in AIMMS. Like parameters, variables can be initialized, used as known quantities in assignment statements, and be referred to as data from within the graphical user interface.

14.1 `Variable` declaration and attributes

Declaration and attributes

Variables have some additional attributes above those of parameters. These extra attributes are used to steer a solver, or to hold additional information about solution values provided by the solver. The possible attributes of variables are given in Table 14.1.

Index domain for variables

By specifying the `IndexDomain` attribute you can restrict the domain of a variable in the same way as that of a parameter. For variables, however, the domain restriction has an additional effect. During the generation of individual constraints AIMMS will reduce the size of the generated mathematical program by including only those variables that satisfy all domain restrictions.

The Range attribute

The values of the `Range` attribute of variables determine the bounds that are passed on to the solver. In addition, during an assignment, the `Range` attribute restricts the range of allowed values that can be assigned to a particular interval (as for parameters). The possible values for the `Range` attribute are:

Attribute	Value-type	See also page
IndexDomain	*index-domain*	42
Range	*range*	43
Default	*constant-expression*	44
Unit	*unit-expression*	45, 513
Priority	*expression*	
NonvarStatus	*expression*	
RelaxStatus	*expression*	
Property	NoSave, *numeric-storage-property*, Inline SemiContinuous, Basic, Stochastic, Adjustable ReducedCost, ValueRange, CoefficientRange, *constraint-related-sensitivity-property*	34, 45
Text	*string*	19
Comment	*comment string*	19
Definition	*expression*	34, 44
Stage	*expression*	216, 316
Dependency	*expression*	216, 343

Table 14.1: Variable attributes

- one of the predefined ranges Real, Nonnegative, Nonpositive, Integer or Binary,
- any one of the interval expressions $[a,b]$, $[a,b)$, $(a,b]$ or (a,b), where a and b can be a constant number, inf, -inf, or a parameter reference involving some or all of the indices on the index list of the declared variable,
- any enumerated integer set expression, e.g. $\{a \;..\; b\}$ with a and b as above, or
- an integer set identifier.

If you specify Real, Nonnegative, Nonpositive, or an interval expression, AIMMS will interpret the variable as a continuous variable. If you specify Integer, Binary or an integer set expression, AIMMS will interpret the variable as a binary or integer variable.

The following example illustrates a simple variable declaration. *Example*

```
Variable Transport {
    IndexDomain  : (i,j) in Connections;
    Range        : [ MinTransport(i), Capacity(i,j) ];
}
```

The declaration of the variable Transport(i,j) sets its lower bound equal to MinTransport(i) and its upper bound to Capacity(i,j). When generating the mathematical program, the variable Transport will only be generated for those

tuples (i,j) that lie in the set Connections. Note that the specification of the lower bound only uses a subdomain (i) of the full index domain of the variable (i,j).

The .Lower and .Upper suffices

Besides using the Range attribute to specify the lower and upper bounds, you can also use the .Lower and .Upper suffices in assignment statements to accomplish this task. The .Lower and .Upper suffices are attached to the name of the variable, and, as a result, the corresponding bounds are defined for the entire index domain. This may lead to increased memory usage when variables share their bounds for slices of the domain. For this reason, you are advised to use the Range attribute as much as possible when specifying the lower and upper bounds.

When allowed

You can only make a bound assignment with either the .Lower or .Upper suffix when you have not used a parameter reference (or a non-constant expression) at the corresponding position in the Range attribute. Bound assignments via the .Lower and .Upper suffices must always lie within the range specified in the Range attribute.

Example

Consider the variable Transport declared in the previous example. The following assignment to Transport.Lower(i,j) is not allowed, because you have already specified a parameter reference at the corresponding position in the Range attribute.

```
Transport.Lower(i,j) := MinTransport(i) ;
```

On the other hand, given the following declaration,

```
Variable Shipment {
    IndexDomain  : (i,j) in Connections;
    Range        : Nonnegative;
}
```

the following assignment is allowed:

```
Shipment.Lower(i,j) := MinTransport(i);
```

AIMMS will produce a run-time error message if any value of MinTransport(i) is less than zero, because this violates the bound in the Range attribute of the variable Shipment.

The Default attribute

Variables that have not been initialized, evaluate to a default value automatically. These default values are also passed as initial values to the solver. You can specify the default value using the Default attribute. The value of this attribute *must* be a constant expression. If you do not provide a default value, AIMMS will use a default of 0.

The Unit attribute

Providing a Unit for every variable and constraint in your model will help you in a number of ways.

- AIMMS will help you to check the consistency of all your constraints and assignments in your model, and
- AIMMS will use the units to scale the model that is sent to the solver.

Proper scaling of a model will generally result in a more accurate and robust solution process. You can find more information on the definition and use of units to scale mathematical programs in Chapter 32.

The Definition attribute

It is not unusual that symbolic constraints in a model are equalities defining just one variable in terms of others. Under these conditions, it is preferable to provide the definition of the variable through its Definition attribute. As a result, you no longer need to specify extra constraints for just variable definitions. In the constraint listing, the constraints associated with a defined variable will be listed with a generated name consisting of the name of the variable with an additional suffix "_definition".

Example

The following example defines the total cost of transport, based on unit transport cost and actual transport taking place.

```
Variable TransportCost {
    Definition : sum( (i,j), UnitTransportCost(i,j)*Transport(i,j) );
}
```

14.1.1 The Priority, Nonvar and RelaxStatus attributes

The Priority attribute

With the Priority attribute you can assign priorities to integer variables (or continuous variables when using the solver BARON). The value of this attribute must be an expression using some or all of the indices in the index domain of the variable, and must be nonnegative and integer. All variables with priority zero will be considered last by the branch-and-bound process of the solver. For variables with a positive priority value, those with the highest priority value will be considered first.

The .Priority suffix

Alternatively, you can specify priorities through assignments to the .Priority suffix. This is only allowed if you have not specified the Priority attribute. In both cases, you can use the .Priority suffix to refer to the priority of a variable in expressions.

Use of priorities

The solution algorithm (i.e. solver) for integer and mixed-integer programs initially solves without the integer restriction, and then adds this restriction one variable at a time according to their priority. By default, all integer variables have equal priority. Some decisions, however, have a natural order in time or space. For example, the decision to build a factory at some site comes before

the decision to purchase production capacity for that factory. Obeying this order naturally limits the number of subsequent choices, and could speed up the overall search by the solution algorithm.

The NonvarStatus attribute

You can use the NonvarStatus attribute to tell AIMMS which variables should be considered as parameters during the execution of a SOLVE statement. The value of the NonvarStatus attribute must be an expression in some or all of the indices in the index list of the variable, allowing you to change the nonvariable status of individual elements or groups of elements at once.

Positive versus negative values

The sign of the NonvarStatus value determines whether and how the variable is passed on to the solver. The following rules apply.

- If the value is 0 (the default value), the corresponding individual variable is generated, along with its specified lower and upper bounds.
- If the value is negative, the corresponding individual variable is still generated, but its lower and upper bounds are set equal to the current value of the variable.
- If the value is positive, the corresponding individual variable is no longer generated but passed as a constant to the solver.

When you specify a negative value, you will still be able to inspect the corresponding reduced cost values. In addition, you can modify the nonvariable status to zero without causing AIMMS to regenerate the model. When you specify a positive value, the size of the mathematical program is kept to a minimum, but any subsequent changes to the nonvariable status will require regeneration of the model constraints.

The .NonVar suffix

Alternatively, you can change the nonvariable status through assignments to the .NonVar suffix. This is only allowed if you have not specified the NonvarStatus attribute. In both cases, you can use the .NonVar suffix to refer to the variable status of a variable in expressions.

When to change the nonvariable status

By altering the nonvariable status of variables you are essentially reconfiguring your mathematical program. You could, for instance, reverse the role of an input parameter (declared as a variable with negative nonvariable status) and an output variable in your model to observe what input level is required to obtain a desired output level. Another example of temporary reconfiguration is to solve a smaller version of a mathematical program by first discarding selected variables, and then changing their status back to solve the larger mathematical program using the previous solution as a starting point.

The RelaxStatus attribute

With the RelaxStatus attribute you can tell AIMMS to relax the integer restriction for those tuples in the domain of an integer variable for which the value of the relax status is nonzero. AIMMS will generate continuous variables for such tuples instead, i.e. variables which may assume any real value between their bounds.

The .Relax suffix

Alternatively, you can relax integer variables by making assignments to the .Relax suffix. This is only allowed if you have not specified the RelaxStatus attribute. In both cases, you can use the .Relax suffix to refer to the relax status of a variable in expressions.

When to relax variables

When solving large mixed integer programs, the solution times may become unacceptably high with an increase in the number of integer variables. You can try to resolve this by relaxing the integer condition of some of the integer variables. For instance, in a multi-period planning model, an accurate integer solution for the first few periods and an approximating continuous solution for the remaining periods may very well be acceptable, and at the same time reduce solution times drastically.

Effect on mathematical program type

As you will see in Chapter 15, there are several types of mathematical programs. By changing the nonvariable and/or relax status of variables you may alter the type of your mathematical program. For instance, if your constraints contains a nonlinear term x*y, then changing the nonvariable status of either x or y will change it into a linear term. Eventually, this may result in a nonlinear mathematical program becoming a linear one. Similarly, changing the nonvariable or relax status of integer variables may at some point change a mixed integer program into a linear program.

14.1.2 Variable properties

Properties of variables

Variables can have one or more of the following properties: NoSave, Inline, SemiContinuous, ReducedCost, CoefficientRange, ValueRange, Stochastic, and Adjustable. They are described in the paragraphs below.

Use of PROPERTY statement

You can also change the properties of a variable during the execution of your model by calling the PROPERTY statement. Identifier properties are changed by adding the property name as a suffix to the identifier name in a PROPERTY statement. When the value is set to off, the property no longer holds.

The NoSave property

With the property NoSave you indicate that you do not want to store data associated with this variable in a case. This property is especially suited for those identifiers that are intermediate quantities in the model, and that are not used anywhere in the graphical end-user interface.

Inline variables

With the property Inline you can indicate that AIMMS should substitute all references to the variable at hand by its defining expression when generating the constraints of a mathematical program. Setting this property only makes sense for defined variables, and will result in a mathematical program with less rows and columns but with a (possibly) larger number of nonzeros. After the mathematical program has been solved, AIMMS will compute the level values of all inline variables by evaluating their definition. However, no sensitivity information will be available.

Semi-continuous variables

To any continuous or integer variable you can assign the property SemiContinuous. This indicates to the solver that this variable is either zero, or lies within its specified range. Not all solvers support semi-continuous variables. In the latter case, AIMMS will automatically add the necessary constraints to the model.

14.1.3 Sensitivity related properties

Basic, superbasic, and nonbasic variables

With the Basic property you can instruct AIMMS to retrieve basic information of a specific variable from the solver. If retrieved, basic information can be accessed through the .Basic suffix. The basic information is presented as an element in the predefined AIMMS set AllBasicValues (i.e. {*Basic, Nonbasic, Superbasic*}). In linear programming a variable will either be basic or nonbasic, while in nonlinear programming the number of variables with zero reduced cost can be larger than the number of constraints. The solution algorithm then divides these variables into so-called *basics* and *superbasics*. The basic variables define a square system of nonlinear equations which is solved for fixed values of the remaining variables. The superbasics are assigned a fixed value between their bounds, while the nonbasics take their value at a bound.

The ReducedCost property

You can use the ReducedCost property to specify whether you are interested in the reduced cost values of the variable after each SOLVE step. Storing the reduced costs of all variables may be very memory consuming, therefore, the default in AIMMS is not to store these values. If reduced costs are requested, the stored values can be accessed through the suffices .ReducedCost or .m.

Interpretation of reduced cost

The reduced cost indicates by how much the cost coefficient in the objective function should be reduced before the variable becomes active (off its bound). By definition, the reduced cost value of a variable between its bounds is zero. The precise mathematical interpretation of reduced cost is discussed in most text books on mathematical programming. Note: if a basic or superbasic variable has a reduced cost of zero then it will be displayed as 0.0, but if a nonbasic variable has a reduced cost of zero then it will be displayed as ZERO.

Unit of reduced cost

When the variables in your model have an associated unit (see Chapter 32), special care is required in interpreting the values returned through the .ReducedCost suffix. To obtain the reduced cost in terms of the units specified in the model, the values of the .ReducedCost suffix must be scaled as explained in Section 32.5.1.

The property CoefficientRange

With the property CoefficientRange you request AIMMS to conduct a first type of sensitivity analysis on this variable during a SOLVE statement of a linear program. The result of this sensitivity analysis are three parameters, representing the smallest, nominal, and largest values for the *objective coefficient* of the variable so that the optimal basis remains constant. Their values are accessible through the suffices .SmallestCoefficient, .NominalCoefficient and .LargestCoefficient.

The property ValueRange

With the property ValueRange you request AIMMS to conduct a second type of sensitivity analysis during a SOLVE statement of a linear program. The result of the sensitivity analysis are two parameters, representing the smallest and largest values that the *variable* can take while holding the objective value constant. Their values are accessible through the .SmallestValue and .LargestValue suffices.

Linear programs only

AIMMS only supports the sensitivity analysis conducted through the properties CoefficientRange and ValueRange for linear mathematical programs. If you want to apply these types of analysis to the final solution of a mixed-integer program, you should fix all integer variables to their final solution (using the .NonVar suffix) and re-solve the resulting mathematical program as a linear program (e.g. by adding the clause WHERE type:='lp' to the SOLVE statement).

Storage and computational costs

Setting any of the properties ReducedCost, CoefficientRange or ValueRange may result in an increase of the memory usage. In addition, the computations required to compute the ValueRange may considerably increase the total solution time of your mathematical program.

Constraint related properties

Whenever a defined variable (which is not declared Inline) is part of a mathematical program, AIMMS implicitly adds a constraint to the generated model expressing this definition. In addition to the variable-related sensitivity properties discussed in this section, you can specify the constraint-related sensitivity properties ShadowPrice, RightHandSideRange and ShadowPriceRange (see also Section 14.2) for such variables to obtain the sensitivity information that can be related to these constraint. You can access the requested sensitivity information by appending the associated suffices to the name of the defined variable.

14.1.4 Uncertainty related properties and attributes

Stochastic programming and robust optimization

The AIMMS modeling language offers facilities for both stochastic programs and robust optimization models. For both types of models you can specify special Variable properties and attributes to define uncertainty-related relationships.

The Stochastic property

Through the Stochastic property you can indicate that, within a stochastic model, the variable can hold scenario-dependent solutions. AIMMS will add a Stage attribute for every variable for which the Stochastic property has been set.

The Stage attribute

The value of the Stage attribute must be a numerical expression evaluating to in integer number indicating the stage at the end of which the variable takes its value during the solution process of a stochastic model. Stochastic programming, and the Stochastic property and Stage attribute are discussed in full detail in Section 19.2.

The Adjustable property

By setting the Adjustable property for a variable, you indicate that a variable in a robust optimization model has a functional dependency on some or all of the uncertain parameters in the model. If you declare a variable to be adjustable, the Dependency attribute also becomes available for that variable.

The Dependency attribute

Through the Dependency attribute you specify the precise collection of uncertain parameters on which the variable at hand depends. At this moment, AIMMS only supports affine relations between uncertain parameters and adjustable variables. The precise semantics of the Dependency attribute is discussed in Section 20.4.

14.2 Constraint declaration and attributes

Definitions

Constraints form the major mechanism for specifying a mathematical program in AIMMS. They are used to restrict the values of variables with interlocking relationships. *Constraints* are numerical relations containing expressions in terms of variables, parameters and constants.

Constraint attributes

The possible attributes of constraints are given in Table 14.2.

Domain restriction for constraints

Restricting the domain of constraints through the IndexDomain attribute influences the matrix generation process. Constraints are generated only for those tuples in the index domain that satisfy the domain restriction.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	42
Unit	*unit-valued expression*	45, 513
Text	*string*	19
Comment	*comment string*	19
Definition	*expression*	44, 211
Property	NoSave, Sos1, Sos2, IndicatorConstraint Level, Bound, Basic, ShadowPrice, RightHandSideRange, ShadowPriceRange, IsDiversificationFilter, IsRangeFilter, IncludeInLazyConstraintPool, IncludeInCutPool, Chance	34, 45 213
SosWeight	*sos-weights*	
ActivatingCondition	*expression*	
Probability	*expression*	225, 341
Aproximation	*element-expression*	225, 342

Table 14.2: Constraint attributes

The Definition attribute

With the Definition attribute of a constraint you specify a numerical relationship between variables in your model. Without a definition a constraint is indeterminate. Constraint definitions consist of two or three expressions separated by one of the relational operators "=", ">=" or "<=".

Example

The following constraints express the simultaneous requirements that the sum of all transports from a city i must not exceed Supply(i), and that for each city j the Demand(j) must be met.

```
Constraint SupplyConstraint {
    IndexDomain  : i;
    Unit         : kton;
    Definition   : sum( j, Transport(i,j) ) <= Supply(i);
}
Constraint DemandConstraint {
    IndexDomain  : j;
    Unit         : kton;
    Definition   : sum( i, Transport(i,j) ) >= Demand(j);
}
```

Allowed relationships

If a and b are expressions consisting of only parameters and $f(x,\dots)$ and $g(x,\dots)$ are expressions containing parameters and variables, the following two kinds of relationships are allowed.

$$a \le f(x,\dots) \le b \qquad \text{or} \qquad f(x,\dots) \gtrless g(x,\dots)$$

where $\gtrless$ denotes any of the relational operators "=", ">=" or "<=". Either a or b can be omitted if there is no lower or upper bound on the expression $f(x,\dots)$,

respectively. When both *a* and *b* are present, the constraint is referred to as a *ranged* constraint. The expressions may have linear and nonlinear terms, and may utilize the full range of intrinsic functions of AIMMS except for the random number functions.

Conditional expressions in constraints

You must take extreme care to ensure continuity when the constraints in your model contain logical conditions that include references to variables. Such constraints are viewed by AIMMS as nonlinear constraints, and thus can only be passed to a solver that can handle nonlinearities. It is possible that the outcome of a logical condition, and thus the form of the constraint, changes each time the underlying solver asks AIMMS for function values and gradients. For example, if x(i) is a decision variable, and a constraint contains the expression

```
sum[ i, if ( x(i) > 0 ) then  x(i)^2 endif ]
```

it may or may not contain the term x(i)^2, depending on the current value of x(i). In this example, both the expression and its gradient are continuous functions at x(i) = 0.

14.2.1 Constraint properties

The Property attribute

With the Property attribute you can specify further characteristics of the constraint at hand. The possible properties of a constraint are NoSave, Sos1, Sos2, Level, Bound, Basic, ShadowPrice, RightHandSideRange, and ShadowPriceRange.

The NoSave property

When you specify the NoSave property you indicate that you do not want AIMMS to store data associated with the constraint in a case, regardless of the specified case identifier selection.

14.2.2 SOS properties

The SOS properties

The constraint types Sos1 and Sos2 are used in mixed integer programming, and mutually exclusive. In the context of mathematical programming SOS is an acronym for Special Ordered Sets. A SOS set is associated with every (individual) constraint of type Sos1 or Sos2.

Additional SOS attribute

When you specify that a constraint is of type Sos1 or Sos2, an additional SOS-specific attributes becomes available, namely the SosWeight attributes. With this attributes, you can provide further information to the solver about the contents and ordering of the SOS set to be associated with the constraint.

Sos1 constraints

A type Sos1 constraint specifies to the solver that at most one of the variables within the SOS set associated with the constraint is allowed to be nonzero, while all other variables in the SOS set must be zero. Inside a Sos1 constraint all variables in the SOS set must have a lower bound of zero and an upper bound greater than zero.

Sos2 constraints

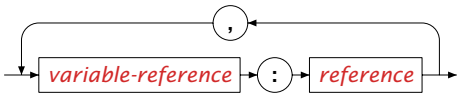
A type Sos2 constraint specifies to the solver that at most two consecutive variables within the SOS set associated with the constraint are allowed to be nonzero, while all other variables within the SOS set must be zero. All individual variables within the SOS set must have a lower bound of zero and an upper bound greater than zero. The order of the individual variables within the SOS set is determined by their weights (as specified in the SosWeight attribute), where the ordering is from low to high weight.

The SosWeight attribute

With the SosWeight attribute you must specify the contents of the SOS set to be associated with a Sos1 or Sos2 constraint, as well the ordering of its elements. Section 7.5 of the AIMMS book on Optimization Modeling describes how these weights are used during the branch-and- bound process. The syntax of the SosWeight attribute is as follows.

Syntax

sos-weights :



Within the SosWeight attribute you can (but need not) specify a weight for every variable occurring in the constraint. Each weight must be an expression using all the indices in the index domain of the variable plus some or all of the indices in the index domain of the constraint. All weights specified for a particular constraint must be unique, i.e. you cannot specify the same weight for two (individual) variables. The SOS set to be associated with the constraint will be constructed from all variables—within the domain of both the constraint and variable—for which a nonzero weight has been specified in the SosWeight attribute, i.e. if the value of the specified weight is 0.0 for a particular tuple, the corresponding individual variable will not be included in the SOS set. The ordering of variables within the SOS set is from low to high weight.

Consistency

If you do not specify SOS weights, AIMMS will make sure that ordering of variables in each SOS set is consistent over all SOS sets. If you specify SOS weights yourself, you have to make sure that the variable orderings of all SOS sets of type Sos2 are consistent, or your model might become infeasible if feasibility requires that two adjacent variables in one SOS set become nonzero, which are ordered inconsistently in another SOS set. Therefore, AIMMS requires that you

specify the SosWeight attributes for *all* SOS constraints in your model, whenever you specify it for *one* SOS constraint.

The following is specification of Sos2 constraint to determine the variable y piece-wise linearly from a variable x(i). *Example*

```
Constraint DetermineY {
    Property     : Sos2;
    Definition   : y = sum[ i, x(i)*c(i) ];
    SosWeight    : x(i) : XWeight(i);
}
```

14.2.3 Solution pool filtering

During the solution process of a MIP problem, the solvers CPLEX and GUROBI are capable of storing multiple feasible integer solutions in a solution pool, for instance, to capture solutions with attractive properties that are hard to express in a linear fashion. *Solution pool*

While populating the solution pool, CPLEX offers advanced filtering capabilities, allowing you to control which solutions end up in the solution pool. CPLEX provides two predefined ways to filter solutions: *Filtering*

- if you want to filter solutions based on their difference as compared to a reference solution, use a *diversity* filter, or
- if you want to filter solutions based on their validity in an additional linear constraint, use a *range* filter.

To enable filters the CPLEX option Do_Populate need to be on.

A diversity filter allows you to generate solutions that are similar to (or different from) a set of reference values that you specify for a set of binary variables. In particular, you can use a diversity filter to generate more solutions that are similar to an existing solution or to an existing partial solution. Several diversity filters can be used simultaneously, for example, to generate solutions that share the characteristics of several different solutions. *Diversity filters*

In AIMMS, a constraint is used as a diversity filter if the constraint property IsDiversificationFilter has been set. In a diversification filter, the Abs function is used to measure the distance from a given binary variable, and all variables should only occur as the argument of an Abs function. *The IsDiversificationFilter property*

This following diversification filter forces the solutions to have a distance of at least 1 from variable x. *Example*

```
Constraint filter1 {
    Property     :  IsDiversificationFilter;
    Definition   :  Abs(x - 1) >= 1;
}
```

A range filter allows you to generate solutions that obey a new constraint, specified as a linear expression within a range. Range filters can be used to express diversity constraints that are more complex than the standard form implemented by diversity filters. In particular, range filters also apply to general integer variables, semi-integer variables, continuous variables, and semi-continuous variables, not just to binary variables. *Range filters*

In AIMMS, a constraint is used as a range filter if the constraint property IsRangeFilter has been set for the constraint. *The IsRangeFilter property*

The following range filter specifies that any solution to be added to the solution pool should satisfy the following constraint. *Example*

```
Contraint filter2 {
   Property     :  IsRangeFilter;
   Definition   :  x + y + z >= 2;
}
```

14.2.4 Indicator constraints, lazy constraints and cut pools

An indicator constraint is a new way of controlling whether or not a constraint takes effect, based on the value of a binary variable. Traditionally, such relationships are expressed by so-called big-*M* formulations. Big-*M* formulations, however, can introduce unwanted side-effects and numerical instabilities into a mathematical program. Using indicator constraints, such relationships between a constraint and a variable can be directly expressed in the constraint declaration. Indicator constraints are supported by the solvers CPLEX, GUROBI and ODH-CPLEX. *Indicator constraints*

You can specify that a constraint is an indicator constraint by settings it IndicatorConstraint property. For indicator constraints, a new attribute called ActivatingCondition will become available in the constraint declaration. *The IndicatorConstraint property*

Through the ActivatingCondition attribute you can specify under which condition the constraint definition should become active during the solution process. Its value should be an expression of the form *The ActivatingCondition attribute*

binary-variable = *expression*

where the *expression* must take one of the values 0 or 1. Note: stochastic variables and parameters are not allowed inside an activation condition.

Example

Consider the following big-*M* constraint

```
Constraint BigMConstraint {
    Definition : x1 + x2 <= M*y;
}
```

where y is a binary variable. Using the IndicatorConstraint property, the constraint can be reformulated as an indicator constraint as follows

```
Constraint NonBigMConstraint {
    Property            : IndicatorConstraint;
    ActivatingCondition  : y = 0;
    Definition          : x1 + x2 = 0;
}
```

The constraint only becomes effective, whenever the binary variable y takes the value 0. To solve the model with the indicator constraint, you need the CPLEX, GUROBI or ODH-CPLEX solver.

Lazy constraints

Sometimes, for a MIP formulation, a user can already identify a group of constraints that are unlikely to be violated (lazy constraints). Simply including these constraints in the original formulation could make the LP subproblem of a MIP optimization very large or too expensive to solve. CPLEX, GUROBI and ODH-CPLEX can handle problems with lazy constraints more efficiently, and therefore AIMMS allows you to identify lazy constraints in your model.

The IncludeInLazyConstraintPool property

You can specify that a constraint should be added to the pool of lazy constraints considered by CPLEX, GUROBI or ODH-CPLEX by setting the property IncludeInLazyConstraintPool. You need the CPLEX, GUROBI or ODH-CPLEX solver to use this constraint property. When solving your MIP model, CPLEX, GUROBI and ODH-CPLEX will only consider these constraints when they are violated.

User cut pools

As discussed in Section 15.2, AIMMS allows you to add cuts to your mathematical program on the fly during the solution process by using the CallbackAddCut callback. However, when the set of cuts you want to generate is fixed and known upfront, using the CallbackAddCut may add significant overhead to the solution process of your model while you don't need its flexibility. For those situations, CPLEX allows you to specify a fixed pool of user cuts during the generation of your mathematical program.

The IncludeInCutPool property

By setting the constraint property IncludeInCutPool you can indicate that this constraint should be included in the pool of user cuts associated with your mathematical program. You need the CPLEX solver to use this constraint property. When solving your MIP model, CPLEX will consider the user cuts added in this manner when appropriate.

14.2.5 Constraint levels, bounds and marginals

Constraint levels and bounds

A constraint in AIMMS can conceptually be divided such that one side consists of all variable terms, whereas the other side consists of all remaining constant terms. The *level* value of a constraint is the accumulated value of the variable terms, while the constant terms make up the *bound* of the constraint.

The Level, Bound, Basic and ShadowPrice properties

With the Level, Bound, Basic and ShadowPrice properties you indicate whether you want to store (and have access to) particular parametric data associated with the constraint.

- When you specify the Level property AIMMS will retain the level values of the constraint as provided by the solver. You can access the level values of a constraint by using the constraint name as if it were a parameter.
- By specifying the Bound property, AIMMS will store the upper and lower bound of the constraint as employed by the solver. You get access to the bounds by using the .Lower and .Upper suffices with the constraint identifier.
- If the Basic property has been specified, AIMMS stores basic information is available through the .Basic suffix as an element in of the predefined AIMMS set AllBasicValues. A constraint is said to be basic (nonbasic or superbasic) if its associated slack variable is basic (nonbasic or superbasic).
- With the ShadowPrice property you indicate that you want to store the shadow prices as computed by the solver. You can access these shadow prices by means of the .ShadowPrice attribute.

Interpretation of shadow prices

The shadow price (or dual value) of a constraint is the marginal change in the objective value with respect to a change in the right-hand side (i.e. the constant part) of the constraint. This value is determined by the solver after a SOLVE statement has been executed. The precise mathematical interpretation of the shadow price is discussed in detail in many text books on mathematical programming. Note: if a basic or superbasic constraint has a shadow price of zero then it will be displayed as 0.0, but if a nonbasic constraint has a shadow price of zero then it will be displayed as ZERO.

Unit of shadow price

When the variables and constraints in your model have an associated unit (see Chapter 32), special care is required in interpreting the values returned through the .ShadowPrice suffix. To obtain the shadow price in terms of the units specified in the model, the values of the .ShadowPrice suffix must be scaled as explained in Section 32.5.1.

The property RightHandSideRange

By specifying the RightHandSideRange property you request AIMMS to conduct a first type of sensitivity analysis on this constraint during a SOLVE statement of a linear program. The result of this sensitivity analysis are three parameters defined over the domain of the constraint. Two of these parameters represent the smallest and largest values of an interval over which an individual *right-hand side* (or left-hand side) value can be varied such that the basis remains constant. Consequently, the shadow prices and the reduced costs remain unchanged for variations of a single value within the interval. The third parameter specifies the nominal value for the right-hand side (or left-hand side) of the constraint.

Single sided or ranged constraint

There are three cases we have to consider for the RightHandSideRange property:

- if the constraint is single sided (i.e. $f(x) \leq a$) then the smallest, nominal, and largest value for the constraint side are reported (both when constraint is binding and not binding)
- if the constraint is of range type (i.e. $a \leq f(x) \leq b$) and it is binding at one side, then the smallest, nominal, and largest value for the binding side of the constraint are reported
- if the constraint is of range type (i.e. $a \leq f(x) \leq b$) and it is not binding at neither side, then the lowest upper bound and the highest lower bound are reported.

The values are accessible through the suffices .SmallestRightHandSide, .NominalRightHandSide, and .LargestRightHandSide.

The property ShadowPriceRange

With the ShadowPriceRange property you request AIMMS to conduct a second type of sensitivity analysis on this constraint during a SOLVE statement of a linear program. The result of the sensitivity analysis are two parameters defined over the domain of the variable. The values assigned to the parameters will be the smallest and largest values that the *shadow price* of the constraint can take while holding the objective value constant. The smallest and largest values of the constraint marginals are accessible through the suffices .SmallestShadowPrice and .LargestShadowPrice.

Linear programs only

As with the advanced sensitivity properties of variables (see Section 14.1.2), AIMMS also supports the advanced sensitivity analysis conducted through the properties RightHandSideRange and ShadowPriceRange for linear mathematical programs only. Again, if you want to apply these types of analysis to the final solution of a mixed-integer program, you should fix all integer variables to their final solution (using the .NonVar suffix) and re-solve the resulting mathematical program as a linear program.

Storage and computational costs

Setting any of the properties ShadowPrice, ShadowPriceRange or RightHandSideRange may result in an increase of the memory usage. In addition, the computations required to compute the ShadowPriceRange may considerably increase the total solution time of your mathematical program.

14.2.6 Constraint suffices for global optimization

Suffices for global optimization

AIMMS provides a number of constraint suffices especially for the global optimization solver BARON. They are:

- the .Convex suffix, and
- the .RelaxationOnly suffix.

By providing additional knowledge, that cannot be determined automatically by BARON itself, about the constraints in your model through these suffices, the BARON solver may be able to optimize your global optimization model in a more efficient manner. For more detailed information about the specific capabilities of the BARON solver, you are referred to the BARON website http://www.theoptimizationfirm.com/.

The .Convex suffix

The algorithm of the BARON solver exploits convexity—either identified automatically by BARON itself or explicitly supplied in the model formulation—in order to generate polyhedral cutting planes and relaxations for multivariate non-convex problems. Through the .Convex suffix you can explicitly indicate that a particular constraint is convex if BARON is unable to determine its convexity automatically.

The .RelaxationOnly suffix

Using the .RelaxationOnly suffix, you can considerably enhance the convexification capabilities of BARON. Some nonlinear problem reformulations can often tighten the relaxation process of BARON's branch-and-bound algorithm while making local search considerably more difficult. By assigning a nonzero value to the .RelaxationOnly suffix, you indicate to BARON that the constraint at hand should only be included as a relaxation to the branch-and-bound algorithm, while it should be excluded from the local search.

14.2.7 Chance constraints

Chance constraints

The AIMMS modeling language offers facilities for robust optimization models, including support for *chance constraints* (see also Section 20.3). By setting the Chance property of a constraint, the constraint will become a chance constraint when solving a mathematical program using robust optimization, using the distributions specified for the random parameters contained in its definition. When setting the Chance property, two new attributes will become available, the Probability attribute and the Approximation attribute.

Only for robust optimization

Note that setting the Chance property does not influence the availability and use of the constraint outside the context of robust optimization. In that case, AIMMS will just use the original, deterministic, constraint definition, completely disregarding the uncertainty of the parameters used in the constraint.

The Probability attribute

Through the Probability attribute, you can specify the probability with which you want the constraint to be satisfied for any feasible solution to the robust counterpart of a robust optimization model. Its value must be a numerical expression in the range $[0, 1]$.

The Approximation attribute

When constructing the robust counterpart, AIMMS can use several types of approximations to approximate the chance constraint at hand. You can use the Approximation attribute to specify the type of approximation you want to be applied. The chosen type of approximation may lead to a robust counterpart which is easier or harder to solve (see also Section 20.3). The value of the attribute must be an element expression into the predefined set AllChance-ApproximationTypes.

Chapter 15

Solving Mathematical Programs

Mathematical program components

A *mathematical program* consists of

- a set of unknowns to be determined,
- a collection of constraints that has to be satisfied, and
- an (optional) objective function to be optimized.

The aim of a mathematical program is to find a solution with the aid of a solver such that the objective function assumes an optimal (i.e. minimal or maximal) value.

Different types

Depending on the characteristics of the variables and constraints, a mathematical program in AIMMS can be classified as one of the following.

- If the objective function and all constraints contain only linear expressions (in terms of the variables), and all variables can assume continuous values within their ranges, then the program is a *linear* program.
- If some of the variables in a linear program can assume only integer values, then the program is a *linear mixed integer* program.
- If the objective is a quadratic function in terms of the variables while the constraints are linear, then the program is a *quadratic* program.
- If the objective is neither linear nor quadratic, or some of the constraints contain nonlinear expressions, the program is a *nonlinear* program.

AIMMS will automatically call the appropriate solver to find an (optimal) solution.

This chapter

This chapter first discusses the declaration of a mathematical program, together with auxiliary functions that you can use to specify its set of variables and constraints. The SOLVE execution statement needed to solve any type of mathematical program is presented, and, finally, AIMMS' capabilities to help resolve infeasibilities in your model are discussed.

15.1 MathematicalProgram declaration and attributes

The attributes of mathematical programs are listed in Table 15.1. *Attributes*

Attribute	Value-type	See also page
Objective	*variable-identifier*	
Direction	minimize, maximize	
Variables	*variable-set*	
Constraints	*constraint-set*	
Type	*model-type*	
ViolationPenalty	*reference*	238
Text	*string*	19
Comment	*comment string*	19
Convention	*convention*	534

Table 15.1: MathematicalProgram attributes

The following example illustrates a typical mathematical program. *Example*

```
MathematicalProgram TransportModel {
    Objective   : TransportCost;
    Direction   : minimize;
    Constraints : AllConstraints;
    Variables   : AllVariables;
    Type        : lp;
}
```

It defines the linear program TransportModel, which is built up from all constraints and variables in the model text. The variable TransportCost serves as the objective function to be minimized.

With the Objective attribute you can specify the objective of your mathematical program. Its value must be a reference to a (defined) variable or any other variable expression. When you want to use the objective value in the end-user interface of your model, the Objective attribute must be a variable reference. *The Objective attribute*

If you do not specify an objective, your mathematical program will be solved to find a feasible solution and it will then terminate. *Omitting the objective*

The Direction attribute

In conjunction with an objective you must use the Direction attribute to indicate whether the solver should minimize or maximize the objective. During a SOLVE statement you can override this direction by using a WHERE clause for the direction option.

The Variables attribute

With the Variables attribute you can specify which set of variables are to be included in your mathematical program. Its must be either the predefined set AllVariables or a subset thereof. The set AllVariables is predefined by AIMMS, and it contains the names of all the variables declared in your model. Its contents cannot be changed. If you mathematical program contains an objective, AIMMS will automatically add this to set of generated variables during generation.

Variables as parameters

If the Variables attribute is assigned a subset of the set AllVariables, AIMMS will treat all the variables outside this set as if they were parameters. That is, all occurrences of such variables will not result in the generation of individual variables for the solver, but will be accounted for in the right-hand side of the constraint according to their value during generation.

Compare to NonvarStatus

The Variables attribute performs a similar function as the NonvarStatus attribute or the .NonVar suffix of a variable (see also Section 14.1). The Variables attribute in a mathematical program allows you to quickly change the status of an entire class of variables, while the NonvarStatus (in a variable declaration) gives much finer control at the individual level. As shown below, the latter is very useful to perform model algebra.

The Constraints attribute

With the Constraints attribute you can specify which constraints are part of your mathematical program. Its value must be either the predefined set AllConstraints or a subset thereof. The set AllConstraints contains the names of all declared constraints plus the names of all variables which have a definition attribute. Its contents is computed at compile time, and cannot be changed.

- If you specify the set AllConstraints, AIMMS will generate individual constraints for all declared constraints and variables with a definition.
- If you specify a subset of the set AllConstraints, AIMMS will only generate individual constraints for the declared constraints and defined variables in that subset.

If you mathematical program has an objective which is a defined variable, its definition is automatically added to the set of generated constraints during generation.

Defined variables

Variables with a nonempty definition attribute have a somewhat special status. Namely, for every defined variable AIMMS will not only generate this variable, but will also generate a constraint containing its definition. Therefore, defined variables are contained in both the predefined sets AllVariables and AllConstraints. You can add a defined variable to the variable and constraint set of a mathematical program independently.

- If you omit a defined variable from the variable set of a mathematical program, all occurrences of the variable will be fixed to its current value and accounted for in the right-hand side of all constraints.
- If you omit a defined variable from the constraint set of a mathematical program, the defining constraint will not be generated.

Performing model algebra

By changing the contents of the identifier sets that you have entered at the Variables and Constraints attributes of a mathematical program you can perform a simple form of *model algebra*. That is, you can investigate the effects of adding or removing constraints from within the graphical interface. Furthermore, it allows you to reconfigure your model based on the value of your model data.

Synchronizing variable and constraint sets

When changing the contents of either the variable or the constraint set of a mathematical program, you may find that the contents of the other set also needs some adjustment. For instance, adding a variable to a mathematical program makes no sense if there are no constraints that refer to it. AIMMS offers two special set-valued functions to help you to accomplish this task.

The function VariableConstraints

The function VariableConstraints takes a subset of the predefined set AllVariables as its argument, and returns a subset of the predefined set AllConstraints. The resulting constraint set contains all constraints which use one or more of the variables in the argument set.

The function ConstraintVariables

The function ConstraintVariables performs the opposite task. It takes a subset of the set AllConstraints as its arguments, and returns a subset of the set AllVariables. The resulting variable set contains all variables which are referred to in one or more constraints in the argument set. Also included are all variables referred to in the definitions of other variables inside the set.

Example

Consider the use of the functions VariableConstraints and ConstraintVariables in conjunction with the following declaration of a mathematical program.

```
MathematicalProgram PartialTransportModel {
    Objective   : TransportCost;
    Direction   : minimize;
    Constraints : PartialConstraintSet;
    Variables   : PartialVariableSet;
}
```

Assume that the set PartialVariableSet contains a subset of the variables declared in the model. Further assume that you would like to build up the contents of the set PartialConstraintSet together with the required additions to PartialVariableSet so that the contents of both sets are maximal. This is referred to as their transitive closure. By successively calling the functions VariableConstraints and ConstraintVariables, the following loop computes the transitive closure of the variable and constraint sets.

```
repeat
   PreviousCardinality  := Card( PartialVariableSet );
   PartialConstraintSet := VariableConstraints( PartialVariableSet   );
   PartialVariableSet   := ConstraintVariables( PartialConstraintSet );

   break when Card( PartialVariableSet ) = PreviousCardinality;
endrepeat ;
```

The break occurs when the set PartialVariableSet has not increased in size.

The Type attribute

With the Type attribute of a mathematical program you can prescribe a solution type. When the specified type is not compatible with the generated mathematical program, AIMMS will return an error message. You can override the type during a SOLVE statement using a WHERE clause for the type option. You can use this, for instance, to easily switch between the mip and rmip types.

Available types

A complete list of the mathematical program types available within AIMMS is given in Table 15.2. Most are self-explanatory. When the type rmip is specified, all integer variables are treated as continuous within their bounds. The rmip type is the global version of the Relax attribute associated with individual variables (see also Section 14.1). The types ls and nls can only be selected in the absence of the Objective attribute.

The Convention attribute

You can use the Convention attribute to specify the unit convention that you want to be used for scaling the variables and constraints in your mathematical program. For further details on this issue you are referred to Section 32.8.

The ViolationPenalty attribute

With the ViolationPenalty attribute you can instruct AIMMS to automatically add artificial terms to the constraints of your mathematical program to help resolve and/or track infeasibilities in your mathematical program. Infeasibility analysis and the use of the ViolationPenalty attribute is discussed in full detail in Section 15.4.

15.2 Suffices and callbacks

Suffices

A mathematical program has a number of suffices which can be used for various purposes. Typical examples are:

Type	Description
lp	linear program
ls	linear system
qp	quadratic program
nlp	nonlinear program
nls	nonlinear system
mip	mixed integer program
rmip	relaxed mixed integer program
minlp	mixed integer nonlinear program
rminlp	relaxed mixed integer nonlinear program
qp	quadratic program
miqp	mixed integer quadratic program
qcp	quadratic constraint program
miqcp	mixed integer quadratic constraint program
network	pure network program
mcp	mixed complementarity program
mpcc	mathematical program with complementarity constraint

Table 15.2: Available model types with AIMMS

- To obtain information about the solution process. This information is filled in by the solver at the end of the solution process. These suffixes are presented in Table 15.3.
- To determine when and how to activate a callback procedure. This information can be filled in between solution steps. See also Chapter 16 where an alternative method for callbacks is presented. These suffixes are presented in Table 15.4.
- To get statistics of the generated mathematical program. These statistics are determined when the generated mathematical program is constructed. These suffixes are presented in Table 15.5.

Solver callbacks

After each iteration the external solver calls back to the AIMMS system to offer AIMMS the opportunity to take control. AIMMS, in turn, allows you to execute a procedure which is referred to as a *callback procedure*. Once the callback procedure has finished, the control is returned to the external solver to continue with the next iteration. By including a callback procedure you can perform several tasks such as:

- inspect the current status of the solution process,
- update one or more model parameters, which can be used, for instance, to provide a graphical overview of the solution process,
- retrieve (part of) the current solution, and
- abort the solution process, and

Suffix	Meaning
Objective	Current objective value
Incumbent	Current incumbent value
BestBound	Best bound on objective value
ProgramStatus	Current program status
SolverStatus	Current solver status
Iterations	Current number of iterations
Nodes	Current number of nodes (mip, miqp, and miqcp only)
GenTime	Current generation time in [second]
SolutionTime	Current solution time in [second]
NumberOfBranches	Number of nodes visited by a CP solver
NumberOfFails	Number of leaf nodes without solution in a CP search tree
NumberOfInfeasibilities	Final number of infeasibilities
SumOfInfeasibilities	Final sum of the infeasibilities

Table 15.3: Suffices of a mathematical program filled by the solver

You can nominate any procedure as a callback procedure by assigning its name to the suffix CallbackProcedure of the associated mathematical program as in:

```
TransportModel.CallbackProcedure := 'MyCallbackProcedure' ;
```

Note that values assigned to the suffix CallbackProcedure or any of the other suffices holding the name of a callback procedure, must be elements of the predefined set AllProcedures. Therefore, if you assign a literal procedure name to such a suffix, you should make sure to quote it, as illustrated in the example above.

When activated

Callback procedures under your control may cause a considerable computational overhead, and should only be activated when necessary. To give you control of the frequency of callbacks, AIMMS provide three separate suffices to trigger a callback procedure. Specifically, a callback procedure can be called

- after a specified number of iterations,
- after a specified number of seconds,
- after a change of status of the solution process, or
- at every new incumbent during the solution process of a mixed integer program.

Activated after iterations

With the suffix CallbackIterations you can indicate after how many iterations the callback procedure specified by the CallbackProcedure suffix must be called again. If you specify the number 0 (default), no such callbacks will be made.

Suffix	Meaning
`CallbackProcedure`	Name of callback procedure
`CallbackIterations`	Return to callback after this number of iterations
`CallbackTime`	Name of callback procedure to be called after some elapsed time
`CallbackStatusChange`	Name of callback procedure to be called after a status change
`CallbackIncumbent`	Name of callback procedure to be called for every new incumbent
`CallbackAddCut`	Name of callback procedure to be called to add additional cuts (CPLEX and GUROBI)
`CallbackReturnStatus`	Return status of callback
`CallbackAOA`	Name of AOA callback procedure

Table 15.4: Suffices of a mathematical program stated by the user

Activated after time

With the suffix `CallbackTime` you specify the name of the callback procedure to be called when a certain number of seconds has elapsed. When not specified (the default), no such callbacks are made.

Activated after status change

With the suffix `CallbackStatusChange` you specify the name of the callback procedure to be performed when the status of the solution process changes. When not specified (the default), no such callbacks are made.

Activated after new incumbent

With the suffix `CallbackIncumbent` you specify the name of the callback procedure to be performed when the solver finds a new incumbent during the solution process of a mixed integer program. When not specified (the default), no such callbacks are made.

Watch objective values

During a callback procedure you can access various objective values as they are reported by the solver during a mixed integer program through several suffices of the mathematical program at hand. The following suffices provide information about the objective values:

- through the suffix `Incumbent` you can obtain the objective value of the best integer solution found so far,
- through the suffix `BestBound` you can obtain the best bound on the objective value during the branch-and-bound process, and
- through the suffix `Objective` you can obtain the current objective value reported by the solver at the precise time of the callback.

For mixed integer programs the suffix `Objective` will be meaningless in most cases during the solution process.

Suffix	Meaning
SolverCalls	Total number of applied SOLVE's
NumberOfConstraints	Number of individual constraints
NumberOfVariables	Number of individual variables
NumberOfNonzeros	Number of nonzeros
NumberOfIntegerVariables	Number of individual integer variables
NumberOfIndicatorConstraints	Number of individual constraints with an activating condition
NumberOfSOS1Constraints	Number of individual SOS1 constraints
NumberOfSOS2Constraints	Number of individual SOS2 constraints
NumberOfNonlinearConstraints	Number of individual nonlinear constraints
NumberOfNonlinearVariables	Number of individual nonlinear variables
NumberOfNonlinearNonzeros	Number of nonlinear nonzeros

Table 15.5: Suffices of a mathematical program statistics from AIMMS

Watch intermediate solution values

In a callback procedure you can access the current solution values of the variables in the mathematical program, and assign these to other identifiers in your model. One possible use of this feature is to store multiple feasible integer solutions of a mixed integer linear program.

The procedure RetrieveCurrentVariableValues

For some solvers there may be a considerable overhead involved to retrieve the current variable values during the running solution process. Therefore, AIMMS will only do so when you explicitly call the procedure

RetrieveCurrentVariableValues(*VariableSet*)

With the *VariableSet* argument you can specify the subset of the set AllVariables consisting of all (symbolic) variables for which you want the current values to be retrieved. When you call this procedure outside the context of a solver callback procedure, AIMMS will produce a runtime error.

Adding additional cuts

When you want to add additional cuts during the solution process of a mixed integer program, you should install a callback procedure to generate these constraints using the CallbackAddCut suffix. This procedure is called at every node that has an LP-optimal solution with an objective function value below the current cutoff and is integer infeasible. The procedure allows you to add individual constraints using the GenerateCut(*row*, *local*) function. The *row* argument should always be a scalar reference to an existing constraint name in your model. The *local* argument should be a scalar binary that indicates whether the cut is a local cut (value 1) or a global one (value 0). The *local* argument is an optional argument, and has a default of 1.

Example

Consider a model with the following constraint.

```
Constraint Triangle_Cut {
    IndexDomain  : (i1,i2,i3) | (i1 < i2) and (i2 < i3);
    Definition   : x(i1) + x(i2) + x(i3) - y(i1,i2) - y(i1,i3) - y(i2,i3) <= 1;
}
```

Then the following piece of code, when specified as the procedure body of the CallbackAddCut procedure, will only add those triangle cuts that are violated.

```
    RetrieveCurrentVariableValues(AllVariables);

    for ( (i1,i2,i3) | (i1 < i2) and (i2 < i3) ) do
        if ( x(i1) + x(i2) + x(i3) - y(i1,i2) - y(i1,i3) - y(i2,i3) > 1 + eps ) then
            GenerateCut( Triangle_Cut(i1,i2,i3), 1 );
        endif;
    endfor;
```

Aborting the solution process

When you want to abort the solution process, you can set the suffix CallbackReturnStatus to 'abort' during the execution of your callback procedure, as in:

```
 TransportModel.CallbackReturnStatus := 'abort' ;
```

After aborting the process, AIMMS will retrieve the current solution and set the final solver status to UserInterrupt.

Example

Consider a mathematical program TransportModel which incorporates a callback procedure. The following callback procedure will abort the solution process if the total solution time exceeded 1800 seconds, and if the progress is less than 1% compared to the last nonzero objective function value.

```
if ( TransportModel.SolutionTime > 1800 [second] and PreviousObjective and
     (TransportModel.Objective - PreviousObjective) < 0.01*PreviousObjective )
then
   TransportModel.CallbackReturnStatus := 'abort';
else
   PreviousObjective := TransportModel.Objective;
endif;
```

Solver and program status

Both the ProgramStatus and the SolverStatus suffix take their value in the predefined set AllSolutionStates presented in Table 15.6.

15.3 The SOLVE statement

The SOLVE statement

With the SOLVE statement you can instruct AIMMS to compute the solution of a MathematicalProgram, resulting in the following actions.

- AIMMS determines which solution method(s) are appropriate, and checks whether the specified type is also appropriate.

Program status	Solver status
ProgramNotSolved	SolverNotCalled
Optimal	NormalCompletion
LocallyOptimal	IterationInterrupt
Unbounded	ResourceInterrupt
Infeasible	TerminatedBySolver
LocallyInfeasible	EvaluationErrorLimit
IntermediateInfeasible	Unknown
IntermediateNonOptimal	UserInterrupt
IntegerSolution	PreprocessorError
IntermediateNonInteger	SetupFailure
IntegerInfeasible	SolverFailure
InfeasibleOrUnbounded	InternalSolverError
UnknownError	PostProcessorError
NoSolution	SystemFailure

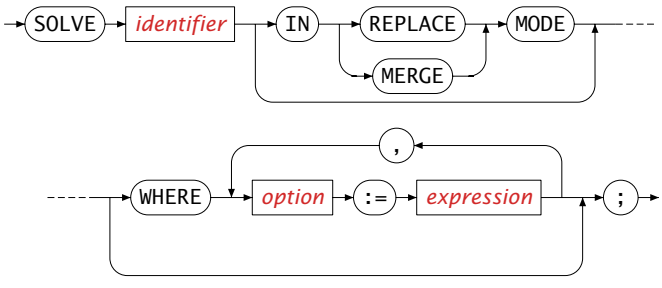
Table 15.6: Mathematical program and solver status

- AIMMS then generates the Jacobian matrix (first derivatives of all the constraints), the bounds on all variables and constraints, and an objective where appropriate.
- AIMMS communicates the problem to an underlying solver that is able to perform the chosen solution method.
- AIMMS finally reads the computed solution back from the solver.

Syntax

In addition to initiating the solution process of a MathematicalProgram, you can also use the SOLVE statement to provide local overrides of particular AIMMS settings that influence the way in which the solution process takes place. The syntax of the SOLVE statement follows.

solve-statement :



Replace and merge mode

You can instruct AIMMS to read back the solution in either *replace* or *merge* mode. If you do not specify a mode, AIMMS assumes replace mode. In replace mode AIMMS will, before reading back the solution of the mathematical

program, remove the values of the variables in the Variables set of the mathematical program for all index tuples except those that are fixed

- because they are not within their current domain (i.e. inactive),
- through the NonvarStatus attribute or the .NonVar suffix of the variable,
- because they are outside the planning interval of a Horizon (see Section 33.3), or
- because their upper and lower bounds are equal.

In merge mode AIMMS will only replace the *individual variable values* involved in the mathematical program. This mode is very useful, for instance, when you are iteratively solving subproblems which correspond to slices of the symbolic variables in your model.

Infeasible and unbounded problems

Whenever the invoked solver finds that a mathematical program is infeasible or unbounded, AIMMS will assign one of the special values na, inf or -inf to the objective variable. For you, this will serve as a reminder of the fact that there is a problem even when you do not check the ProgramStatus and SolverStatus suffices. For all other variables, AIMMS will read back the last values computed by the solver just before returning with infeasibility or unboundedness.

Temporary option settings

Sometimes you may need some temporary option settings during a single SOLVE statement. Instead of having to change the relevant options using the OPTION statement and set them back afterwards, AIMMS also allows you to specify values for options that are used only during the current SOLVE statement. The syntax is similar to that of the OPTION statement.

Also for attributes

Apart from specifying temporary option settings you can also use the WHERE clause to override the type and direction attributes specified in the declaration of the mathematical program, as well as the solver to use for the solution process.

Example

The following SOLVE statement selects 'cplex' as its solver, sets the model type to 'rmip', and sets the CPLEX option LpMethod to 'Barrier'.

```
solve TransportModel in replace mode
    where solver    := 'cplex',
          type      := 'rmip',
          LpMethod  := 'Barrier' ;
```

15.4 Infeasibility analysis

Infeasibility analysis

One of the more daunting tasks in mathematical programming is to find the cause of an infeasible mathematical program. Such infeasibilities may occur

- either when you are developing a new model due to modeling errors,

- or in a complete (and well-tested), model-based, end-user application employed by your customers due to inconsistencies in the model data.

Infeasibilities due to modeling errors

There are several types of modeling errors that you can make during the development of a mathematical program that can lead to hard-to-explain infeasibilities. The most common are:

- simple typing errors, leading, for instance, to a wrong variable being referenced in a constraint,
- a logical flaw in the model formulation, i.e. the formulation of one or more constraints just makes no sense,
- the domain restriction of a constraint is not restrictive enough, i.e. constraints are generated that should not be generated,
- the domain restriction of a variable is wrong, leading to too many or too few terms being generated in constraints referring to such a variable, or
- the restriction in iterative operators (such as `SUM` or `PROD`) in the definition of constraints or defined variables is wrong, leading to too many or too little terms being generated in that particular constraint.

In general, trying to find infeasibilities that occur during model development may force you to generate a constraint listing of your mathematical program and carefully examine the generated constraints in order to find the modeling error.

Infeasibilities due to data inconsistencies

Even when the formulation of a mathematical program is internally consistent, and shipped as an end-user application to your customers, infeasibilities may occur due to inconsistencies in the model data. The most common data errors are:

- inconsistencies in the structural data defining the topology of a model, e.g. in a network model a demand node may have been added for which no incoming arcs have been specified, or
- inconsistencies in the quantitive model data, e.g. to total demand exceeds the total supply.

While most data inconsistencies may be detected by methodically checking the consistency all input data prior to actually solving the mathematical program (for example, by using `Assertions`, see also Section 25.2), it is often hard to cover all possible data inconsistencies.

Adding excess variables

A commonly used approach to try and deal with infeasibilities, is to add explicit excess variables to all or some constraints in a model, along with a penalty term in the objective that will keep all excess variables equal to 0 if the model is feasible. If this procedure is executed properly, the *modified* mathematical program will always be feasible, while the *original* mathematical program is feasible if and only if the excess variables are all equal to 0. In the

case of an infeasibility, an examination of the excess variables may provide useful information about the cause of infeasibility.

Laborious procedure

While adding excess variables to your model may certainly help you to resolve any infeasibilities, the process of manually adding these excess variables to a mathematical program is laborious and error-prone:

- you have to add the declarations of the excess variables for all (or some) constraints in your model,
- the selected constraints have to be modified to include these excess variables, and
- the objective has to be modified to include the excess-related penalty terms.

In addition, adding excess variables may considerably increase the size of the generated matrix, so you may want to write supporting code to exclude the excess variables from your mathematical program unless you encounter an infeasibility.

15.4.1 Adding infeasibility analysis to your model

The ViolationPenalty attribute

To ease the manual process described above, AIMMS offers support to *automatically* extend your mathematical program with excess variables during the generation of the matrix for the solver. You enable this feature through the ViolationPenalty attribute of a MathematicalProgram declaration. The value of the ViolationPenalty attribute must be either a

- 1-dimensional parameter with index domain AllVariablesConstraints, or
- 2-dimensional parameter defined over AllVariablesConstraints and AllViolationTypes.

The predefined set AllVariablesConstraints is a subset of the set AllIdentifiers and contains the names of all the variables and constraints in your model. Through one of these two types of parameters you can specify for which variables and constraints in your mathematical program AIMMS must generate excess variables, as well as the penalty coefficient of these excess variables in the modified objective.

The set AllViolationTypes

The predefined set AllViolationTypes is a fixed set containing the three types of possible violations for which AIMMS can generate excess variables. The elements in the set AllViolationTypes are

- Lower: generate excess variables for the violation of a lower bound,
- Upper: generate excess variables for the violation of an upper bound, and
- Definition: generate excess variables for the violation of the equality between a defined variable and its definition.

Interpretation of ViolationPenalty attribute

If a parameter you entered in the ViolationPenalty attribute contains no data, AIMMS will generate the mathematical program without any generated excess variables. If you specify a 2-dimensional parameter which is not empty, all values must be nonnegative or assume the special value ZERO (see also Section 6.1.1), and AIMMS will interpret its contents as follows.

Penalty for objective variable

The modified objective will include the original objective, unless a value of ZERO has been assigned to Definition violation type for the original objective variable. AIMMS will treat any other penalty value than ZERO assigned to the objective variable as 1.0! Note that by including the original objective the penalized mathematical program may become unbounded.

Penalty for constraints

AIMMS will add nonnegative excess variables for the violation of a (finite) lower and/or upper bound of every constraint for which a penalty value other than 0.0 has been specified for the Lower and/or Upper violation type, respectively. If a bound is infinite, no corresponding excess variable will be generated. A penalty term will be added to the modified objective consisting of the product of the specified (nonnegative) penalty coefficient times the excess variable associated with the constraint, unless a penalty of ZERO has been specified in which case the corresponding term will not be added to the modified objective.

Penalty for variables

AIMMS will add nonnegative excess variables for the violation of a (finite) lower and/or upper bound of every variable for which a penalty value other than 0.0 has been specified for the Lower and/or Upper violation type, respectively. If a bound is infinite, no corresponding excess variable will be generated. A penalty term will be added to the modified objective consisting of the product of the specified (nonnegative) penalty coefficient times the excess variable associated with the variable, unless a penalty of ZERO has been specified in which case the corresponding term will not be added to the modified objective. The effect of using Lower and/or Upper violations is that the variable can assume values outside their bounds throughout the mathematical program.

Penalty for variable definitions

AIMMS will add nonnegative excess variables for the violation of the equality between a defined variable and its definition for every defined variable for which a penalty value other than 0.0 has been specified for the Definition violation type. A penalty term will be added to the modified objective consisting of the product of the specified (nonnegative) penalty times the excess variable(s) associated with the constraint expressing the equality, unless a penalty of ZERO has been specified in which case the corresponding term(s) will not be added to the modified objective.

Definition versus lower/upper violations

You can both use the Lower and/or Upper violation types and Definition violation type to compensate for a violation between the value of the defined variable and its definition. However, when you use the Definition violation type, the value of the variable will remain within its specified bounds throughout the mathematical program. It is up to you to decide which violation type suits your needs best for a particular defined variable.

Interpretation of 1-dimensional parameter

If you specify a 1-dimensional parameter for the ViolationPenalty attribute, AIMMS will interpret this parameter as if it were a 2-dimensional parameter, with the same value for all three violation types Lower, Upper and Definition.

15.4.2 Inspecting your model for infeasibilities

Finding violations

After you have let AIMMS extend your model with excess variables to find an infeasibility, you must inspect the variables and constraints in your model to find the violations. AIMMS allows you to do this through the use of two suffices, the .Violation suffix and the .DefinitionViolation suffix.

The .Violation suffix...

The .Violation suffix denotes the amount by which a variable or constraint violates its lower or upper bound. If you have specified a nonzero violation penalty for the Upper violation type, the .Violation suffix can assume positive values, while it can assume negative values whenever you have specified a nonzero violation penalty for the Lower violation type.

... for variables

For variables the .Violation suffix denotes the amount by which the variable violates its

- upper bound (if the suffix assumes a positive value), or
- lower bound (if the suffix assumes a negative value).

... for constraints

For constraints the .Violation suffix denotes the amount by which the constraint violates its

- upper bound (if the suffix assumes a positive value),
- lower bound (if the suffix assumes a negative value, for ranged constraints).

If the constraint is an equality constraint, the .Violation suffix denotes the (positive or negative) amount by which the left hand side differs from the (constant) right hand side.

The .DefinitionViolation suffix

With the .DefinitionViolation suffix, you can locate violations in the definitions of defined variables for which you have specified a positive penalty for the Definition violation type. The value of the suffix denotes the (positive or negative) amount by which the defined variable differs from its definition. Note that a defined variable may violate both its bounds and its definition, depending on the type of allowed violations you have specified.

Locating violations

To locate violations in a model which was extended by AIMMS with excess variables, you may use the Card function to locate variables and constraints with nonzero .Violations suffices. The following example shows how to proceed, where v is assumed to be an index in AllVariables.

```
for ( v | Card(v, 'Violation'}) ) do

    ! Take any action that you want to perform on this violated variable

endfor;
```

15.4.3 Application to goal programming

Goal programming ...

In goal programming a distinction is made between *hard constraints* that cannot be violated and *soft constraints*, which represent goals or targets one would like to achieve. The objective function in goal programming is to minimize the weighted sum of deviations from the goals set by the soft constraints.

... interpreted as violations

In AIMMS, goal programming can be easily implemented using the ViolationPenalty attribute of a mathematical program, without the need to modify the formulation of all soft constraints. For each soft constraint in your goal programming model, you can assign the appropriate weight to the ViolationPenalty attribute to penalize deviations from the set target for that constraint.

Inspecting deviations

Through the .Violation suffix of constraints and variables you can inspect the deviations from the goals of the soft constraints in your goal programming model.

Chapter 16

Implementing Advanced Algorithms for Mathematical Programs

Control over the solution process

The SOLVE statement discussed in Section 15.3 offers a convenient way to execute all necessary steps to generate and solve a single instance of a mathematical program in one simple statement. For most applications, this level of control over the individual steps required to execute the generation and solution process is sufficient. However, for advanced applications, you may need a finer-grained level of control, e.g. to

- work with multiple, differing, instances of a single symbolic mathematical program,
- manipulate the individual rows and columns and the coefficient matrix of a mathematical program instance, for example to efficiently implement a column generation scheme,
- work with a repository of solutions associated with a mathematical program instance, for instance as a means to store multiple starting solutions or, within a solver callback, to setup and update a collection of incumbents of a mixed integer model, or
- start multiple solver sessions for a mathematical program instance, either locally or remotely.

This chapter

This chapter describes a library of procedures that offers you fine-grained control over the generation, manipulation and solution of a mathematical program instance, and allows you to manage a collection of solutions and solver sessions associated with such mathematical program instances. As you will see later on, the SOLVE statement can be completely expressed in terms of the procedures in this library.

16.1 Introduction to the GMP library

Introduction

With every MathematicalProgram declared as part of your model, the GMP library allows you to associate

- one or more *Generated Math Program instances* (GMPs),

and with each GMP

- a conceptual matrix of coefficients that can be manipulated,
- a repository of initial, intermediate or final solutions, and
- a pool of local or remote solver sessions.

Figure 16.1 illustrates the interrelationship between symbolic mathematical programs and the concepts of the GMP library, as well as the main properties that can be associated with each of them.

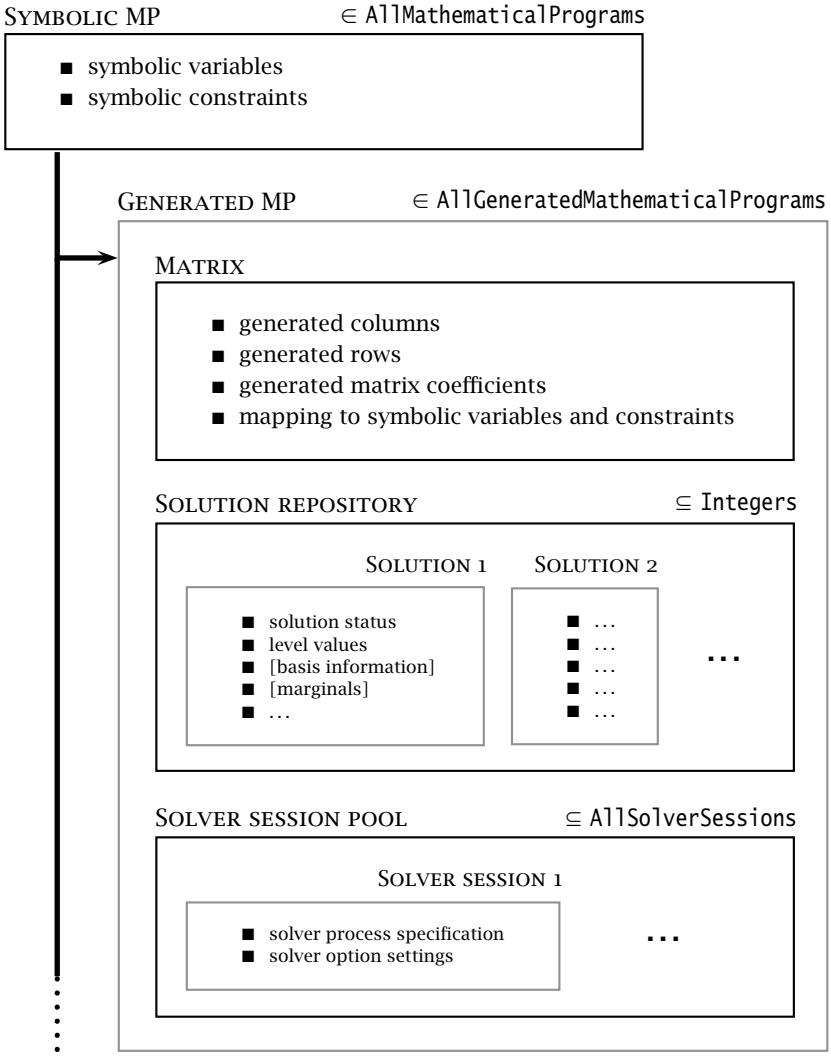


Figure 16.1: Concepts associated with a GMP

Generated mathematical program instances

For every MathematicalProgram declaration in your model, modifications in the index sets and input data referenced in constraints and variable definitions may give rise to completely different instances of the coefficient matrix when the mathematical program at hand is being generated.

An example: indexed instances

An illustrative example of such differing instances occurs when the constraints and variables of a symbolic mathematical program are indexed over a subset of some other superset. If you let the subset contain a single element of the superset, the generated instances will be completely different for each element of the superset. The effect of changing the contents of the subset in this manner, would almost compare to having an indexed MathematicalProgram *declaration* (which AIMMS does not support). In the worked example of Section 16.13.1 you will see, however, how you can obtain an indexed collection of generated mathematical program *instances* using the GMP library.

Need for multiple instances

With the standard SOLVE statement (see Section 15.3) you only have access to a single generated mathematical program instance for every symbolic mathematical program, namely the instance associated with the last call to the SOLVE statement for that particular mathematical program. This effectively eliminates the capability to efficiently implement an algorithm which requires the interaction between two or more generated instances of the same symbolic mathematical program. For this reason, the GMP library allows you to maintain and work with a collection of generated mathematical program instances simultaneously.

Matrix manipulation

The GMP library also allows you to manipulate the rows, columns and coefficients of the matrix of a mathematical program instance once it has been generated. If the number of modifications is relatively small, manipulating the matrix directly will save a considerable amount of time compared to letting AIMMS completely regenerate the matrix again through the standard SOLVE statement. You can use matrix manipulation, for instance

- to quickly add columns, and adapt the existing rows of the matrix accordingly, in a column generation scheme, or
- to dynamically add cuts to a mixed integer linear program.

Keeping multiple solutions

With the standard SOLVE statement, you only have access to a single solution of a mathematical program, namely the one stored in the symbolic variables and constraints that make up the mathematical program. There are, however, many situations where it would be convenient to have access to a repository of solutions. A solution repository can be used, for instance

- to store a collection of starting solutions for a NLP or MINLP problem. Solving the problem, in either a serial or parallel manner, with each of these starting solutions may help you find a better solution than by simply solving the problem with only a single starting solution.

- during the solution process of a mixed integer program, if you are interested in other integer solutions than the final solution returned by the solver. You can use the solution repository to store a fixed size collection of the best incumbent solutions returned by the solver during the solution process.

Solution repository

The GMP library comes with a solution repository for each generated mathematical program instance, and offers a number of functions to easily transfer a solution from and to either

- the data of the variables and constraints that make up the associated mathematical program in your model, or
- any solver session (explained below) associated with the generated mathematical program instance.

In fact, in the GMP library there is no direct solution/starting point transfer between a solver and the model, but such transfer always takes place through the solution repository.

Solver session pool

The final concept that is part of the GMP library is that of solver sessions. In principle, the GMP library is prepared to allow a generated mathematical program instance to keep a pool of associated solver sessions, each possibly set up with a different solver, or with different solver settings, and to be run either locally or remotely.

When useful

Using multiple solver session it becomes possible, for example, to let the same (or another) solver with different solver settings solve a mixed integer program instance in parallel, and pass tighter bound information found by one solver session to the other sessions by means of a callback implemented in your model.

GMP namespace

To prevent naming conflicts, all functions and procedure in the GMP library are member of the predefined `GMP` namespace. The `GMP` namespace is further partitioned into the namespaces

- `GMP::Instance`,
- `GMP::Row`,
- `GMP::Column`,
- `GMP::Coefficient`,
- `GMP::Event`,
- `GMP::QuadraticCoefficient`,
- `GMP::Solution`,
- `GMP::SolverSession`,
- `GMP::Stochastic`,
- `GMP::Robust`,
- `GMP::Benders`,

- `GMP::Linearization`, and
- `GMP::ProgressWindow`.

In the following sections we will discuss the procedures and functions contained in each of these namespaces.

Return values

When using the GMP library, it may be particularly important to check for any kind of error conditions that can occur. To help you catch such errors, the procedures and functions in the `GMP` namespace either return

- a 1 when successful, or 0 otherwise (for procedures), or
- a non-empty element in one of the GMP-related predefined sets when successful, or the empty element otherwise (for functions).

Note that, for the sake of brevity, most of the examples in this chapter do not perform error checking of any kind.

16.2 Managing generated mathematical program instances

Managing math program instances

The procedures and functions of the `GMP::Instance` namespace are listed in Table 16.1 and take care of the creation and management of generated mathematical program instances. Mathematical program instances also provide access to the solution repository and solver sessions associated with the instance.

Creation of mathematical program instances

New mathematical program instances can be created by calling

- the `SOLVE` statement,
- the `GMP::Instance::Generate` function,
- the `GMP::Instance::GenerateRobustCounterpart` function,
- the `GMP::Instance::GenerateStochasticProgram` function,
- the `GMP::Instance::Copy` function,
- the `GMP::Instance::CreateDual` function,
- the `GMP::Instance::CreateFeasibility` function,
- the `GMP::Instance::CreatePresolved` function,
- the `GMP::Instance::CreateMasterMIP` function,
- the `GMP::Stochastic::CreateBendersRootproblem` function,
- the `GMP::Stochastic::BendersFindFeasibilityReference` function, or
- the `GMP::Stochastic::BendersFindReference` function.

Generate(*MP*, *name*)→AllGeneratedMathematicalPrograms Copy(*GMP*, *name*)→AllGeneratedMathematicalPrograms Rename(*GMP*, *name*) Delete(*GMP*)	
GenerateRobustCounterpart(*MP*, *UncertainParameters*, *UncertaintyConstraints* [, *Name*])→AllGeneratedMathematicalPrograms	
GenerateStochasticProgram(*MP*, *StochasticParameters*, *StochasticVariables*, *Scenarios*, *ScenarioProbability*, *ScenarioTreeMap*, *RootScenario* [, *GenerationMode*][, *Name*])→AllGeneratedMathematicalPrograms	
CreateMasterMIP(*GMP*, *name*)→AllGeneratedMathematicalPrograms FixColumns(*GMP1*, *GMP2*, *solNr*, *varSet*) AddIntegerEliminationRows(*GMP*, *solNr*, *elimNo*) DeleteIntegerEliminationRows(*GMP*, *elimNo*)	
CreateDual(*GMP*, *name*)→AllGeneratedMathematicalPrograms CreateFeasibility(*GMP*[, *name*][, *useMinMax*])→AllGeneratedMathematicalPrograms CreatePresolved(*GMP*, *name*)→AllGeneratedMathematicalPrograms	
GetSymbolicMathematicalProgram(*GMP*)→AllMathematicalPrograms GetNumberOfRows(*GMP*) GetNumberOfColumns(*GMP*) GetNumberOfNonzeros(*GMP*)	
GetDirection(*GMP*)→AllMathematicalProgrammingDirections SetDirection(*GMP*, *dir*)	
GetOptionValue(*GMP*, *OptionName*) SetOptionValue(*GMP*, *OptionName*, *Value*)	
CreateProgressCategory(*GMP*[, *Name*])→AllProgressCategories	
GetMathematicalProgrammingType(*GMP*)→AllMathematicalProgrammingTypes SetMathematicalProgrammingType(*GMP*, *type*)	
GetSolver(*GMP*)→AllSolvers	SetSolver(*GMP*, *solver*)
SetCallbackAddCut(*GMP*, *CB*) SetCallbackBranch(*GMP*, *CB*) SetCallbackIncumbent(*GMP*, *CB*) SetCallbackHeuristic(*GMP*, *CB*) SetCallbackTime(*GMP*, *CB*)	SetCallbackAddLazyConstraint(*GMP*, *CB*) SetCallbackCandidate(*GMP*, *CB*) SetCallbackStatusChange(*GMP*, *CB*) SetCallbackIterations(*GMP*, *CB*, *nrIters*)
SetIterationLimit(*GMP*, *nrIters*) SetTimeLimit(*GMP*, *nrSeconds*)	SetMemoryLimit(*GMP*, *nrMB*) SetCutoff(*GMP*, *value*)
Solve(*GMP*) FindApproximatelyFeasibleSolution(*GMP*, *sol1*, *sol2*, *nrIter*[, *maxIter*][, *feasTol*] [, *moveTol*][, *imprTol*][, *maxTime*][, *useSum*][, *augIter*][, *useBest*])	
GetObjective(*GMP*) GetBestBound(*GMP*) GetMemoryUsed(*GMP*) MemoryStatistics(*GMPSet*, *OutputFileName*[, *optional-arguments . . .*])	
GetColumnNumbers(*GMP*, *varSet*)→Integers GetRowNumbers(*GMP*, *conSet*)→Integers GetObjectiveColumnNumber(*GMP*)→Integers GetObjectiveRowNumber(*GMP*)→Integers	
DeleteMultiObjectives(*GMP*)	
CreateSolverSession(*GMP*[, *Name*][, *Solver*])→AllSolverSessions DeleteSolverSession(*solverSession*)	

Table 16.1: Procedures and functions in GMP::Instance namespace

All mathematical program instances created through each of these calls, are uniquely represented by elements in the predefined set AllGeneratedMathematicalPrograms. For the functions in the GMP::Instance namespace creating GMPs you can explicitly specify the name of the associated set element to be created. When calling the SOLVE statement, AIMMS will generate an element with the same name as the MathematicalProgram at hand. When the name of the element to be created is already contained in the set AllGeneratedMathematicalPrograms, the mathematical program instance associated with the existing element will be completely replaced by the newly created mathematical program instance.

Special math programming types

Stochastic programming and the use of the function GenerateStochasticProgram is discussed in Section 19.4. Robust optimization and the use of the function GenerateRobustCounterpart is explained in Section 20.5. The functionality of the CreateDual function is explained in more detail in Section 16.2.1. The function CreateMasterMIP is used by the AIMMS Outer Approximation solver, which is discussed in full detail in Chapter 18. Presolving of mathematical programs is discussed in Section 17.1.

Deleting and renaming instances

Through the procedures GMP::Instance::Delete and GMP::Instance::Rename you can delete and rename mathematical program instances and their associated elements in the set AllGeneratedMathematicalPrograms. If you rename a mathematical program instance to a name that already exists in the set AllGeneratedMathematicalPrograms, the associated mathematical program instance will be deleted prior to renaming.

CLEANDEPENDENTS statement

Note that also the CLEANDEPENDENTS statement may remove mathematical program instances from memory when it affects any constraint or variable referenced by that instance.

Retrieving and setting basic properties

Through the functions

- GMP::Instance::GetSymbolicMathematicalProgram,
- GMP::Instance::GetNumberOfRows,
- GMP::Instance::GetNumberOfColumns,
- GMP::Instance::GetNumberOfNonzeros,
- GMP::Instance::GetDirection, and
- GMP::Instance::GetMathematicalProgrammingType

you can retrieve the current value of some basic properties of a mathematical program instance. The number of rows, columns and nonzeros can be changed by manipulating the matrix of the mathematical program instance (see also Section 16.3). You can use the functions

- GMP::Instance::SetDirection, and
- GMP::Instance::SetMathematicalProgrammingType

to modify the optimization direction and mathematical programming type. The type of a mathematical program must be a member of the set `MathematicalProgrammingTypes` (see also section 15.1) The direction associated with a mathematical program is either

- `'maximize'`,
- `'minimize'`, or
- `'none'`.

The direction `'none'` is the instruction to the solver to find a feasible solution.

Installing callbacks

For each mathematical program instance, you can set up to six callback functions that will be called by any solver session associated with the mathematical program instance at hand. Through the following procedures you can install or uninstall a callback function for a mathematical program instance.

- `GMP::Instance::SetCallbackAddCut`
- `GMP::Instance::SetCallbackAddLazyConstraint`
- `GMP::Instance::SetCallbackBranch`
- `GMP::Instance::SetCallbackCandidate`
- `GMP::Instance::SetCallbackIncumbent`
- `GMP::Instance::SetCallbackStatusChange`
- `GMP::Instance::SetCallbackHeuristic`
- `GMP::Instance::SetCallbackIterations`
- `GMP::Instance::SetCallbackTime`

Each of these procedures expects an element of the set `AllProcedures`, or an empty element `''` to uninstall the callback.

Callback procedures

Callback procedures for each type of callback should be declared as follows:

`AnExampleCallback`(*solverSession*)

where the *solverSession* argument should be a scalar input element parameter into the set `AllSolverSessions`. Callback procedures should have a return value of

- 0, if you want the solver session to stop, or
- 1, if you want the solver session to continue.

As discussed before, each solver session can be uniquely associated with a single mathematical program instance. You can find this instance by calling the function `GMP::SolverSession::GetInstance` (see also Section 16.5), and, within the callback procedure, use this instance to get access to its associated properties.

Example

The following example implements a callback procedure for the incumbent callback. The callback procedure finds the associated mathematical program instance, and stores all incumbents reported by the solver into the next solution of the solution repository.

```
Procedure IncumbentCallBack {
    Arguments  : solvSess;
    Body       : {
        theGMP := GMP::SolverSession::GetInstance( solvSess );
        GMP::Solution::RetrieveFromSolverSession( solvSess, solutionNumber(theGMP) );
        solutionNumber(theGMP) += 1;

        return 1;   ! continue solving
    }
}
```

Note that the callback procedure uses the GMP::Solution::RetrieveFromSolver-Session function (discussed in Section 16.4) to retrieve the solution from the solver.

Solving mathematical program instances

In contrast to the SOLVE statement, the philosophy behind the GMP library is to break down the optimization functionality in AIMMS to a level which offers optimum support for implementing advanced algorithms around a MathematicalProgram in your model. One of the consequences of this philosophy is that the solution is never directly transferred between the symbolic variables and constraints and the solver, but is intermediately stored in a solution repository. Therefore, solving a MathematicalProgram using the GMP library breaks down into the following basic steps:

1. generate a mathematical program instance for the MathematicalProgram,
2. create a solver session for the mathematical program instance,
3. transfer the initial point from the model to the solution repository,
4. transfer the initial point from the solution repository to the solver session,
5. let the solver session solve the problem,
6. transfer the final solution from the solver session to the solution repository, and
7. transfer the final solution from the solution repository to the model.

Solving the instance directly

For your convenience, however, the GMP library contains a procedure

- GMP::Instance::Solve

which, given a generated mathematical program instance, takes care of all intermediate steps (i.e. steps 2-7) necessary to solve the mathematical program instance. In case you need access to the solution in the solution repository after calling the GMP::Instance::Solve call, you should notice that the GMP::Instance::Solve procedure (as well as the SOLVE statement) performs all of its solution transfer through the fixed solution number 1 in the solution repository.

Emulating the SOLVE statement

The following AIMMS code provides an emulation of the SOLVE statement in terms of GMP::Instance functions.

```
! Generate an instance of the mathematical program MPid and add
! the element 'MPid' to the set AllGeneratedMathematicalPrograms.
! This element is returned into the element parameter genGMP.
genGMP := GMP::Instance::Generate(MPid, FormatString("%e", MPid));

! Actually solve the problem using the solve procedure for an
! instance (which communicates through solution number 1).
GMP::Instance::Solve(genGMP);
```

Multistart support

The function FindApproximatelyFeasibleSolution is used by the AIMMS multistart algorithm (see Section 17.2) to compute an approximately feasible solution for an NLP problem. The algorithm used by this function to find the approximately feasible solution is described in [Ch04].

Creating solver sessions

For each generated mathematical program instance, you can explicitly create and delete one or more solver sessions using the following functions:

- GMP::Instance::CreateSolverSession, and
- GMP::Instance::DeleteSolverSession.

Once created, you can use the solver session to solve the generated mathematical program

- in a blocking manner by calling the GMP::SolverSession::Execute function, or
- in a non-blocking manner by calling the GMP::SolverSession::AsynchronousExecute function.

Prior to calling the GMP::SolverSession::Execute or GMP::SolverSession::AsynchronousExecute functions, you should call the function GMP::Solution::SendToSolverSession to initialize the solver session with a solution stored in the solution repository. Using an explicit solver session allows you, for instance, to solve an NLP problem with several initial solutions stored in the solution repository.

Multiple sessions allowed

AIMMS allows you to create multiple solver sessions per mathematical program instance, and solve them in parallel. You can solve multiple mathematical program instances in parallel, by calling the function GMP::SolverSession::AsynchronousExecute multiple times. The function starts a separate thread of execution to solve the math program instance asynchronously, and returns immediately. To solve multiple mathematical program instances in parallel, your computer should have multiple processors or a multi-core processor.

Deleting solver sessions

Once the function GMP::SolverSession::Execute or GMP::SolverSession::AsynchronousExecute has been called, the internal solver representation of the mathematical program instance will be created. The solver representation will only be deleted—and its associated resources freed—when the corresponding solver session has been deleted by calling the function GMP::Instance::DeleteSolverSession.

Implementing the procedure GMP::Instance::Solve

The GMP:Instance::Solve procedure discussed previously can be emulated using solver sessions, as illustrated in the equivalent code below.

```
! Create a solver session for genMP, which will create an element
! in the set AllSolverSessions, and assign the newly created element
! to the element parameter session.
session := GMP::Instance::CreateSolverSession(genMP);

! Copy the initial solution from the variables in AIMMS to
! solution number 1 of the generated mathematical program.
GMP::Solution::RetrieveFromModel(genMP,1);

! Send the solution stored in solution 1 to the solver session
GMP::Solution::SendToSolverSession(session, 1);

! Call the solver session to actually solve the problem.
GMP::SolverSession::Execute(session);

! Copy the solution from the solver session into solution 1.
GMP::Solution::RetrieveFromSolverSession(session, 1);

! Store this solution in the AIMMS variables and constraints.
GMP::Solution::SendToModel(genMP, 1);
```

Setting default solver session limits

You can use the following procedures to set various default limits that apply to all solver sessions created through GMP::Instance::CreateSolverSession.

- GMP::Instance::SetIterationLimit
- GMP::Instance::SetMemoryLimit
- GMP::Instance::SetTimeLimit
- GMP::Instance::SetCutoff

Setting GMP-specific options

For every *GMP* you can override the default project options using the function GMP::Instance::SetOptionValue. You can also set options for a specific solver session associated with a *GMP* through the function GMP::SolverSession::SetOptionValue. In turn, option values set for a specific solver session override the option values for the associated *GMP*.

Setting the default solver

Similarly, you can get and set the default solver that will be used by all solver sessions created through GMP::Instance::CreateSolverSession.

- GMP::Instance::GetSolver
- GMP::Instance::SetSolver

Outer approximation support

Through the functions

- GMP::Instance::CreateMasterMIP
- GMP::Instance::FixColumns
- GMP::Instance::AddIntegerEliminationRows
- GMP::Instance::DeleteIntegerEliminationRows

the GMP library offers support for solving mixed integer nonlinear (MINLP) problems using a white box outer approximation approach. The AIMMS Outer Approximation solver is discussed in full detail in Chapter 18.

16.2.1 Dealing with degeneracy and non-uniqueness

Background

When solving a mathematical program, some practical difficulties may arise when the optimal solution of the underlying model is either degenerate and/or not unique (i.e. there are multiple optimal solutions). These difficulties may concern both the primal and dual solution (i.e. the shadow prices).

Problems with degeneracy

In the case of degeneracy (see also Section 4.2 of the AIMMS Modeling Guide for an explanation), the solution status of one or more variables is "basic at bound". In the presence of degeneracy, shadow prices are no longer unique, and their interpretation is therefore ambiguous. As a result, if the shadow prices have an economic interpretation in the application, the particular shadow prices found by the solver cannot be presented to the end-user in a meaningful and reliable fashion.

Problems with multiple solutions

In the case of multiple solutions, the situation is even worse. There are multiple optimal bases, and the associated shadow prices differ between these bases (just as with degeneracy). In addition, the solution presented to the end-user is no longer unique, which may raise questions by the end-user as to why a particular solution is presented.

Degeneracy and multiple solutions

Both degeneracy and multiple solutions can occur at the same time, having their combined effect on the non-uniqueness of both the primal and the dual solution (the optimal shadow prices). The following two paragraphs present possible solutions to deal with multiple primal and dual solutions.

Towards a unique primal solution

One way to deal with multiple solutions is to find a new and second objective function specifically designed to deal with eliminating the multiplicity of solutions. This might be accomplished, for instance, by adding new sets of variables and constraints to cap some aspect of the primal model, and the maximum cap could then be minimized. Or perhaps a straightforward modification of the original objective function could become the second auxiliary objective. It is important to note that this second objective function is opti-

mized only after the first objective function is fixed at its previous optimal value and has been added as a constraint.

Implementing primal uniqueness

Using the functionality provided by the GMP library, constructing a second objective function for a mathematical program is a straightforward task:

- generate and solve the original mathematical program,
- use the matrix manipulations procedures discussed in Section 16.3 to create a new objective and fix the original one in the associated mathematical program instance,
- resolve the modified mathematical program instance.

Towards a unique dual solution

In the presence of primal degeneracy and/or multiple primal solutions, it is impossible to influence the selection of shadow prices, as this decision is made by the solver. To give the control back to you as a model developer, the only sensible step is to go directly to the dual formulation, and work with the model expressed in terms of shadow prices. It is then possible to construct a second auxiliary objective function designed to produce economically meaningful shadow prices. Again, it is important to note that this second objective function is optimized only after the original objective function is fixed at the optimal objective function value of the primal model, and has been added as a constraint.

Creating a dual mathematical program instance

To support the procedure for reaching dual uniqueness, the GMP library contains the function

- `GMP::Instance::CreateDual`

which creates the dual mathematical program instance associated with a given primal mathematical program instance.

Standard dual formulation

For a mathematical program of the form

Minimize:

$$\sum_i c_i x_i$$

Subject to:

$$\sum_i A_{ij} x_i \geq b_j \qquad \forall j$$

$$x_i \geq 0 \qquad \forall i$$

the dual mathematical program can be formulated as follows

Maximize:

$$\sum_j b_j \lambda_j$$

Subject to:

$$\sum_j A_{ij}\lambda_j \le c_i \qquad \forall i$$

$$\lambda_j \ge 0 \qquad \forall j$$

where the λ_j represent the shadow prices of the constraints of the primal formulation.

Sign changes

If the primal formulation contains nonpositive or free variables, or contains $\le$ or equality constraints, a number of simple substitution will bring the formulation back into the standard form above, after which the above dual formulation can be used directly. The resulting changes to the dual formulation are as follows:

- a nonpositive variable x_i corresponds to a dual $\ge$ constraint,
- a free variable x_i corresponds to a dual equality constraint,
- a $\le$ constraint corresponds to a nonpositive dual variable λ_j, and
- an equality constraint corresponds to a free dual variable λ_j.

Bounded variables and ranged constraints

However, such simple transformation are not possible anymore if the primal model contains:

- bounded variables, i.e. $l_i \le x_i \le u_i$, or
- ranged constraints, i.e. $d_i \le \sum_i A_{ij}x_i \le b_j$.

In these cases, additional constraints (implicitly) have to be added as follows to satisfy the above standard formulation:

- $x_i \ge l_i$ whenever $l_i \ne 0, -\infty$,
- $x_i \le u_i$ whenever $u_i \ne 0, \infty$, and
- $\sum_i A_{ij}x_i \ge d_j$.

In the generated dual mathematical program, such implicit constraint additions in the primal formulation will lead to the explicit introduction of additional variables in the dual formulation. Such variable additions to the dual formulation are taken care of by AIMMS automatically, but will have consequences when you want to manipulate the matrix of the dual mathematical program instance, as discussed in Section 16.3.7.

Implementing dual uniqueness

Using the function `GMP::Instance::CreateDual`, it is relatively straightforward to implement the procedure outlined above to reach dual uniqueness:

- generate and solve the original mathematical program,
- generate a dual mathematical program instance from the primal mathematical program instance,
- use the matrix manipulations procedures discussed in Section 16.3 to create a new dual objective and fix the original dual objective in the newly created dual mathematical program instance,
- solve the modified dual mathematical program instance.

16.3 Matrix manipulation procedures

Matrix manipulation

The matrix manipulation procedures in the GMP library allow you to implement efficient algorithms for generated mathematical program instances which require only slight modifications of the matrix associated with the mathematical program instance during successive runs. These procedures operate directly on the coefficient matrix underlying the mathematical program, and thus avoid the constraint-generation process normally initiated by the SOLVE statement after input data has been modified.

This section

Prior to discussing the individual matrix manipulation procedures, the following section will provide some motivation when and when not to use matrix manipulation.

16.3.1 When to use matrix manipulation

When to use matrix manipulation

Even though AIMMS offers a library of matrix manipulation procedures, you should not use them blindly. As explained below, it is important to distinguish between manual and automatic input data changes inside an AIMMS application. Your decision whether or not to use the matrix manipulation procedures described in this section, should depend on this distinction.

Manual data input ...

Consider an end-user of an AIMMS application who, after having looked at the results of a mathematical program, wants to make changes in the input data and then look again at the new solution of the mathematical program. The effect of the data changes on the input to the solver cannot be predicted in advance. Even a single data change could lead to multiple changes in the input to the solver, and could also cause a change in the number of constraints and variables inside the particular mathematical program.

... requires structure recognition

As a result, AIMMS has to determine whether or not the structure of the underlying mathematical program has changed. Only then can AIMMS decide whether the value of existing coefficients can be overwritten, or whether a new and structurally different data set has to be provided to the solver. This structure recognition step is time consuming, and cannot be avoided in the absence of any further information concerning the changes in input data.

Automatic data input ...

Whenever input data are changed inside an AIMMS procedure, their effect on the input to the solver can usually be determined in advance. This effect may be nontrivial, in which case it is not worth the effort to establish the consequences. Rather, letting AIMMS perform the required structure recognition step through the regular SOLVE statement before passing new information to

the solver seems to be a better remedy. There are several instances, however, in which the effect of data changes on the solver input data is easy to determine.

... may reflect particular structure

Consider, for instance, automatic data changes that have a one-to-one correspondence with values in the underlying mathematical program. In these instances, the incidence of variables in constraints is not modified, and only the replacement values of some coefficients need to be supplied to the particular solver. Other examples include automatic data changes that could create new values for particular variable-constraint combinations, or that could even cause new constraints or variables to be added to the input of the solver. In all these instances, the exact effects on the input of the solver can easily be determined in advance, and there is no need to let AIMMS perform of the computationally intensive structure recognition step of the SOLVE statement before passing new information to the solver.

Restrictions on usage

The above effects of data input modifications on the input to the solver are straightforward to implement with linear and quadratic mathematical programs, because the underlying data structures are matrices with rows, columns and nonzero elements. The input data structures for nonlinear mathematical programs are essentially nonlinear expressions. Modifications of the type discussed in the previous paragraph are not easily passed onto these nonlinear data structures. For this reason, the efficient updating of solver input has been confined to

- linear and quadratic constraints, and
- coefficients of nonlinear constraints with respect to variables that only occur linearly in that constraint.

Regeneration of nonlinear constraints

Whenever the input data of a nonlinear expression in a nonlinear constraint has changed, it is not possible anymore to change the nonlinear expression used by the solver directly to reflect the data change. You can still request AIMMS to regenerate the entire row, which will then use the updated inputs. You should note, however, that any modifications to the linear part of the regenerated constraint are lost after the constraint has been regenerated.

Scalar arguments only

All matrix procedures listed in Tables 16.2–16.5 and most procedures listed in Table 16.13 have scalar-valued arguments. The *row* argument should always be

- a scalar reference to an existing constraint name in your model, or
- a row number which is an integer in the range $\{0..m-1\}$ whereby m is the number of rows.

The *column* argument should always be

- a scalar reference to an existing variable name in your model, or

- a column number which is an integer in the range $\{0..n-1\}$ whereby n is the number of columns.

Modifying a group of columns or rows

For most matrix procedures listed in Tables 16.2–16.5, that can be used to modify a generated mathematical program, there also exists a "multi" variant which can be applied to a group of columns of rows, belonging to one variable or constraint respectively. These procedures are listed in Section 16.3.6.

Mathematical program instance required

Before you can apply any of the procedures of Tables 16.2–16.5, you must first create a mathematical program instance using any of the functions for this purpose discussed in Section 16.2. Either of these methods will set up the initial row-column matrix required by the matrix manipulation procedures. Also, any row or column referenced in the matrix manipulation procedures must either have been generated during the initial generation step, or must have been generated later on by a call to the procedures `GMP::Row::Add`, or `GMP::Column::Add`, respectively.

16.3.2 Coefficient modification procedures

Coefficient modification procedures

The procedures and functions of the `GMP::Coefficient` namespace are listed in Table 16.2 and take care of the modification of coefficients in the matrix and objective of a generated mathematical program instance.

`Get`(*GMP*, *row*, *column*)
`Set`(*GMP*, *row*, *column*, *value*)
`GetQuadratic`(*GMP*, *column1*, *column2*)
`SetQuadratic`(*GMP*, *column1*, *column2*, *value*)

Table 16.2: Procedures and functions in `GMP::Coefficient` namespace

Modifying coefficients

You can instruct AIMMS to modify any particular coefficient in a matrix by specifying the corresponding row and column (in AIMMS notation), together with the new value of that coefficient, as arguments of the procedure `GMP::Coefficient::Set`. This procedure can also be used when a value for the coefficient does not exist prior to calling the procedure.

Quadratic coefficients

For quadratic mathematical programs, you can modify the quadratic objective coefficients by applying the function `GMP::Coefficient::SetQuadratic` to the objective row. For every two columns x_1 and x_2 you can specify the modified coefficient c_{12} if $c_{12}x_1x_2$ is to be part of the quadratic objective.

16.3.3 Quadratic coefficient modification procedures

Quadratic coefficient modification procedures

The procedures and functions of the GMP::QuadraticCoefficient namespace are listed in Table 16.3 and take care of the modification of coefficients of quadratic rows in the matrix other than the objective of a generated mathematical program instance.

Get(*GMP*, *row*, *column1*, *column2*)
Set(*GMP*, *row*, *column1*, *column2*, *value*)

Table 16.3: Procedures and functions in GMP::QuadraticCoefficient namespace

Modifying coefficients

You can instruct AIMMS to modify any particular quadratic coefficient in a matrix by specifying the corresponding row and columns (in AIMMS notation), together with the new value of that coefficient, as arguments of the procedure GMP::QuadraticCoefficient::Set. This procedure can also be used when a value for the quadratic coefficient does not exist prior to calling the procedure.

16.3.4 Row modification procedures

Row modification procedures

The procedures and functions of the GMP::Row namespace are listed in Table 16.4 and take care of the modification of properties of existing rows and the creation of new rows.

Row types

The row type refers to one of the four possibilities

- '<=',
- '=',
- '>=', and
- 'ranged'

You are free to change this type for each row. Deactivating and subsequently reactivating a row are instructions to the solver to ignore the row as part of the underlying mathematical program and then reconsider the row again as an active row.

Row generation

When you add a new row to a matrix using GMP::Row::Add, the newly added row will initially only have any zero coefficients, regardless of whether the corresponding AIMMS constraint had a definition or not. Through the procedure GMP::Row::Generate you can tell AIMMS to discard the current contents of a row in the matrix, and insert the coefficients as they follow from the definition of the corresponding constraint in your model.

Add(*GMP*, *row*) Delete(*GMP*, *row*) Activate(*GMP*, *row*) Deactivate(*GMP*, *row*)
Generate(*GMP*, *row*)
GetLeftHandSide(*GMP*, *row*) SetLeftHandSide(*GMP*, *row*, *value*) GetRightHandSide(*GMP*, *row*) SetRightHandSide(*GMP*, *row*, *value*)
GetType(*GMP*, *row*) → AllRowTypes SetType(*GMP*, *row*, *type*) GetStatus(*GMP*, *row*) → AllRowColumnStatuses
DeleteIndicatorCondition(*GMP*, *row*) GetIndicatorColumn(*GMP*, *row*) GetIndicatorCondition(*GMP*, *row*) SetIndicatorCondition(*GMP*, *row*, *column*, *value*)
GetConvex(*GMP*, *row*) GetRelaxationOnly(*GMP*, *row*) SetConvex(*GMP*, *row*, *value*) SetRelaxationOnly(*GMP*, *row*, *value*)
SetPoolType(*GMP*, *row*, *value*[, *mode*])

Table 16.4: Procedures and functions in GMP::Row namespace

Indicator conditions

When you are using the CPLEX, GUROBI or ODH-CPLEX solver, you can declaratively specify indicator constraints through the IndicatorConstraint property of a constraint declaration (see Section 14.2.4). You can also set and delete indicator constraints programmatically for a given *GMP* using the functions GMP::Row::SetIndicatorCondition and GMP::Row::DeleteIndicatorCondition

Lazy and cut pool constraints

When you are using the CPLEX, GUROBI or ODH-CPLEX solver, you can declaratively specify constraints to be part of a pool of lazy constraints or cuts through the IncludeInLazyConstraintPool and IncludeInCutPool properties of a constraint declaration respectively (see Section 14.2.4). You can also specify lazy and cut pool constraints programmatically for a given *GMP* using the function GMP::Row::SetPoolType.

Convex and relaxation-only constraints

Through the .Convex and .RelaxationOnly suffices of constraints you can set special constraint properties for the BARON global optimization solver (see also Section 14.2.6). For a given *GMP* you can also set these constraint properties programmatically using the GMP::Row::SetConvex and GMP::Row::SetRelaxationOnly functions.

16.3.5 Column modification procedures

The procedures and functions of the GMP::Column namespace are listed in Table 16.5 and take care of the modification of properties of existing columns and the creation of new columns.

Add(*GMP*, *column*)
Delete(*GMP*, *column*)
Freeze(*GMP*, *column*, *value*)
Unfreeze(*GMP*, *column*)
GetLowerBound(*GMP*, *column*)
SetLowerBound(*GMP*, *column*, *value*)
GetUpperBound(*GMP*, *column*)
SetUpperBound(*GMP*, *column*, *value*)
GetType(*GMP*, *column*) → AllColumnTypes
SetType(*GMP*, *column*, *type*)
GetStatus(*GMP*, *column*) → AllRowColumnStatuses
SetDecomposition(*GMP*, *column*, *value*)
SetAsObjective(*GMP*, *column*)
SetAsMultiObjective(*GMP*, *column*, *priority*, *weight*)

Table 16.5: Procedures and functions in GMP::Column namespace

Column types

The column type refers to one of the three possibilities

- 'integer',
- 'continuous', and
- 'semi-continuous'.

You are free to specify a different type for each column. For newly added columns, AIMMS will (initially) use the lower bound, upper bound and column type as specified in the declaration of the (symbolic) variable associated with the added column. Freezing a column and subsequently unfreezing it are instructions to the solver to fix the corresponding variable to its current value, and then free it again by letting it vary between its bounds.

Changing the objective column

If you want to implement the procedures for reaching primal or dual uniqueness as described in Section 16.2.1, you can use the procedure

- GMP::Column::SetAsObjective

to change the objective function used by either the primal or dual mathematical program instance that you want to solve for a second time. Notice that the defining constraint for this variable should be

- part of the original mathematical program formulation for which AIMMS has generated a mathematical program instance, or
- added later on to the primal or dual generated mathematical program instance using the GMP::Row::Add procedure, where the row definition is generated by AIMMS through the GMP::Row::Generate procedure or constructed explicitly through several calls to the GMP::Coefficient::Set procedure.

16.3.6 More efficient modification procedures

If you want to change the data of many columns or rows belonging to some variable or constraint then it is more efficient to use the multi variant of a modification procedure. The available multi procedures are listed in Table 16.6.

Coefficient::SetMulti(*GMP*, *binding*, *row*, *column*, *value*)
Column::AddMulti(*GMP*, *binding*, *column*)
Column::DeleteMulti(*GMP*, *binding*, *column*)
Column::FreezeMulti(*GMP*, *binding*, *column*, *value*)
Column::UnfreezeMulti(*GMP*, *binding*, *column*)
Column::SetLowerBoundMulti(*GMP*, *binding*, *column*, *value*)
Column::SetUpperBoundMulti(*GMP*, *binding*, *column*, *value*)
Column::SetTypeMulti(*GMP*, *binding*, *column*, *type*)
Column::SetDecompositionMulti(*GMP*, *binding*, *column*, *value*)
Row::AddMulti(*GMP*, *binding*, *row*)
Row::DeleteMulti(*GMP*, *binding*, *row*)
Row::GenerateMulti(*GMP*, *binding*, *row*)
Row::ActivateMulti(*GMP*, *binding*, *row*)
Row::DeactivateMulti(*GMP*, *binding*, *row*)
Row::SetRightHandSideMulti(*GMP*, *binding*, *row*, *value*)
Row::SetTypeMulti(*GMP*, *binding*, *row*, *type*)
Row::SetPoolTypeMulti(*GMP*, *binding*, *row*, *value*, *mode*)

Table 16.6: Multi procedures in GMP namespace

Binding argument

All procedures in Table 16.6 contain an index binding argument. The index binding argument specifies which columns or rows will be modified. If the procedure contains a value argument then the size of this vector is defined by the index binding argument. Further information on index binding can be found in Chapter 9.

16.3.7 Modifying an extended math program instance

Extended math program instances

To use the matrix manipulation routines of the GMP library, you must be able to associate every row and column of the matrix of the math program instance you want to manipulate with a symbolic constraint or variable within your model. However, some routines in the GMP library generate rows and columns that cannot be directly associated with specific symbolic constraints and variables in your model. Examples of such routines are:

- the `GMP::Instance::CreateDual` procedure, which may generate additional variables in the dual formulation for bounded variables and ranged constraints in the primal formulation (see also Section 16.2.1),
- the `GMP::Linearization::Add` and `GMP::Linearization::AddSingle` procedures, which add linearizations of nonlinear constraints to a specific math program instance (see also Section 16.11), and
- the `GMP::Instance::AddIntegerEliminationRows` procedure.

The rows and columns generated by these procedures can, however, be *indirectly* associated with symbolic constraints, variables or mathematical programs, as will be explained below.

Extended suffices

To support the use of the matrix manipulation routines in conjunction with rows and columns generated by AIMMS that can only be indirectly associated with symbolic identifers in the model, AIMMS provides the following suffices which allow you to do so:

- `.ExtendedVariable`, and
- `.ExtendedConstraint`.

These suffices are supported for `Variables`, `Constraints` and `MathematicalPrograms`. They behave like variables and constraints, which implies that it is possible to refer to the `.ReducedCost` and `.ShadowPrice` suffices of these extended suffices to get hold of their sensitivity information.

Suffix dimensions

Each of the suffices listed above has one additional dimension compared to the dimension of the original identifier, over the predefined set `AllGMPExtensions`. For example, assuming that `ae` is an index into the set `AllGMPExtensions`,

- if `z(i,j)` is a variable or constrain, the `.ExtendedVariable` suffix will have indices `z.ExtendedVariable(ae,i,j)`,
- if `mp` is a mathematical program, the `.ExtendedConstraint` suffix will have indices `mp.ExtendedConstraint(ae)`.

Each of the procedures listed above, will add elements to the set `AllGMPExtensions` as necessary. The names of the precise elements added to the set is explained below in more detail.

Suffices generated by CreateDual

The procedure GMP::Instance::CreateDual will add the following elements to the set AllGMPExtensions:

- DualObjective, DualDefinition, DualUpperBound, DualLowerBound.

In addition, it will generate the following extended variables and constraints

- For the mathematical program mp at hand
  - the variable mp.ExtendedVariable('DualDefinition'),
  - the constraint mp.ExtendedConstraint('DualObjective').
- For every ranged constraint c(i)
  - the constraint c.ExtendedConstraint('DualLowerBound',i),
  - the constraint c.ExtendedConstraint('DualUpperBound' i).
- For every bounded variable x(i) in $[l_i, u_i]$
  - the constraint x.ExtendedConstraint('DualLowerBound',i) (if $l_i \neq 0, -\infty$),
  - the constraint x.ExtendedConstraint('DualUpperBound' i) (if $u_i \neq 0, \infty$).

Modifying the dual math program

Using the matrix manipulation procedures, you can modify the matrix or objective associated with a dual mathematical program instance created by calling the procedure GMP::Instance::CreateDual. Below you will find how you can access the rows and columns of a dual mathematical program instance created by AIMMS.

Row and column names

For each procedure in the GMP::Coefficient, GMP::Row and GMP::Column namespaces you must refer to a scalar constraint and/or variable reference from your symbolic model. For the dual formulation, you must

- use the symbolic primal constraint name, to refer to the dual shadow price variable associated with that constraint in the dual mathematical program instance, and
- use the symbolic primal variable name, to refer to the dual constraint associated with that variable in the dual mathematical program instance.

In other words, when modifying matrix coefficients, rows or columns the role of the symbolic constraints and variables is interchanged.

Implicitly added variables and constraints

You can refer to the implicitly added variables and constraints in the procedures of the GMP::Coefficient, GMP::Row and GMP::Column namespaces through the .ExtendedVariable and .ExtendedConstraint suffices described above. After solving the dual math program, AIMMS will store the dual solution in the suffices .ExtendedVariable.ReducedCost and .ExtendedConstraint.ShadowPrice, respectively.

Extended suffices for linearization

By calling the procedures GMP::Linearization:Add or GMP::Linearization::AddSingle, AIMMS will add the linearization for a single nonlinear constraint instance, or for all nonlinear constraints from a set of nonlinear constraints to a given math program instance. When doing so, AIMMS will add an element Linearization*k* (where *k* is a counter) to the set AllGMPExtensions, and will create for each nonlinear constraint c(i)

- a constraint c.ExtendedConstraint('Linearization*k*',i), and
- a variable c.ExtendedVariable('Linearization*k*',i) if deviations from the constraint are permitted (see also Section 16.11).

Elimination constraints and variables

By calling the procedure GMP::Instance::AddIntegerEliminationRows, AIMMS will add one or more constraints and variables to a math program instance, which will eliminate the current integer solution from the math program instance. When called, AIMMS will add elements of the form

- Elimination*k*,
- EliminationLowerBound*k*, and
- EliminationUpperBound*k*

to the set AllGMPExtensions. In addition, AIMMS will add

- a constraint mp.ExtendedConstraint('Linearization*k*') to exclude current solution for all binary variables from the math program mp at hand, and
- for every integer variable c(i) with a level value between its bounds the variables and constraints
  - c.ExtendedVariable('Elimination*k*',i),
  - c.ExtendedVariable('EliminationLowerBound*k*',i),
  - c.ExtendedVariable('EliminationUpperBound*k*',i),
  - c.ExtendedConstraint('Elimination*k*',i),
  - c.ExtendedConstraint('EliminationLowerBound*k*',i), and
  - c.ExtendedConstraint('EliminationUpperBound*k*',i).

16.4 Managing the solution repository

The solution repository

The GMP library maintains a solution repository for every generated mathematical program instance. You can use this repository, for instance, to store

- a number of starting solutions for a NLP problem to be solved successively,
- a number of incumbent solutions as reported by a MIP solver, or
- let a solver store multiple solutions.

If you are using solver sessions to initiate a solver, you must explicitly transfer the initial, intermediate or final solutions between the model, the solution repository and the solver session. As discussed in Section 16.2, the function GMP::Instance::Solve performs these necessary solution transfer steps for you, and uses the fixed solution number 1 for all of its communication.

Some solvers are capable of finding multiple solutions instead of at most one. Examples of such solvers are BARON, CPLEX and GUROBI. When such a solver finds multiple solutions, these solutions are stored in the solution repository from number 1 on upwards. The control mechanism to let solvers find multiple solutions is solver specific:

- `BARON 19`: For more information see the **Help** file for option `Number of best solutions` in option category `Specific solvers – BARON 19 – General`.
- `CPLEX 12.9`: For more information see the **Help** file for option `Do populate` in option category `Specific solvers – CPLEX 12.9 – MIP solution pool`.
- `GUROBI 8.1`: For more information see the **Help** file for option `Pool search mode` in option category `Specific solvers – GUROBI 8.1 – Solution pool`.

Solution repository functions

The procedures and functions of the `GMP::Solution` namespace are listed in Table 16.7. Through these functions you can

- transfer a solution between the solution repository on the one side and the symbolic model or the solver on the other side,
- obtain and set solution properties of a solution in the repository, or
- perform a feasibility check on a solution in the repository.

Solution contents

Each solution in the repository is represented by a solution vector containing all relevant solution data, such as

- solution status,
- level values,
- basis information,
- marginals, and
- other relevant requested sensitivity information.

Solution numbering

Each generated mathematical program instance has its own associated solution repository. Each solution in the repository is represented by an integer solution number. Through the function `GMP::Solution::GetSolutionsSet` you can retrieve a subset of the predefined set `Integers` containing the set of all solution numbers that are currently in use for the given mathematical program instance.

Solution transfer to the model

Through the functions

- `GMP::Solution::RetrieveFromModel`,
- `GMP::Solution::SendToModel`, and
- `GMP::Solution::SendToModelSelection`

you can (re-)initialize a solution with the values currently contained in the symbolic model, and vice versa. The function `SendToModelSelection` allows you to only initialize a part of the model identifiers and suffices with a solution of from the solution repository.

Copy(*GMP*, *fromSol*, *toSol*)
Move(*GMP*, *fromSol*, *toSol*)
Delete(*GMP*, *solNo*)
DeleteAll(*GMP*)
GetSolutionsSet(*GMP*)→Integers
Count(*GMP*)
RetrieveFromModel(*GMP*, *SolNr*)
SendToModel(*GMP*, *SolNr*)
SendToModelSelection(*GMP*, *SolNr*, *Identifiers*, *Suffices*)
RetrieveFromSolverSession(*solverSession*, *SolNr*)
SendToSolverSession(*solverSession*, *SolNr*)
GetObjective(*GMP*, *SolNr*)
GetBestBound(*GMP*, *SolNr*)
GetProgramStatus(*GMP*, *SolNr*)→AllSolutionStatus
GetSolverStatus(*GMP*, *SolNr*)→AllSolutionStatus
GetIterationsUsed(*GMP*, *SolNr*)
GetMemoryUsed(*GMP*, *SolNr*)
GetTimeUsed(*GMP*, *SolNr*)
SetObjective(*GMP*, *SolNr*, *value*)
SetProgramStatus(*GMP*, *SolNr*, *PrStatus*)
SetSolverStatus(*GMP*, *SolNr*, *PrStatus*)
SetIterationCount(*GMP*, *SolNr*, *IterCnt*)
GetColumnValue(*GMP*, *SolNr*, *column*)
SetColumnValue(*GMP*, *SolNr*, *column*, *value*)
GetRowValue(*GMP*, *SolNr*, *row*)
SetRowValue(*GMP*, *SolNr*, *row*, *value*)
Check(*GMP*, *SolNr*, *NumInf*, *SumInf*, *MaxInf*[, *skipObj*])
IsInteger(*GMP*, *SolNr*)
IsPrimalDegenerated(*GMP*, *SolNr*)
IsDualDegenerated(*GMP*, *SolNr*)
GetFirstOrderDerivative(*GMP*, *SolNr*, *row*, *column*)
ConstraintListing(*GMP*, *SolNr*, *name*)

Table 16.7: Procedures and functions in GMP::Solution namespace

Solution transfer to a solver session

Through the functions

- GMP::Solution::RetrieveFromSolverSession, and
- GMP::Solution::SendToSolverSession

you can set a solution in the repository equal to a solution reported by a given solver session, or initialize the (initial) solution of a solver session with a solution stored in the repository. Notice that these functions do not have a *GMP* argument. Because each solver session is uniquely associated with a single mathematical program instance, AIMMS is able to determine the correct solution repository.

Computing first order derivatives

Using the function GMP::Solution::GetFirstOrderDerivative, you can compute, for the given solution, first order derivative of a particular row in a mathematical program with respect to a given variable. You can use such a function, for instance, to implement a sequential linear programming approach for nonlinear programs, as outlined in Section 16.13.5.

16.5 Using solver sessions

Using solver sessions

The procedures and functions of the GMP::SolverSession namespace are listed in Table 16.8. Solver sessions are created implicitly by AIMMS or explicitly by calling the procedure GMP::Instance::CreateSolverSession.

Execute(*solverSession*) AsynchronousExecute(*solverSession*)
ExecutionStatus(*solverSession*)→AllExecutionStatuses Interrupt(*solverSession*) WaitForCompletion(*Objects*) WaitForSingleCompletion(*Objects*)→AllSolverSessionCompletionObjects
CreateProgressCategory(*solverSession*[, *Name*][, *Size*])
GetOptionValue(*solverSession*, *optionName*) SetOptionValue(*solverSession*, *optionName*, *value*)
GetInstance(*solverSession*)→AllGeneratedMathematicalPrograms GetSolver(*solverSession*)→AllSolvers GetCallbackInterruptStatus(*solverSession*)→AllSolverInterrupts
GetIterationsUsed(*solverSession*) GetMemoryUsed(*solverSession*) GetTimeUsed(*solverSession*)
GetBestBound(*solverSession*) GetCandidateObjective(*solverSession*) GetObjective(*solverSession*)
GetProgramStatus(*solverSession*)→AllSolutionStates GetSolverStatus(*solverSession*)→AllSolutionStates SetSolverStatus(*solverSession*)
GenerateCut(*solverSession*, *row*[, *local*][, *purgeable*]) RejectIncumbent(*solverSession*)
GetNodeNumber(*solverSession*) GetNodeObjective(*solverSession*) GetNodesLeft(*solverSession*) GetNodesUsed(*solverSession*) GetNumberOfBranchNodes(*solverSession*)
Transfer(*solverSession*, *GMP*)

Table 16.8: Procedures and functions in GMP::SolverSession namespace

Solving a mathematical program instance

By calling the GMP::SolverSession::Execute procedure, the given solver session will take care of solving the associated mathematical program instance in a blocking manner, i.e. the function will not return until the solver has completed the solution process. This function is called implicitly by the GMP::Instance::Solve function or by the SOLVE statement.

Asynchronous solve

Alternatively, you can solve a mathematical program instance in an non-blocking manner by using the function GMP::SolverSession::AsynchronousExecute. Rather than waiting for the solution process to complete, this function will dispatch the solution process to a separate thread of execution, and return immediately. This allows multiple mathematical program instances to be solved in parallel, assuming your computer has multiple processors or a multi-core processor. Note that requests for a synchronous solve through the SOLVE statement will fail if a AIMMS is still executing an asynchronous solution process.

Session synchronization

To allow your application to synchronize its execution when multiple solver sessions are executed asynchronously, AIMMS offers the following synchronization procedures

- GMP::SolverSession::Interrupt,
- GMP::SolverSession::ExecutionStatus,
- GMP::SolverSession::WaitForCompletion, and
- GMP::SolverSession::WaitForSingleCompletion.

Through the GMP::SolverSession::Interrupt function you can request AIMMS to interrupt a solver session that is executing (asynchronously). You can call the function GMP::SolverSession::ExecutionStatus to check the status of a given solver session.

Waiting for multiple completions

Using the function GMP::SolverSession::WaitForCompletion you can halt the main AIMMS thread of execution to wait until the entire set of solver sessions passed as an argument to the function have completed. You can use this function, for instance, to end the solution phase of your model, prior to moving on to the post-processing phase of your model.

... and for single completion

In addition, AIMMS offers a function GMP::SolverSession::WaitForSingleCompletion which returns as soon as a single solver session from the given set of solver sessions has completed its execution. The return value of the function is the completed solver session that caused the function to return. You can use WaitForSingleCompletion, for instance, to asynchronously solve the next mathematical program instance from a queue of mathematical program instances waiting to be solved.

No solution transfer

Note that neither GMP::SolverSession::Execute and GMP::SolverSession::AsynchronousExecute will copy the initial solution into the solver, or copy the final solution back into solution repository or model identifiers. When you use these functions you always have to explicitly call functions from the GMP::Solution namespace to accomplish these tasks.

Support for callbacks

When callbacks for the mathematical program instance associated with a solver session have been set (see also Section 16.2), AIMMS will make sure that the specified callback procedures in your model will be called whenever appropriate. If you have specified a single callback procedure for multiple callback reasons, you can call the procedure

- GMP::SolverSession::GetCallbackInterruptStatus

to retrieve the reason why your callback procedure was called. The result is an element in the predeclared set AllSolverInterrupts which contains the elements

- Candidate,
- Incumbent,
- AddCut,
- Iterations,
- Heuristic,
- StatusChange, and
- Finished.

When the solver session has not yet been called, the status is '' (empty element). During a callback, you can call the function

- GMP::SolverSession::GetInstance

if you need the mathematical program instance associated with the given solver session, and you can retrieve the current objective values using the functions

- GMP::SolverSession::GetBestBound, and
- GMP::SolverSession::GetObjective.

Synchronous nested solves allowed

During any callback you are allowed to generate and solve other mathematical program instances *in a synchronous manner*. You can use such nested solves, for instance, for finding a heuristic solution during a Heuristic callback. Once you have found a heuristic solution, you can pass it onto the running solver session using the function GMP::Solution::SendToSolverSession. Note that this functionality is currently only supported by CPLEX and GUROBI.

No asynchronous solves

During a callback AIMMS does not allow you to call the function GMP::SolverSession::AsynchronousExecute to solve another mathematical program instance in an asynchronous manner. However, AIMMS offers a special class of synchronization objects called *events*, which allow you to notify the main thread of execution that some event has occurred and act accordingly. When set during a callback, the main thread of execution may respond, for instance, by generating a mathematical program instance based on solver data set by the callback, and solve that mathematical program instance in an asynchronous manner. Events are discussed in full detail in Section 16.6.

Adding cuts

During an AddCut callback you may use the procedure GMP::SolverSession::GenerateCut to generate a local or global cut. A local cut will only be added to the current node in the solution process and all its descendant nodes, while a global cut will remain to exist for all nodes onwards. The result of the procedure will be the temporary addition of row to the matrix, as if GMP::Row::Generate had been called. Note that this functionality is currently only supported by CPLEX, GUROBI and ODH-CPLEX.

Rejecting incumbents

During a Candidate callback you can reject the incumbent found by the solver by calling the procedure GMP::SolverSession::RejectIncumbent. Note that this functionality is currently only supported by CPLEX.

Setting options

You can set options for a specific solver session associated through the function GMP::SolverSession::SetOptionValue. These option values override the option values for the associated *GMP*, set through GMP::Instance::SetOptionValue, which in their turn override the project options.

16.6 Synchronization events

Events for synchronization

To allow for more advanced thread synchronization during parallel solves, AIMMS offers synchronization objects called *events*, which can be manipulated using the function listed in Table 16.9.

Create(*name*)→AllGMPEvents
Delete(*event*)
Set(*event*)
Reset(*event*)

Table 16.9: Procedures and functions in GMP::Event namespace

Creating events

Through the function GMP::Event::Create you can create a new event, while the function GMP::Event::Delete deletes existing events. Using the function GMP::Event::Set you can notify AIMMS that an event has occurred. The function GMP::Event::Reset resets the event.

Waiting for events

Events are elements of the predefined set AllGMPEvents, which, along with the set AllSolverSessions, is a subset of the predefined set AllSolverSessionCompletionObjects. As both the functions

- GMP::SolverSession::WaitForCompletion, and
- GMP::SolverSession::WaitForSingleCompletion

expect a subset of the set AllSolverSessionCompletionObjects as their arguments, these functions can be used to wait for both solver session completion and the occurrence of events.

Using events

You can use events, for example, to notify the main thread of execution in your model that you want a new mathematical program instance to be generated and solved asynchronously based on input data provided by a solver callback. AIMMS does not allow asynchronous solves to be started from within a callback itself.

16.7 Multi-objective optimization

Multi-objective optimization

Multi-objective optimization deals with mathematical optimization problems involving more than one objective function that have to be optimized simultaneously. Optimal decisions need to take into account the presence of trade-offs between two or more conflicting objectives. For example, minimizing the travelling distance while minimizing the travelling time (which might conflict if the shortest route is not the fastest). AIMMS allows you to define multiple objectives for linear models only. Multi-objective optimization in AIMMS is currently only supported by CPLEX and GUROBI.

Blended or lexicographic objective

You can define a mixture of blended and lexicographic (or hierarchical) objectives. A blended objective consists of the linear combination of several objectives with given weights. A lexicographic objective assumes that the objectives can be ranked in order of importance. A solution is considered lexicographically better than another solution if it is better in the first objective where they differ (following the order). For a minimization problem, an optimal solution is one that is lexicographically minimal.

The procedure GMP::Column::SetAsMultiObjective

Currently, the only way to specify a multi-objective optimization model is by using the GMP library. The procedure GMP::Column::SetAsMultiObjective can be used to mark a variable as an objective used for multi-objective optimization. Typically, the definition of such a variable defines the objective but the variable can also be used in an equality constraint which then defines the objective.

Example

Consider the following declarations

```
Variable TotalDistance {
    Definition :  sum( (i,j), Distance(i,j) * X(i,j) );
}
Variable TotalTime {
    Definition :  sum( (i,j), TravelTime(i,j) * X(i,j) );
}
```

Here X(i,j) is a (binary) variable indicating whether the road between i and j is used. The variables TotalDistance and TotalTime can be specified as objectives in a multi-objective optimization model using:

```
GMP::Column::SetAsMultiObjective( genMP, TotalDistance, 2, 1.0, 0, 0.1 );
GMP::Column::SetAsMultiObjective( genMP, TotalTime, 1, 1.0, 0, 0.0 );
```

In this example, AIMMS will only pass the coefficients of the variable X as the multi-objective coefficients to the solver, so Distance(i,j) for the first objective and TravelTime(i,j) for the second objective. (In other words, the multi-objective variables TotalDistance and TotalTime will be substituted by their definitions.) After solving the model, the objectives can be deleted by calling the procedure GMP::Instance::DeleteMultiObjectives.

Priority and weight

The priority of the objective can be specified using the third argument of the procedure GMP::Column::SetAsMultiObjective. Its fourth argument defines the weight by which the objective coefficients are multiplied when forming a blended objective, i.e., if multiple objectives have the same priority. The last two (optional) arguments specify the absolute and relative tolerance respectively, which define the amount by which a solution may deviate from the optimal value for the objective.

Mathematical program objective

In case of multi-objective optimization, the variable specified in the Objective attribute of the mathematical program will be treated as a normal variable, that is, it will not be used as one of the multi-objectives.

16.8 Supporting functions for stochastic programs

Supporting functions for stochastic models

The stochastic Benders algorithm (see Section 19.4.2) is implemented in AIMMS as a combination of a system module that can be included into your model, and a number of supporting functions in the GMP::Stochastic namespace of the

BendersFindFeasibilityReference(*GMP*, *stage*, *scenario*) →AllGeneratedMathematicalPrograms BendersFindReference(*GMP*, *stage*, *scenario*) →AllGeneratedMathematicalPrograms CreateBendersRootproblem(*GMP*[, *name*]) →AllGeneratedMathematicalPrograms UpdateBendersSubproblem(*GMP*, *solution*)
AddBendersFeasibilityCut(*GMP*, *solution*, *cutNo*) AddBendersOptimalityCut(*GMP*, *solution*, *cutNo*)
MergeSolution(*GMP*, *solution1*, *solution2*[, *updObj*]) GetRepresentativeScenario(*GMP*, *stage*, *scenario*)→AllStochasticScenarios GetObjectiveBound(*GMP*, *solution*) GetRelativeWeight(*GMP*, *stage*, *scenario*)

Table 16.10: Procedures and functions in GMP::Stochastic namespace

GMP library. The procedures and functions of the GMP::Stochastic namespace are listed in Table 16.10.

Overview of functionality

For a more detailed overview of the functionality offered by the functions in the GMP::Stochastic namespace, we refer to

- Section 19.4.2 for an outline of the stochastic Benders algorithm,
- the system module containing the AIMMS implementation of the stochastic Benders algorithm, and
- the AIMMS Function Reference for a detailed explanation of the functionality of each function.

16.9 Supporting functions for robust optimization models

Supporting functions for robust models

Table 16.11 lists the functions available in the GMP::Robust namespace in support of working with robust optimization models.

EvaluateAdjustableVariables(*GMP*, *Variables*[, *merge*])

Table 16.11: Procedures and functions in GMP::Robust namespace

Overview of functionality

For a more detailed overview of the functionality offered by the functions in the GMP::Robust namespace, we refer to

- Section 20.4 for an outline of the functionality offered by the procedure GMP::Robust::EvaluateAdjustableVariables, and
- the AIMMS Function Reference for a more detailed explanation.

16.10 Supporting functions for Benders' decomposition

Supporting functions for Benders' decomposition

The Benders' decomposition algorithm (see Chapter 21) is implemented in AIMMS as a combination of a system module that can be included into your model, and a number of supporting functions in the GMP::Benders namespace of the GMP library. The procedures and functions of the GMP::Benders namespace are listed in Table 16.12.

CreateMasterProblem(*GMP*, *Variables*, *name*[, *feasibilityOnly*][, *addConstraints*]) →AllGeneratedMathematicalPrograms
CreateSubProblem(*GMP1*, *GMP2*, *name*[, *useDual*][, *normalizationType*]) →AllGeneratedMathematicalPrograms
UpdateSubProblem(*GMP1*, *GMP2*, *solution*[, *round*])
AddFeasibilityCut(*GMP1*, *GMP2*, *solution*, *cutNo*)
AddOptimalityCut(*GMP1*, *GMP2*, *solution*, *cutNo*)

Table 16.12: Procedures and functions in GMP::Benders namespace

Overview of functionality

For a more detailed overview of the functionality offered by the functions in the GMP::Benders namespace, we refer to

- Chapter 21 for an outline of the Benders' decomposition algorithm,
- the system module containing the AIMMS implementation of the Benders' decomposition algorithm, and
- the AIMMS Function Reference for a detailed explanation of the functionality of each function.

16.11 Creating and managing linearizations

MINLP problems and linearizations

When solving a mixed integer nonlinear (MINLP) problem using an outer approximation approach (see also Chapter 18 for a more detailed description), an associated master MIP problem is created and extended with linearizations of the nonlinear constraints of the original problem with respect to successive solutions of the underlying NLP sub-problem. Using the procedures in the GMP::Linearization namespace, AIMMS allows you to add linearizations of nonlinear constraints to a particular math program instance. Together with the GMP::Instance::CreateMasterMIP procedure to create the initial master MIP problem, these procedures form the heart of the implementation of the outer approximation algorithm in AIMMS, as discussed in Section 18.6.

Managing linearizations

The procedures and functions of the GMP::Linearization namespace are listed in Table 16.13.

Add(*GMP1*, *GMP2*, *solNr*, *conSet*, *devPermitted*, *penalty*, *linNr*[, *jacTol*])
AddSingle(*GMP1*, *GMP2*, *solNr*, *row*, *devPermitted*, *penalty*, *linNr*[, *jacTol*])
Delete(*GMP*, *linNr*)
RemoveDeviation(*GMP*, *row*, *linNr*)
GetDeviation(*GMP*, *row*, *linNr*)
GetDeviationBound(*GMP*, *row*, *linNr*)
GetWeight(*GMP*, *row*, *linNr*)
GetLagrangeMultiplier(*GMP*, *row*, *linNr*)
GetType(*GMP*, *row*, *linNr*)→AllRowTypes
SetDeviationBound(*GMP*, *row*, *linNr*, *value*)
SetWeight(*GMP*, *row*, *linNr*, *value*)
SetType(*GMP*, *row*, *linNr*, *rowType*)

Table 16.13: Procedures and functions in GMP::Linearization namespace

Creating and deleting linearizations

Through the procedures

- GMP::Linearization::Add,
- GMP::Linearization::AddSingle,
- GMP::Linearization::Delete, and
- GMP::Linearization::RemoveDeviation

you can instruct AIMMS to add and delete one or more rows and columns to a given math program instance, representing the linearizations of (nonlinear) constraints of another math program instance at a particular solution point.

Modifying linearizations

You can modify the rows and columns generated by these procedures using the matrix manipulation routines discussed in Section 16.3. The rows and columns generated by AIMMS cannot be associated directly with constraints and variables in your model, but must be addressed using the .ExtendedConstraint and .ExtendedVariable suffices. Section 16.3.7 discusses the precise suffices generated by AIMMS when using the functions GMP::Linearization::Add and GMP::Linearization::AddSingle.

Remaining functions

Through the remaining functions in the GMP::Linearization namespace you can

- get and set information about the devation variables added to the linearized constraints, and their penalties added to the objective, and
- get and set the row types of the generated constraints.

Note the you must use the appropriate .ExtendedConstraint suffix to refer to the particular linearization constraint when using these functions.

16.12 Customizing the progress window

Customizing the progress window

When you are using the GMP library to implement a customized algorithm for a particular problem or problem class, you can use the procedures in the GMP::ProgressWindow namespace to customize the contents of the AIMMS Progress Window. This allows you to provide customized feedback to the end-user regarding the progress of the overall solution algorithm, or to provide simultaneous progress information about multiple solver session executing in parallel.

Customizing progress

The procedures and functions of the GMP::ProgressWindow namespace are listed in Table 16.14. They allow you to modify every aspect of the solver part of the AIMMS progress window.

DisplaySolver(*name*[, *Category*])
DisplayLine(*lineNr*, *title*, *value*[, *Category*])
DisplayProgramStatus(*status*[, *Category*][, *lineNo*])
DisplaySolverStatus(*status*[, *Category*][, *lineNo*])
FreezeLine(*lineNo*, *totalFreeze*[, *Category*])
UnfreezeLine(*lineNo*[, *Category*])
DeleteCategory(*Category*)
Transfer(*Category*, *solverSession*)

Table 16.14: Procedures and functions in GMP::ProgressWindow namespace

Creating a new progress category

When your model executes multiple solver sessions in parallel, you can request AIMMS to create a new progress *category* to display separate solver progress for each solver session in a separate area of the progress window. Using the function GMP::Instance::CreateProgressCategory, you can create a new progress category for a specific mathematical program instance that will subsequently be used to display solver progress for *all* solver sessions associated with that mathematical program instance. Alternatively, you can create a per-session category to display separate solver progress for every single solver session using the function GMP::SolverSession::CreateProgressCategory. The procedure GMP::ProgressWindow::Transfer allows you to share a progress category among several solver sessions. Through the function GMP::ProgressWindow::DeleteCategory you can delete progress categories created by either function.

Freezing category content

Through the functions GMP::ProgressWindow::FreezeLine and GMP::ProgressWindow::UnfreezeLine you can instruct AIMMS to stop and start updating particular areas of the solver progress area associated with the progress category.

Displaying custom content

When you are writing a custom algorithm you can use the progress window to display custom progress information supplied by you using the functions

- GMP::ProgressWindow::DisplaySolver,
- GMP::ProgressWindow::DisplayLine,
- GMP::ProgressWindow::DisplayProgramStatus, and
- GMP::ProgressWindow::DisplaySolverStatus.

When your custom algorithm consists of a sequence of solves, you can use these functions, for instance, to display custom progress information for the overall algorithm, possibly in combination with regular progress for the underlying solves in a separate category.

Example of use

An example of the usage of the GMP::ProgressWindow can be found in the AIMMS module containing the GMP Outer Approximation algorithm discussed in Section 18.6. In this module, the contents of the AIMMS progress window is adapted for the AIMMS Outer Approximation solver.

16.13 Examples of use

This section

In this section there are five examples to illustrate the use of the GMP library. Each example consists of two paragraphs. The first paragraph explains the basic problem and an algorithmic approach, while the second paragraph provides the corresponding implementation in AIMMS using the GMP procedures. Note that these algorithms could also have been implemented using AIMMS' regular SOLVE statement, but at the cost of one or more structure recognition steps during every iteration.

16.13.1 Indexed mathematical program instances

Indexed mathematical program instances

AIMMS does not support indexed mathematical program declarations, which would result in a different mathematical program for every index value when generated. Using the GMP library, however, it is straightforward to generate indexed mathematical program *instances*

Declarations in AIMMS

Consider the following declarations

```
Set Cities {
    Index        : c, j;
}
Set  SelectedCities {
    SubsetOf     : Cities;
    Index        : i;
}
```

together with a mathematical program declaration TransportModel defining a standard transportation problem determining transports from cities i to j.

When SelectedCities equals Cities we would solve the mathematical program for all possible combinations of cities.

Further assume that we have an element parameter IndexedTransportModel(c) into the set AllGeneratedMathematicalPrograms. The following procedure illustrates how indexed mathematical program instances for every city c which restricts the transports from c to all cities j.

Procedure in AIMMS

```
for ( c ) do
    SelectedCities := {c};

    IndexedTransportModel(c) := GMP::Instance::Generate( TransportModel,
                          "TransportModel-" + FormatString("%e", c) );
endfor;
```

16.13.2 Sensitivity analysis

Sensitivity analysis considers how the optimal solution, and the corresponding objective function value, change as a result of changes in input data. Using the GMP library, it is straightforward to write a procedure to determine these sensitivities for a discrete set of input values.

Parametric changes

The following procedure illustrates how parametric changes can be implemented using matrix manipulation functions. The resulting objective function values are stored in a separate identifier.

Procedure in AIMMS

```
myGMP := GMP::Instance::Generate( MathProgramOfInterest );

for ( n ) do
   GMP::Coefficient::Set( myGMP,
                          ResourceConstraint(SelectedResource),
                          ActivityVariable(SelectedActivity),
                          OriginalCoefficient + Delta(n) );

   GMP::Instance::Solve( myGMP );

   ObjectiveValue(n) := GMP::Solution::GetObjective(myGMP, 1);
endfor;
```

16.13.3 Finding a feasible solution for a binary program

There have been instances in which the following simple but greedy heuristic was used successfully to solve a binary program. The algorithm considers linear programming solutions in sequence. During each iteration, the algorithm

Fixing one variable at a time

- selects the single variable that, of all the variables, is nearest but not equal to one of its bounds, and
- fixes the value of this variable to that of the nearest bound.

As soon as such variables can no longer be found (and the last linear programming solution is optimal), a feasible integer solution to the binary program has been found.

Procedure in AIMMS

The following procedure illustrates how fixing one variable at a time can be implemented using matrix manipulation functions. The procedure terminates as soon as there is no solution, or all variables have been fixed.

```
relaxedGMP := GMP::Instance::Generate( RelaxedBinaryProgram );
GMP::Instance::Solve( relaxedGMP );

repeat
    LargestLessThanOne      := ArgMax( j | x(j) <= 1 - Tolerance, x(j) );
    SmallestGreaterThanZero := ArgMin( j | x(j) >= Tolerance,     x(j) );

    break when ( RelaxedBinaryProgram.ProgramStatus = 'Infeasible' or
                 not ( LargestLessThanOne or SmallestGreaterThanZero ) );

    if ( x(SmallestGreaterThanZero) < 1 - x(LargestLessThanOne) )
    then GMP::Column::Freeze( relaxedGMP, x(SmallestGreaterThanZero), 0 );
    else GMP::Column::Freeze( relaxedGMP, x(LargestLessThanOne), 1 );
    endif;

    GMP::Instance::Solve( relaxedGMP );
endrepeat;
```

16.13.4 Column generation

Adding columns

Chapter 20 of the AIMMS book on Optimization Modeling describes a cutting stock problem. This problem is modeled as a linear program with an initial selection of cutting patterns. An auxiliary integer programming model is introduced to generate a new "best" pattern based on the current solution of the linear program and the corresponding shadow prices. Such a pattern is then added to the existing patterns in the linear program, and the next optimal solution is found. This process continues until no further improvement in the value of the objective function can be achieved.

Procedure in AIMMS

The following procedure illustrates how adding columns can be implemented using matrix manipulation functions. During each iteration of the overall process, two different mathematical programs are modified in turn.

```
cuttingStockGMP := GMP::Instance::Generate( CuttingStock );
GMP::Instance::Solve( cuttingStockGMP );

findPatternGMP := GMP::Instance::Generate( FindPattern );
GMP::Instance::Solve( findPatternGMP );

MaxPattern := 0;
while ( PatternContribution > 1 ) do
    MaxPattern += 1;
    AllPatterns += MaxPattern;
    LastPattern := last(AllPatterns);
```

```
    GMP::Column::Add( GMP: cuttingStockGMP, column: RollsUsed(LastPattern) );

    for ( width ) do
        GMP::Coefficient::Set( GMP   : cuttingStockGMP,
                               row   : MeetCutDemand(width),
                               column: RollsUsed(LastPattern),
                               value : CutsInPattern(width)  );
    endfor;
    GMP::Instance::Solve( cuttingStockGMP );

    for ( width ) do
        GMP::Coefficient::Set( GMP   : findPatternGMP,
                               row   : PatternContribution,
                               column: CutsInPattern(width),
                               value : MeetCutDemand(width).ShadowPrice );
    endfor;
    GMP::Instance::Solve( findPatternGMP );
endwhile;
```

Here `MaxPattern` is a parameter of type integer, `AllPatterns` a subset of `Integers`, and `LastPattern` an element parameter with range `AllPatterns`.

16.13.5 Sequential linear programming

Sequential linear programming

Linear constraints and a nonlinear objective function together form a special class of nonlinear programs. It is possible to solve a problem of this class by solving a sequence of linear programs. The main requirement is that the nonlinear objective function has first-order derivatives. The objective function can then be linearized around the solution of a previous linear program. By restricting the linearized function to an appropriate finite box, a new solution point is found. The sequence of linear programs terminates when the appropriate box has become sufficiently small. Upon termination, the optimal solution, as last found, is considered to be a local optimum of the underlying nonlinear program.

Procedure in AIMMS

The following procedure illustrates how sequential linear programming can be implemented using matrix manipulation functions. The procedure assumes the existence of finite upper and lower bounds on the variables, and the presence of a function `ComputeGradient` to compute the required first partial derivatives with respect to the variables in the objective function. To implement the function `ComputeGradient` one can, for instance, use the built-in GMP function `GMP::Solution::GetFirstOrderDerivative` (see also Section 16.4).

```
linearizedGMP := GMP::Instance::Generate(LinearizedProgram);
GMP::Instance::Solve(linearizedGMP);

BoxWidth(j)  := 0.1 * (x.upper(j) - x.lower(j));
x(j)         := 0.5 * (x.upper(j) + x.lower(j));

while ( max( j, BoxWidth(j) ) > Tolerance ) do
   ObjCoeff(j) := ComputeGradient(x)(j);
```

```
   for (j) do
      GMP::Column::SetLowerBound ( linearizedGMP, x(j),
                                   max(x.lower(j), x(j) - 0.5*BoxWidth(j)) );
      GMP::Column::SetUpperBound ( linearizedGMP, x(j),
                                   min(x.upper(j), x(j) + 0.5*BoxWidth(j)) );
      GMP::Coefficient::Set( linearizedGMP, ObjectiveRow,
                             x(j), ObjCoeff(j)            );
   endfor;
   GMP::Instance::Solve(linearizedGMP);

   BoxWidth(j) *= ShrinkFactor;
endwhile;
```

Chapter 17

Advanced Methods for Nonlinear Programs

Problems of nonlinear programs

For non-convex nonlinear mathematical programs (NLPs), nonlinear solvers have no guarantee of returning the global optimum. Due to the local search algorithms employed by nonlinear solvers, their solution process depends on the starting point provided by the user. Nonlinear solvers can, therefore, easily end up in, non-unique, local optima, or, even worse, may not even find a feasible solution for a given starting point.

How to counteract?

To counteract these facts, a number of possible actions can be taken.

- Use a global solver, such as BARON, to solve the NLP. Global solvers, however, usually only work well on relatively small NLP problems.
- Use a multistart algorithm to solve the NLP problem for multiple starting points in order to have a better chance to find the global optimum.
- Use the AIMMS Presolver to reduce the problem size and tighten the bounds of the remaining variables and constraints of the NLP. This will reduce the space which the nonlinear solver needs to search in order to find an optimal solution.

This chapter

This chapter discusses the presolve techniques for nonlinear programs available in AIMMS. The chapter also discusses the multistart algorithm built into AIMMS. Using the multistart algorithm will increase the total solution time, but, in general, will also improve the solution found by nonlinear solvers.

17.1 The AIMMS Presolver

The need for a presolver

Of all nonlinear solvers in AIMMS only a couple use (limited) preprocessing techniques. Therefore, AIMMS itself has implemented a presolve algorithm with the goal to reduce the size of the problem and to tighten the variable bounds, which may help the solver to solve nonlinear problems faster. Besides the BARON global solver, all nonlinear solvers in AIMMS are local solvers, i.e. the solution found by the solver is a local solution and cannot be guaranteed to be a global solution. The presolve algorithm may help the solver in finding a better solution. A local solver might sometimes fail to find a solution and then it is often not clear whether that is caused by the problem being infeasible or

by the solver failing to find a solution for a feasible problem. The presolve algorithm may reveal inconsistent constraints and/or variable bounds and hence identify a problem as infeasible.

Presolve techniques

Consider the following constrained nonlinear optimization problem:

Minimize:

$$f(x)$$

Subject to:

$$\begin{aligned} g(x) &\leq d \\ Ax &\leq b \\ l \leq x &\leq u \end{aligned}$$

The objective function $f(x)$ can either be linear or nonlinear, while $g(x)$ is a nonlinear function. The AIMMS presolve algorithm will (amongst others)

- remove singleton rows by moving the bounds to the variables,
- reduce variable bounds from linear and nonlinear constraints that contain bounded variables,
- delete fixed variables,
- remove one variable of a doubleton, and
- delete redundant constraints.

A detailed description of each of these techniques can be found in [Fo94].

Singleton rows

A singleton row is a linear constraint that contains only one variable. An equality singleton row fixes the variable to the right-hand-side value of the row, and unless this value conflicts with the current bounds of the variable in which case the problem is infeasible, AIMMS can remove both the row and variable from the problem. An inequality singleton row introduces a new bound on the variable which can be redundant, tighter than an existing bound in which case AIMMS will update the bound, or infeasible. The AIMMS presolve algorithm will remove all singleton rows.

Deleting fixed variables

If a variable is fixed then sometimes another row becomes a singleton row, and if that row is an equality row AIMMS can fix the remaining variable and remove it from the problem. By repeating this process AIMMS can solve any triangular system of linear equations that is part of the problem.

Bound reductions using linear constraints

The following analysis applies to a linear "less than or equal to" constraint. A similar analysis applies to other constraint types. Assume we have a linear constraint i that originally has the form

$$\sum_j a_{ij} x_j \leq b_i \tag{17.1}$$

Assuming that all variables in this constraint have finite bounds, we can determine the following lower and upper limits on constraint i

$$\underline{b_i} = \sum_{j \in P_i} a_{ij} l_j + \sum_{j \in N_i} a_{ij} u_j \tag{17.2}$$

and

$$\overline{b_i} = \sum_{j \in P_i} a_{ij} u_j + \sum_{j \in N_i} a_{ij} l_j \tag{17.3}$$

where $P_i = \{j \mid a_{ij} > 0\}$ and $N_i = \{j \mid a_{ij} < 0\}$ define the sets of variables with a positive coefficient and negative coefficient in constraint i respectively.

Bound analysis

By comparing the lower and upper limits of a constraint with the right-hand-side value we obtain one of the following situations:

- $\underline{b_i} > b_i$: constraint (17.1) cannot be satisfied and is infeasible.
- $\underline{b_i} = b_i$: constraint (17.1) can only be satisfied if all variables in the constraint are fixed on their lower bound if they have a positive coefficient, or fixed on their upper bound if they have a negative coefficient. The constraint and all its variables can be removed from the problem.
- $\overline{b_i} \leq b_i$: constraint (17.1) is redundant, i.e. will always be satisfied, and can be removed from the problem.
- $\underline{b_i} < b_i < \overline{b_i}$: constraint (17.1) cannot be eliminated but can often be used to improve the bounds of one or more variables as we will see below.

If $\underline{b_i} < b_i < \overline{b_i}$, then combining (17.1) with (17.2) gives the following bounds on a variable k in constraint i:

$$x_k \leq l_k + (b_i - \underline{b_i})/a_{ik} \qquad \text{if } a_{ik} > 0 \tag{17.4}$$

and

$$x_k \geq u_k + (b_i - \underline{b_i})/a_{ik} \qquad \text{if } a_{ik} < 0 \tag{17.5}$$

If the upper bound given by (17.4) is smaller than the current lower bound of variable k then the problem is infeasible. If it is smaller then the current upper bound of variable k, AIMMS will update the upper bound for variable k. Something similar holds for the lower bound as given by (17.5). Note that bounds (17.4) and (17.5) can only be derived if all bounds l_j and u_j in (17.2) are finite. But also if exactly one of the bounds in (17.2) is an infinite bound, AIMMS can still find an implied bound for the corresponding variable.

Bound reductions using nonlinear constraints

We can rewrite a nonlinear constraint $g_i(x) \leq d_i$ as

$$\sum_j a_{ij} x_i + h_i(y) \leq d_i \tag{17.6}$$

separating the linear variables x in this constraint from the nonlinear variables y. As before, we can find lower and upper limits on the linear part of the constraint, and again we denote them by $\underline{b_i}$ and $\overline{b_i}$ respectively. For this constraint

we can derive the following upper bound on the nonlinear term in (17.6):

$$h_i(y) \le d_i - \underline{b_i} \tag{17.7}$$

Note that if there are no linear terms in constraint (17.6) then $\underline{b_i} = 0$.

Nonlinear analysis using expression trees

Nonlinear expressions in AIMMS are stored in an expression tree. By going through the expression tree from the top node to the leafs we can sometimes derive bounds on some of the variables in the expression. For example, assume we have the constraint

$$\sqrt{\ln x} \le 2$$

with x unbounded. It follows that the $\ln x$ sub-expression should be in the range $[0,4]$ since $\sqrt{y}$ is not defined for $y < 0$, which in turn implies that x should be in the range $(1, e^4]$.

Types of nonlinear analysis

AIMMS can analyze nonlinear expressions for various types of reductions, and uses various types of techniques, such as:

- operator domain analysis: reduce bounds on operator arguments by the implicit domains of operators such as $\sqrt{x}$ or $\ln x$,
- operator range analysis: compute the bounds of a nonlinear expression on the basis of known bounds on the argument(s) and use those bounds for further reductions, and
- for invertible functions, compute bounds on operator arguments on the basis of bounds on a known operator range.

Supported operators

The presolve algorithm can handle nonlinear expressions build up by the operators listed in Table 17.1. If a nonlinear constraint contains an operator that is not in this table then it will be ignored by the presolve algorithm.

$\log_{10} x$, $\ln x$	$\exp x$, e^x
x^a, a^x $(a \neq 0)$	x^y
$\sin x$, $\cos x$, $\tan x$	$\arcsin x$, $\arccos x$, $\arctan x$
$x + y$, $x - y$	$x \cdot y$, x/y

Table 17.1: Operators used by the presolve algorithm

Doubletons

If a problem contains a constraint of the form $x = ay$, $a \neq 0$, then the variables x and y define a doubleton. If the presolve algorithm detects a doubleton then it will replace the variable x by the term ay in every constraint in which x appears, and remove the variable x from the problem. For some problems good initial values are given to the variables. In case the initial value given to x does not match the initial value of y according to the relationship $x = ay$, it is unclear which initial value we should assign to y. Preliminary test results

showed that in such a case it is better not to remove the doubleton, and pass both variables to the solver with their own initial value. This has become the default behavior of our presolve algorithm regarding doubletons.

The presolve algorithm

The AIMMS Presolver iteratively applies all reduction techniques discussed above until no further reductions are available anymore, or an iteration limit has been reached. Various options are available in the **Solvers general - AIMMS presolver** section of the option tree to steer the presolve algorithm. For instance a user can choose to only use linear constraints for reducing bounds, or to not remove doubletons.

Mixed integer programming problems

If the optimization problem contains binary variables then the AIMMS Presolver can apply probing which is a technique that looks at the logical implications of fixing a binary variable to 0 or 1. Probing can be used to reduce more variables bounds, reformulate constraints or improve coefficients. In some cases quadratic constraints containing binary variables can be reformulated as linear constraints. Coefficient improvement is a process of improving the coefficients of the binary variables such that the relaxation becomes more tight. A detailed description of probing and coefficient improvement can be found in [Sa94].

Successes may vary

The benefits of using the AIMMS Presolver may vary from model to model. The solution of presolved NLPs may become better or worse compared to the original NLP. Presolving may change infeasible NLPs to feasible problems for a given starting point, or vice versa. Also, presolving may make the model more degenerate and harder to solve. Finaly, for eliminated constraints and variables dual information is lost, and AIMMS makes no effort yet to recover the lost dual information, as this may be very hard in the presence of nonlinear reductions.

17.2 The AIMMS multistart algorithm

The multistart algorithm

A multistart algorithm calls an NLP solver from multiple starting points, keeps track of (all) feasible solutions found by the NLP solver, and reports back the best of these as its final solution.

Why use multistart?

A multistart algorithm can improve the reliability of any NLP solver, by calling it with many starting points. A single call to a NLP solver can fail (return a status of infeasible), but multiple calls from the widely spaced starting points provided by a multistart algorithm have a much better chance of success.

Basic techniques

In a pure multistart algorithm many local searches will converge to the same local minimum. Computational effort can be reduced if the minimizations leading to the same local minimum point can be identified and combined at early stages. An improvement is to use cluster analysis techniques to identify regions of points that will lead to the same local minimum.

Algorithm used by AIMMS

AIMMS uses a multistart algorithm that does not use advanced cluster analysis techniques, but instead tries to identify areas of points that will lead to the same local solution. These areas are updated (and become larger) whenever a starting point is found that leads to a local solution that has already been found before. An more detailed description of a multistart algorithm similar to the one used by AIMMS can be found in [Ka87].

Definitions

The following terminology is used for the multistart algorithm

- Sample points: a set of points that were randomly sampled.
- Cluster point: a point that defines the center of a cluster, i.e., a cluster is a circle/ball with a cluster point as its center.
- Starting point: a point used as an initial solution ("hotstart") for solving the NLP.
- Local solution: a solution found by the NLP solver (by using a starting point). A local solution belongs to exactly one cluster point. A local solution can be infeasible.

Two algorithms

The multistart module implements two algorithms, namely a basic algorithm and a dynamic algorithm in which the number of iterations is changed dynamically. The inputs for both algorithms are:

- a GMP associated with an NLP,
- `NumberOfSamplePoints`, and
- `NumberOfSelectedSamplePoints`.

The basic algorithm

The basic algorithm employs the following steps:

0. Set `IterationCount` equal to 1.
1. Generate `NumberOfSamplePoints` sample points from the uniform distribution. Calculate the penalized objective for all sample points and select the best `NumberOfSelectedSamplePoints` sample points.
2. For all sample points (`NumberOfSelectedSamplePoints` in total) do:
   - For all clusters, calculate the distance between the sample point and the center of the cluster. If the distance is smaller than the radius of the cluster (i.e., the sample point belongs to the cluster) then delete the sample point.
3. For all (remaining) sample points do:
   - Solve the NLP by using the sample point as its starting point to obtain a candidate local solution.

- For all clusters do:
  - Calculate the distance between the candidate local solution and the local solution belonging to the cluster.
  - If the distance equals 0 (which implies that the candidate local solution is the same as the local solution belonging to the cluster) then update the center and radius of the cluster by using the sample point.
  - Else, construct a new cluster by using the mean of the sample point and the candidate local solution as its center with radius equal to half the distance between these two points. Assign the candidate local solution as the local solution belonging to the cluster.
- For all remaining sample points, calculate the distance between the sample point and the center of the updated or the new cluster. If the distance is smaller than the radius of the cluster then delete the sample point.

4. Increment `IterationCount`. If the number of iterations exceeds the `IterationLimit`, then go to step (5). Else go to step (1).
5. Order the local solutions and store the `NumberOfBestSolutions` solutions in the solution repository.

By default, the algorithm uses the initial variable values as the first "sample" point in the first iteration.

The dynamic algorithm: first phase

The dynamic algorithm contains two phases. The first phase is similar to the basic algorithm but with some differences. The dynamic algorithm starts by determining the best sampling box for the creation of the random points (in step 0). For the first sample point, which can be an initial point provided by the user or the first randomly generated point, a method is applied to compute an approximately feasible solution (see [Ch04]) to increase the chance that this first sample point will lead to a feasible solution. Finally, if the dynamic algorithm did not find any feasible solution during the first iterations, and all local solutions found contain large infeasibilities, then a heuristic will be used to update the variable bounds (in step 4).

The dynamic algorithm: second phase

The second phase of the dynamic algorithm is only conducted if no feasible solution was found in the first phase, or if the objective values of the feasible solutions found in the first phase vary. The second phase differs for both situations. If no feasible solution was found in the first phase then the algorithm will continue with steps 1 to 4 until a feasible solution is found or the time limit is hit. In each iteration, the algorithm will now use the method for computing an approximately feasible solution for the first randomly generated point. In the other case, in which the objective values of the feasible solutions found in the first phase vary, the second phase will continue with steps 1 to 4 until enough feasible solutions are found to satisfy a Bayesian estimate for the number of local feasible solutions (or if the time limit is hit).

Using the AIMMS *multistart algorithm*

The AIMMS multistart algorithm is implemented as a system module, with the name Multi Start, that you can add to your project. You can install this module using the **Install System Module** command in the AIMMS **Settings** menu. The algorithm outlined above is implemented in the AIMMS language. Some supporting functions that are computationally difficult, or hard to express in the AIMMS language, have been added to the GMP library in support of the AIMMS multistart algorithm.

Calling the multistart algorithm

The main procedure to start the multistart algorithm is the procedure DoMultiStart. The only mandotory input is a generated mathematical program obtained by calling the GMP::Instance::Generate function of the GMP library discussed in Section 16.2. Therefore the multistart algorithm can be called by using for example:

```
MulStart::DoMultiStart( myGMP );
```

Here MulStart is the prefix of the multistart module. The behavior of the multistart algorithm is influenced by several control parameters, which are discussed in Section 17.3.

Optional arguments

The procedure DoMultiStart contains two optional arguments (with a default value of 0) which can be used to specify the number of sample points and the number of selected sample points (as outlined above). If both arguments are not specified (like in the example of the previous paragraph) or are equal to 0, then the multistart algorithm will use the dynamic algorithm, and otherwise the basic algorithm. For example, if

```
MulStart::DoMultiStart( myGMP, 20, 10 );
```

is used then the basic algorithm will be used with 20 sample points and 10 selected sample points. If the dynamic algorithm is used then the multistart algorithm will automatically select values for the number of sample points and the number of selected sample points. It is possible to use the dynamic algorithm and specify the number of sample points and the number of selected sample points yourself by calling the procedure DoMultiStartDynamic.

Supporting GMP *functions*

The GMP library contains the following functions to support the multistart algorithm:

- GMP::Solution::RandomlyGenerate (used in step (1))
- GMP::Solution::GetPenalizedObjective (used in step (1))
- GMP::Solution::GetDistance (used in steps (2) and (4))
- GMP::Solution::ConstructMean (used in step (4))
- GMP::Solution::UpdatePenaltyWeights (used during initialization)

Optionally it is possible to (approximately) project each sample point to the feasible region by using the procedure GMP::Instance::FindApproximatelyFeasibleSolution.

Modifying the algorithm

Because the multistart algorithm is written in the AIMMS language, you have complete freedom to modify the algorithm in order to tune it for your nonlinear programs.

17.3 Control parameters that influence the multistart algorithm

Control parameters

The multistart module defines several parameters that influence the multistart algorithm. These parameters have a similar functionality as options of a solver, e.g., CPLEX. The most important parameters, with their default setting, are shown in Table 17.2. The parameters that are not self-explanatory are explained in this section; the last column in the table refers to the subsection that discusses the corresponding parameter.

Parameter	Default	Range	Subsection
IterationLimit	5	{1,maxint}	17.3.1
TimeLimit	0	{0,maxint}	17.3.2
TimeLimitSingleSolve	0	{0,maxint}	17.3.2
ThreadLimit	0	{0,maxint}	17.3.3
UseOpportunisticAlgorithm	0	{0,1}	17.3.4
NumberOfBestSolutions	1	{1,1000}	17.3.5
ShrinkFactor	0.95	[0,1]	17.3.6
UsePresolver	1	{0,1}	17.3.7
UseInitialPoint	1	{0,1}	17.3.8
UseConstraintConsensusMethod	0	{-1,1}	17.3.9
MaximalVariableBound	1000	[0,inf)	17.3.10
ShowSolverProgress	0	{0,1}	17.3.11

Table 17.2: Control parameters in the multistart module

17.3.1 Specifying an iteration limit

Parameter IterationLimit

The parameter IterationLimit can be used to set a limit on the number of iterations used by the multistart algorithm. This limit is use in the basic algorithm and in the first phase of the dynamic algorithm.

17.3.2 Specifying a time limit

Parameter TimeLimit

The parameter TimeLimit can be used to set a limit on the total elapsed time (in seconds) used by the multistart algorithm. The default value of 0 has a special meaning; in that case there is no time limit.

It is also possible to set a time limit for every single solve started by the multistart algorithm by using the parameter TimeLimitSingleSolve. Also the default value of 0 of this parameter has a special meaning; in that case there is no time limit.

Parameter TimeLimitSingleSolve

17.3.3 Using multiple threads

The parameter ThreadLimit controls the number of threads that should be used by the multistart algorithm. Each thread will be used to solve one NLP using an asynchronous solver session. At its default setting of 0, the algorithm will automatically use the maximum number of threads, which is limited by the number of cores on the machine and the amount of solver sessions allowed by the AIMMS license.

Parameter ThreadLimit

17.3.4 Deterministic versus opportunistic

By default the multistart algorithm runs in deterministic mode. Deterministic means that multiple runs with the same model using the same parameter settings and the same solver on the same computer will reproduce the same results. The number of NLP problems solved by the multistart algorithm will then also be the same. In contrast, opportunistic implies that the results, and the number of NLP problems solved, might be different. Usually the opportunistic mode provides better performance. The parameter UseOpportunisticAlgorithm can be used to switch to the opportunistic mode. Note that if the multistart algorithm uses only one thread then the algorithm will always be deterministic.

Parameter UseOpportunisticAlgorithm

17.3.5 Getting multiple solutions

By default the multistart algorithm will return one solution, namely the best solution that the algorithm finds. By setting the parameter NumberOfBestSolutions to a value higher than 1, the multistart algorithm will store the best n solutions found in the solution repository (see Section 16.4). Here n denotes the value of this parameter.

Parameter NumberOfBestSolutions

17.3.6 Shrinking the clusters

The clusters created by the multistart algorithm would normally grow as more and more points are assigned to the clusters. As a side effect, a new sample point is then more likely to be directly assigned to a cluster, in which case no NLP is solved for that sample point, thereby increasing the chance that it ends up in the wrong cluster. To overcome this problem, the multistart

Parameter ShrinkFactor

algorithm automatically shrinks all clusters after each iteration by a constant factor which is specified by the parameter ShrinkFactor.

17.3.7 Combining multistart and presolver

Parameter UsePresolver

By default the multistart algorithm starts by applying the AIMMS Presolver to the NLP problem. By preprocessing the problem, the ranges of the variables might become smaller which has a positive effect on the multistart algorithm as then the randomly generated sample points are more likely to be good starting points. The parameter UsePresolver can be used to switch off the preprocessing step.

17.3.8 Using a starting point

Parameter UseInitialPoint

Sometimes the level values, assigned to the variables before solving the NLP problem, provide a good starting point. By default the multistart algorithm will use this initial point as the first sample point but only in the first iteration. This behavior is controlled by the parameter UseInitialPoint.

17.3.9 Improving the sample points

Parameter UseConstraintConsensusMethod

The sample points are randomly generated by using the intervals defined by the lower and upper bounds of the variables. Such a sample point is very likely to be infeasible with respect to the constraints. The constraint consensus method, which is described in [Ch04], tries to find an approximately feasible point for a sample point. Using this method might slow down the multistart algorithm but the chance of generating (almost) feasible sample points increases. The constraint consensus method can be activated by using the parameter UseConstraintConsensusMethod. If this parameter is set to 1 then the constraint consensus method will be used whenever possible, and if it is set to -1 then it will never be used. At its default value of 0, the algorithm automatically decides when to use the constraint consensus method.

17.3.10 Unbounded variables

Parameter MaximalVariableBound

A multistart algorithm requires that all variable bounds are finite. Therefore the multistart algorithm in AIMMS will use a fixed value for all infinite upper and lower variable bounds. This fixed value is specified by the parameter MaximalVariableBound. The value of this parameter might be updated automatically in case the dynamic algorithm is used.

17.3.11 Solver progress

Parameter ShowSolverProgress

By default the progress window will only show general progress information for the multistart algorithm, including the objective value, the number of iterations, the elapsed time, etc. By switching on the parameter `ShowSolverProgress` also progress information by the NLP solver will be displayed. If multiple solver sessions are (asynchronous) executing at the same time then only the progress information of one of them will be shown.

Chapter 18

AIMMS Outer Approximation Algorithm for MINLP

Open solver approach

Outer approximation (see [Du86]) is a basic approach for solving Mixed-Integer NonLinear Programming (MINLP) models. The underlying algorithm is an interplay between two solvers, one for solving mixed-integer linear models and one for solving nonlinear models. Even though the standard outer approximation algorithm is provided with AIMMS, you as an algorithmic developer may want to customize the individual steps in order to obtain better performance and/or a better solution for your particular model.

The AIMMS Outer Approximation algorithm

The outer approximation algorithm in AIMMS is, therefore, provided as a customizable procedure written in the AIMMS language itself using functions and procedures provided by the GMP library (a white box solver), whereas most other outer approximation solvers are provided as a closed implementation (a black box solver). The outer approximation algorithm in AIMMS is implemented as a system module with the name `GMP Outer Approximation`. You can install this module using the **Install System Module** command in the AIMMS **Settings** menu. In the remainder of this chapter, we will refer to the outer approximation algorithm as AIMMS Outer Approximation (AOA).

Convex algorithm

Besides the basic algorithm, the AOA module also implements the Quesada-Grossmann algorithm (see [Qu92]) which is designed to solve convex MINLP models. The basic algorithm can also be used to solve convex models but the Quesada-Grossmann algorithm is often more efficient.

This chapter

In this chapter you find the description of, and the motivation behind, the open approach to solving MINLP models based on outer approximation. We continue with a brief introduction to the problem statement and the basic algorithm. Next we explain how the AOA algorithm can be setup, followed by a detailed explanation of the parameters inside the AOA module that can be used to control the outer approximation algorithm. We then describe the Quesada-Grossmann algorithm for convex MINLP models. Next we describe an initial implementation of the basic solution algorithm using procedures in the AIMMS language is described. These procedures use functions that are especially designed to support the open approach. The chapter concludes with

suggestions on additional ways to vary the individual steps of the overall algorithm in order to obtain customized versions of the outer approximation algorithm.

18.1 Problem statement

MINLP

The mixed-integer nonlinear programming models to be solved can be expressed as follows.

Minimize:

$$f(x, y)$$

Subject to:

$$\begin{aligned} h(x, y) &= 0 \\ g(x, y) &\leq 0 \\ Cy + Dx &\leq d \\ x \in \mathcal{X} &= \{x \in \mathbb{R}^n | x^L \leq x \leq x^U\} \\ y \in \mathcal{Y} &= \mathbb{Z}^m \end{aligned}$$

Usual assumption

The usual assumption is that the nonlinear subproblem (i.e. the model in which all integer variables are fixed) is convex. This assumption is to guarantee that each locally optimal solution of the nonlinear subproblem is also a globally optimal solution. In practice this assumption does not always hold, but the algorithm can still be applied. Convergence to a global optimum of the MINLP using the outer approximation algorithm is then no longer guaranteed.

18.2 Basic algorithm

Algorithm in words

The algorithm solves an alternating sequence of mixed-integer linear models and nonlinear models.

1. First, the entire model is solved as a nonlinear program with all the integer variables relaxed as continuous variables between their bounds.
2. Then a linearization is carried out around the optimal solution, and the resulting constraints are added to the linear constraints that are already present. This new linear model is referred to as the master MIP model.
3. The master MIP problem is solved as an mixed-integer linear program.
4. The integer part of the resulting optimal solution is then temporarily fixed, and the original MINLP model with fixed integer variables is solved as a nonlinear subproblem.
5. Again, a linearization around the optimal solution is constructed and the new linear constraints are added to the master MIP problem. To prevent

cycling, one or more constraints are added to cut off the previously-found integer solution of the master problem.

6. Steps 3–5 are repeated until one of the termination criteria is satisfied.

A more detailed description of the general Outer Approximation algorithm can be found in [Du86].

Convexity and convergence

As linearizations are added to the master MIP problem, the model becomes an improved approximation of the original MINLP model. Using the usual convexity assumption regarding the nonlinear subproblem, convergence to a global optimum occurs when the objective function value of the master MIP problem is worse than the value associated with the NLP subproblem.

Termination ...

Several termination criteria are used in practice. These criteria can be used in isolation or in some logical combination. Three of them are discussed in the following paragraphs.

... iteration limit

Perhaps the most frequently-used criterion is the iteration limit. One reason is that a good solution is usually found during the first few iterations. Another reason for using an iteration limit is that the size of the underlying master MIP problem tends to grow significantly each time linearization constraints are added, causing an increase in computation time.

... objective worsening

A second criterion is the worsening of the objective function value of two successive nonlinear subproblems. This worsening occurs quite frequently, even if the NLP subproblem is convex. The underlying reason is that the master MIP problem will not always select binary solutions that lead to successively improving NLPs. This criterion seems appropriate when the worsening is persistent over several iterations.

... crossover

A third termination criterion is insufficient improvement in the objective function value of the master MIP problem in relation to the objective function value of the previously solved NLP subproblem. The difference between these two values represents the optimality gap, since the master MIP problem represents an outer approximation (thus a relaxation) of the original MINLP model. When the gap is closed at crossover, the optimal solution has been found provided the NLP subproblem is convex.

Final solution

Upon termination of the algorithm, the known best solution (also referred to as the incumbent solution) is declared as the final solution. In many practical applications, this solution is not necessarily optimal due to termination based on an iteration limit. In addition, it is often not possible to verify that the NLP subproblem is convex.

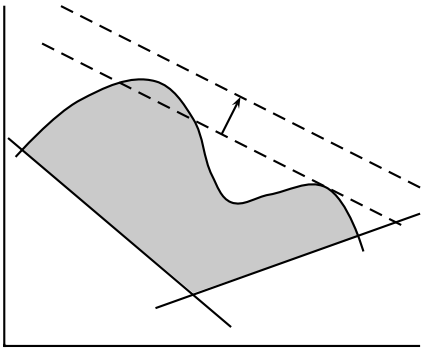


Figure 18.1: Effect of loosening a linearization

Linearizations

The term 'outer approximation' refers to the linear approximation of the convex nonlinear constraints at selected points along the boundary of the convex solution region. The accumulation of such inequality constraints forms an outer approximation of the solution region, and this approximation can be used in the optimization rather than the nonlinear constraints from which it was derived. The formula for the linearization of a scalar nonlinear inequality $g(x, y) \leq 0$ around the point $(x, y) = (x^0, y^0)$ is as follows.

$$g(x^0, y^0) + \nabla g(x^0, y^0)^T \begin{bmatrix} x - x^0 \\ y - y^0 \end{bmatrix} \leq 0$$

The nonconvex case

The linear approximation ceases to be an outer approximation if the solution region is not convex. In this situation there is the possibility that portions of the solution region are cut off as illustrated in Figure 18.1.

Loosening inequalities

In practical implementations of the outer approximation algorithm, the linearizations are allowed to move away from the feasible region. Such heuristic flexibility allows solutions to be found that would otherwise have been cut off. The implementation allows deviations through the use of artificial nonnegative variables and then penalizing them while solving the master problem.

Open solver approach

The basic outer approximation algorithm that is part of the AOA module has been completely implemented using functionality provided by the GMP library.

- From the math program instance representing the original MINLP model, a new math program instance representing the initial master MIP problem can be created using the function `GMP::Instance::CreateMasterMIP`.
- The functions from the `GMP::Linearization` namespace can be used to add linearizations of the nonlinear constraints of the original MINLP model to the master MIP, in a customizable manner.

- Using the GMP::Instance::FixColumns procedure, the integer columns of the nonlinear subproblem can fixed to the current integer solution of the master MIP.
- Using the GMP::Instance::AddIntegerEliminationRows procedure, prior integer solutions of the master MIP are excluded from subsequent solves.

18.3 Using the AOA algorithm

AOA *module ...*

The basis GMP implementation of the AIMMS Outer Approximation (AOA) algorithm can be found in a single AIMMS module, called GMP Outer Approximation, that is provided as part of the AIMMS system. You can install this module using the **Install System Module** command in the AIMMS **Settings** menu.

Basic algorithm

The procedure DoOuterApproximation inside the module implements the basic algorithm from the previous section. The procedure DoOuterApproximation has one input argument, namely:

- MyGMP, an element parameter with range AllGeneratedMathematicalPrograms.

This procedure is called as follows:

```
generatedMP := GMP::Instance::Generate( SymbolicMP );

GMPOuterApprox::DoOuterApproximation( generatedMP );
```

Here SymbolicMP is the symbolic mathematical program containing the MINLP model, and generatedMP is an element parameter in the predefined set AllGeneratedMathematicalPrograms. GMPOuterApprox is the prefix of the AOA module. The implementation of this procedure will be discussed in Section 18.6.

Modifying the algorithm

Because the AIMMS Outer Approximation algorithm is completely implemented using functionality provided the GMP library, you have the complete freedom to modify the math program instances generated by the basic AOA algorithm using the matrix manipulation routines discussed in Section 16.3. Such problem-specific modifications to the basic algorithm may help you to find a better overall solution to your MINLP model, or to find a good solution faster.

18.4 Control parameters that influence the AOA algorithm

Control parameters

The multistart module defines several parameters that influence the outer approximation algorithm. These parameters have a similar functionality as options of a solver, e.g., CPLEX. The most important parameters, with their default setting, are shown in Table 18.1. The parameters that are not self-explanatory are explained in this section; the last column in the table refers to the subsection that discusses the corresponding parameter.

Parameter	Default	Range	Subsection
IterationMax	20	{0,maxint}	
TimeLimit	0	{0,maxint}	18.4.1
CreateStatusFile	0	{0,1}	
UsePresolver	1	{0,1}	18.4.2
UseMultistart	0	{0,1}	18.4.3
TerminateAfterFirstNLPIsInteger	1	{0,1}	18.4.4
IsConvex	0	{0,1}	18.4.5
RelativeOptimalityTolerance	1e-5	{0,1}	18.4.5
NLPUseInitialValues	1	{0,1}	18.4.6

Table 18.1: Control parameters in the outer approximation module

18.4.1 Specifying a time limit

Parameter TimeLimit

The parameter TimeLimit can be used to set a limit on the total elapsed time (in seconds) used by the outer approximation algorithm. The default value of 0 has a special meaning; in that case there is no time limit.

18.4.2 Using the AIMMS Presolver

Parameter UsePresolver

By default the outer approximation algorithm starts by applying the AIMMS Presolver to the MINLP model. By preprocessing the MINLP model, the model might become smaller and easier to solve. The parameter UsePresolver can be used to switch off the preprocessing step.

18.4.3 Combining outer approximation with multistart

Parameter UseMultistart

If the parameter UseMultistart is switched on then the outer approximation algorithm will use the multistart algorithm to solve the nonlinear subproblems. For non-convex models this can have a positive effect on the quality of the solution that is returned by the outer approximation algorithm. The multistart algorithm is described in section 17.2. The parameters MultistartNumberOfSamplePoints and MultistartNumberOfSelectedSamplePoints can be used to specify the number of sample and selected sample points, respecively, as used by the multistart algorithm.

Multistart module

To use the multistart algorithm, the system module Multi Start should be added to your project. You can install this module using the **Install System Module** command in the AIMMS **Settings** menu.

18.4.4 Terminate if solution of relaxed model is integer

Parameter TerminateAfterFirstNLPIsInteger

By default the outer approximation algorithm will terminate if it finds an integer solution for the initial NLP problem, which is obtained from the MINLP model by relaxing the integer variables. By switching off the parameter TerminateAfterFirstNLPIsInteger you can enforce the algorithm to continue.

18.4.5 Solving a convex model

Parameter IsConvex

The parameter IsConvex can be used to indicate that the model is convex. In that case the outer approximation algorithm will no longer stop after the iteration limit is hit, as specified by the parameter IterationMax. Instead, the algorithm will stop if the gap between the objective values of the master MIP problem and the nonlinear subproblem is sufficiently small, as controlled by the parameter RelativeOptimalityTolerance. Note that AIMMS cannot identify whether a model is convex or not.

18.4.6 Starting point strategy for NLP subproblems

Parameter NLPUseInitialValues

The parameter NLPUseInitialValues specifies the starting point strategy used for solving the NLP subproblems. For nonconvex nonlinear problems the starting point often has a big influence on the solution that the NLP solver will find. By default the AOA algorithm will use the initial values as provided by the user for all NLP subproblems that are solved. By setting this parameter to 0, the algorithm will use the solution of the previous master MIP problem as the starting point for the next NLP subproblem (and for the initial NLP it will use the initial values provided by the user). Note: if one of the parameters UseMultistart or IsConvex equals 1 then NLPUseInitialValues is automatically set to 0.

18.5 The Quesada-Grossmann algorithm

One MIP problem

Quesada and Grossmann ([Qu92]) noticed that the classic outer approximation algorithm often spends a large amount of time in solving the MIP problems in which a significant amount of rework is done. They proposed an algorithm in which only one MIP problem is solved. The algorithm implemented in AIMMS uses a callback procedure for lazy constraints which is supported by modern MIP solvers like CPLEX and GUROBI.

Convex models

The Quesada-Grossmann algorithm is designed to solve convex MINLP models. The basic outer approximation algorithm can also be used to solve convex models by using the parameter IsConvex, but the Quesada-Grossmann algorithm is often more efficient. The Quesada-Grossmann algorithm is also available in the GMP Outer Approximation module.

Calling procedure

The procedure DoConvexOuterApproximation inside the module implements the Quesada-Grossmann algorithm. This procedure is called in the same way as the DoOuterApproximation procedure of Section 18.3, which implements the basic algorithm. The following control parameters in Table 18.1 can be used to influence the Quesada-Grossmann algorithm: TimeLimit, CreateStatusFile, UsePresolver and RelativeOptimalityTolerance.

18.6 A first and basic implementation

Calling the AOA *algorithm*

To call the AOA algorithm, the GMP library is used to generate a number of math program instances, and associated solver sessions, where SymbolicMP is the symbolic mathematical program containing the MINLP model.

```
! Generate the MINLP model.
GMINLP := GMP::Instance::Generate(SymbolicMP, FormatString("%e", SymbolicMP)) ;

! Create NLP subproblem.
GNLP := GMP::Instance::Copy( GMINLP, 'OA_NLP' ) ;
GMP::Instance::SetMathematicalProgrammingType( GNLP, 'RMINLP' ) ;
ssNLP := GMP::Instance::CreateSolverSession( GNLP ) ;

! Create Master MIP problem.
GMIP := GMP::Instance::CreateMasterMip( GMINLP, 'OA_MasterMIP' ) ;
ssMIP := GMP::Instance::CreateSolverSession( GMIP ) ;

BasicAlgorithm;
```

The basic algorithm outlined above is available in the GMP Outer Approximation module as the procedure DoOuterApproximation.

The basic algorithm

The basic algorithm is straightforward, and makes a call to five other procedures that execute the various algorithm steps. The naming convention is self-explanatory, and the following lines make up this first example of a main procedure. For the sake of brevity and clarity, the parts of the code used to create a status file and to customize the contents of the progress window have been left out. They can be found in the basic implementation of the AOA algorithm in the AOA module.

```
InitializeAlgorithm;
SolveRelaxedMINLP;

while ( not MINLPAlgorithmHasFinished ) do
    AddLinearizationsAndSolveMasterMIP;
    FixIntegerVariablesAndSolveNLP;
```

```
    TerminateOrPrepareForNextIteration;
endwhile;
```

Note that the scalar parameter MINLPAlgorithmHasFinished must be initially set to zero, and should only get a nonzero value when the algorithm is ready to terminate.

Initialize-Algorithm

The following procedure is used to set all algorithmic parameters and options, and to prepare the status file and progress window output.

```
IterationCount               := 0 ;
LinearizationCount           := 1 ;
EliminationCount             := 1 ;
IncumbentSolutionHasBeenFound := 0 ;
MINLPAlgorithmHasFinished    := 0 ;

if ( NLPUseInitialValues ) then
    GMP::Solution::RetrieveFromModel( GNLP, SolNumbInitialValues ) ;
endif;

if ( GMP::Instance::GetDirection( GMINLP ) = 'maximize' ) then
    MINLPOptimizationDirection := 1;
else
    MINLPOptimizationDirection := -1;
endif;

GMP::Solution::SetProgramStatus( GMINLP, SolNumb, 'ProgramNotSolved' ) ;
GMP::Solution::SetSolverStatus( GMINLP, SolNumb, 'Unknown' ) ;

! The marginals of the NLP solver are needed.
option always_store_marginals := 'On';
```

The algorithmic parameters are initially set such that the AOA algorithm will always select the original initial values (i.e. the values of the variables prior to starting the AOA algorithm) as the starting values for each NLP subproblem to be solved. This setting has found to work quite well in extensive tests performed using this algorithm.

MINLPTerminate

The following termination procedure is used in several of the procedures that are described later.

```
if ( IncumbentSolutionHasBeenFound ) then
    GMP::Solution::SetProgramStatus( GMINLP, SolNumb, 'LocallyOptimal' ) ;
    GMP::Solution::SetSolverStatus( GMINLP, SolNumb, 'NormalCompletion' ) ;
else
    GMP::Solution::SetProgramStatus( GMINLP, SolNumb, 'LocallyInfeasible' ) ;
    GMP::Solution::SetSolverStatus( GMINLP, SolNumb, 'NormalCompletion' ) ;
endif;

GMP::Solution::SendToModel( GMINLP, SolNumb ) ;
MINLPAlgorithmHasFinished := 1 ;
```

The parameter IncumbentSolutionHasBeenFound contains a value of one or zero depending on whether the AOA algorithm has received an incumbent solution to the original MINLP model. Such a solution may be found when solving the

NLP subproblem, and this must then be communicated to the AOA algorithm. Note that you also need to set the program status and indicate when the MINLP algorithm has finished.

SolveRelaxed-MINLP

The first model that is solved during the algorithm is the relaxed MINLP model. All integer variables are relaxed to continuous variables. The following procedure implements this first solution step of the outer approximation algorithm.

```
SolveNLPSubProblem( 1 );
ProgramStatus := GMP::Solution::GetProgramStatus( GNLP, SolNumb ) ;

if ( ProgramStatus in NLPOptimalityStatus ) then
    ! Save NLP solution as MINLP solution if an integer solution has been found.

    if ( GMP::Solution::IsInteger( GNLP, SolNumb ) ) then

        ! Set incumbent solution for MINLP.
        GMP::Solution::RetrieveFromModel( GMINLP, SolNumb ) ;
        IncumbentSolutionHasBeenFound := 1 ;

        if ( TerminateAfterFirstNLPIsInteger ) then
            ! Terminate if an integer solution has been found.

            MINLPTerminate;
        endif;
    endif;
else
    ! Terminate if no linearization point has been found.

    SolverStatus := GMP::Solution::GetSolverStatus( GNLP, SolNumb ) ;

    if not ( SolverStatus in NLPContinuationStatus ) then
        MINLPTerminate;
    endif;
endif ;

IterationCount += 1 ;
GMP::Solution::SetIterationCount( GMINLP, SolNumb, IterationCount ) ;
```

When the procedure SolveNLPSubProblem has terminated, the AOA algorithm has typically found a point for the linearization step. The exception being when the NLP solver does not produce a solution at all (either feasible or infeasible). In such a situation the outer approximating algorithm should be terminated. Note that in the special event that the solution is feasible and has integer values for the integer variables, a locally optimal solution has been found and the AOA algorithm is instructed accordingly. Otherwise, the next step of the outer approximation algorithm can be executed.

Add-Linearizations-AndSolve-MasterMIP

If a termination flag has not been set, the following procedure adds linearizations to the master MIP problem prior to solving it. If this model becomes infeasible, the outer approximation algorithm will be terminated.

```
return when ( MINLPAlgorithmHasFinished );
```

```
GMP::Linearization::Add( GMIP, GNLP, SolNumb, AllNonLinearConstraints,
                         DeviationsPermitted, PenaltyMultiplier,
                         LinearizationCount, JacobianTolerance ) ;

LinearizationCount += 1 ;

GMP::SolverSession::Execute( ssMIP ) ;

GMP::Solution::RetrieveFromSolverSession( ssMIP, SolNumb ) ;
GMP::Solution::SendToModel( GMIP, SolNumb ) ;

ProgramStatus := GMP::Solution::GetProgramStatus( GMIP, SolNumb ) ;

if not ( ProgramStatus in MIPOptimalityStatus ) then
    MINLPTerminate;
endif ;
```

The AIMMS parameters DeviationsPermitted and PenaltyMultiplier are part of the AOA module. By default, deviations are allowed and are penalized with the value 1000 in the objective function of the master MIP.

FixInteger-VariablesAnd-SolveNLP

The following procedure implements the next major step of the outer approximation algorithm. First, the NLP subproblem is solved after fixing all the integer variables in the MINLP model using the values found from solving the previous master MIP problem. Then, if the combination of integer values and feasible NLP solution values improves the current MINLP incumbent solution, a new incumbent solution is set. When the NLP subproblem does not produce a solution (either feasible or infeasible), the outer approximation algorithm will be terminated.

```
return when ( MINLPAlgorithmHasFinished );

SolveNLPSubProblem( 0 );
ProgramStatus := GMP::Solution::GetProgramStatus( GNLP, SolNumb ) ;

if ( ProgramStatus in NLPOptimalityStatus ) then
    ! Save NLP solution as MINLP solution if no incumbent solution
    ! has been found yet, or if the NLP solution is better than
    ! the current incumbent.

    if ( not IncumbentSolutionHasBeenFound ) then
        ! Set incumbent solution for MINLP.

        GMP::Solution::RetrieveFromModel( GMINLP, SolNumb ) ;
        IncumbentSolutionHasBeenFound := 1 ;
    else

        NLPobjectiveValue   := GMP::Solution::GetObjective( GNLP  , SolNumb ) ;
        MINLPIncumbentValue := GMP::Solution::GetObjective( GMINLP, SolNumb ) ;

        if ( MINLPSolutionImprovement( NLPobjectiveValue, MINLPIncumbentValue ) ) then
            ! Set incumbent solution for MINLP.

            GMP::Solution::RetrieveFromModel( GMINLP, SolNumb ) ;
            IncumbentSolutionHasBeenFound := 1 ;
        endif;
    endif ;
```

```
else
    ! Terminate if no linearization point has been found.

    SolverStatus := GMP::Solution::GetSolverStatus( GNLP, SolNumb ) ;

    if not ( SolverStatus in NLPContinuationStatus ) then
        MINLPTerminate;
    endif;
endif ;
```

The AOA algorithm maintains the MINLP problem, the master MIP problem, the NLP subproblem, and the incumbent solution of the MINLP. As a result, direct access to the corresponding objective function values is available.

SolveNLPSubProblem

The procedure SolveNLPSubProblem solves the NLP subproblem using various routines from the GMP library. The procedure has a single argument initialSolve which indicates whether this is the solve of the initial relaxed MINLP problem. In that case some steps in the procedure are not necessary.

```
if ( NLPUseInitialValues ) then
    GMP::Solution::SendToModel( GNLP, SolNumbInitialValues ) ;
elseif ( not initialSolve ) then
    GMP::Solution::SendToModel( GMIP, SolNumb ) ;
endif;

GMP::Solution::RetrieveFromModel( GNLP, SolNumb ) ;
GMP::Solution::SendToSolverSession( ssNLP, SolNumb ) ;

if ( not initialSolve ) then
    GMP::Instance::FixColumns( GNLP, GMIP, SolNumb, AllIntegerVariables ) ;
endif;

GMP::SolverSession::Execute( ssNLP ) ;

GMP::Solution::RetrieveFromSolverSession( ssNLP, SolNumb ) ;
GMP::Solution::SendToModel( GNLP, SolNumb ) ;
```

TerminateOrPrepareForNextIteration

The following procedure implements the final major step of the outer approximation algorithm. If a termination flag has not been set previously, and the maximum number of iterations has not yet been reached, then the previously found integer solution of the master MIP problem will be eliminated by adding the appropriate cuts. This will ensure that the next master MIP will have a new integer solution (or none at all).

```
return when ( MINLPAlgorithmHasFinished );

if ( IterationCount = IterationMax ) then
    MINLPTerminate;
else
    ! Prepare for next iteration

    IterationCount += 1 ;
    GMP::Solution::SetIterationCount( GMINLP, SolNumb, IterationCount ) ;
    GMP::Instance::AddIntegerEliminationRows( GMIP, SolNumb, EliminationCount ) ;
    EliminationCount += 1 ;
endif ;
```

Note that you are responsible for determining the appropriate iteration count for the overall outer approximation algorithm. As you are free to develop a solution algorithm in any way you desire, it is not always possible for the AOA algorithm to determine the correct setting of the MINLP iteration count.

18.7 Alternative uses of the open approach

Customize algorithm

Using the open outer approximation approach for solving MINLP models it is possible to add to the existing procedures or write alternative procedures to meet the needs of the final user. For instance, a user evaluating the performance of the algorithm may want to add certain performance measurements and print statements to the existing code. Some less trivial examples of modifications are provided in the next few paragraphs.

Solve more NLPs

Practical experience has shown that it is sometimes difficult to get a feasible solution to the initial relaxed NLP model. Based on the particular application, the user may specify how multiple starting values can be found, and then modify the algorithm to solve multiple NLPs to get a feasible and/or a better solution. While doing so, it is also possible to specify how the algorithm should switch between different solvers (using the predefined AIMMS identifier `CurrentSolver`). Such extensions could then also be applied to the NLP subproblem inside the `While` statement.

Retain integer solutions

It is possible to activate a MIP callback procedure whenever the MIP solver finds an integer solution. Even though these intermediate solutions are not optimal, the user may want to save the integer portion of these solutions for later evaluation. Once the main algorithm has terminated, all these integer solutions can be retrieved and evaluated by solving the corresponding nonlinear subproblem. In some instances, one of these extra solutions may be a better solution to the original MINLP model than the one produced by the main algorithm.

Adjust penalties

Setting the penalties for the deviations of the linear approximation constraints in the master MIP subproblem is a delicate manner, and has an effect on the solution quality when the nonlinear subproblems are nonconvex. The user can consider several problem-dependent strategies to adjust the penalty values, and implement them inside the basic AOA algorithm.

Example of modified procedure

The following procedure is a variant of the termination procedure provided in the previous section. Assuming that the two parameters that refer to the previous and current NLP objective function values have been properly set in the procedure that solves the NLP subproblem, then termination is invoked whenever there is insufficient progress between two subsequent NLP solutions, or between the objective values of the master MIP problem and the current

NLP subproblem. The third termination criterion is the number of iterations reaching its maximum.

```
return when ( MINLPAlgorithmHasFinished );

if    (not MINLPSolutionImprovement( NLPCurrentObjectiveValue,
                                     NLPPreviousObjectiveValue ))
   or (not MINLPSolutionImprovement( GMP::Solution::GetObjective(GMINLP, SolNumb),
                                     NLPCurrentObjectiveValue ))
   or ( IterationCounter = IterationMax ) then

    MINLPTerminate;

else
    ! Prepare for next iteration

    IterationCount += 1 ;
    GMP::Solution::SetIterationCount( GMINLP, SolNumb, IterationCount ) ;
    GMP::Instance::AddIntegerEliminationRows( GMIP, SolNumb, EliminationCount ) ;
    EliminationCount += 1 ;
endif ;
```

Conclusion

The above paragraphs indicate just a few of the ways in which you can alter the basic implementation of the outer approximation algorithm in AIMMS. Of course, it is not necessary to develop your own variant. Whenever you need to solve a MINLP model using the AOA algorithm, you can simply call the basic implementation described in the previous section. As soon as you can see improved ways to solve a particular model, you can apply your own ideas by modifying the procedures as you see fit.

Chapter 19

Stochastic Programming

Deterministic vs. stochastic models

The mathematical programming types discussed so far have a common assumption that all the input data used in the formulation of the mathematical program is known with certainty. This is known as "decision making under certainty," and the corresponding models are called *deterministic* models. Models that account for uncertainty in the input data are called *stochastic* models, and the theory and techniques used to solve stochastic models is commonly referred to as stochastic programming. You can find an introduction to stochastic programming in Chapters 16 and 17 of the AIMMS Optimization Modeling Guide. A more in-depth discussion of stochastic programming and its solution methods can be found, for instance, in [Bi97] and [Ka05].

Stochastic programming in AIMMS

In this chapter, you will find a description of the facilities built into AIMMS for creating and solving stochastic models. From any existing deterministic linear (LP) or mixed-integer (MIP) model, AIMMS is able to automatically create a stochastic model as well, *without the need for you to reformulate any of the constraint definitions*. The only steps necessary to create a stochastic model are

- to indicate which parameters and variables in your deterministic model are to become stochastic in a declarative manner, and
- to provide the scenario tree and the stochastic input data.

Single formulation

Being able to generate both a deterministic and stochastic model from an identical symbolic formulation allows for any changes you make in the deterministic formulation to automatically propagate to the stochastic model. This significantly reduces the effort involved with maintaining a stochastic model associated with a given deterministic model.

This chapter

Section 19.1 discusses a number of basic concepts in stochastic programming. These provide a common understanding necessary for the introduction of the stochastic programming facilities of AIMMS discussed in Section 19.2. Section 19.3 describes the facilities available in AIMMS for the generation of a scenario tree, while Section 19.4 discusses the steps necessary to solve a stochastic model in AIMMS.

19.1 Basic concepts

Basic concepts

In this section you will find a number of basic concepts that are commonly used in stochastic programming. They will help you to unambiguously understand the stochastic programming facilities in AIMMS.

Stages

In stochastic programming *stages* define a collection of consecutive periods of time. Stages are usually identified through positive integers $1, 2, \ldots$, and are characterized as follows:

- during each stage one or more stochastic (i.e. uncertain) events take place, and
- at the end of a stage decisions are taken, taking into account the specific outcomes of the stochastic events of this and previous stages.

Stochastic events may be such quantities as the demand realized during a period. They are usually represented as input data used in the deterministic model. Any variables in the deterministic model that are modeled conceptually to take their value *at the beginning* of a period (for instance, the stock at the beginning of a period), should be considered as decisions taken at the end of the previous period/stage within the stage concept.

Stages versus model periods

If the deterministic model already contains a `Horizon` (see Section 33.3) or any other set of time periods, the stages of the stochastic model may naturally coincide with the time periods from the deterministic model, but this certainly needs not be the case. A single period model, possibly even without an explicit period set, but with variables representing decisions taken at the beginning *and* at the end of the period, may still constitute a two-stage stochastic model. For a multi-period model, a single stage in the stochastic model may consist of multiple time periods from the deterministic model, and hence one can always construct a mapping from the deterministic period set to stages in the stochastic model.

Variables and stages

Every individual stochastic variable in a stochastic model should be uniquely associated with a single stage. This stage represents the period at the end of which the variable conceptually takes it value

- taking into consideration the *specific* outcomes of stochastic events taking place during the stage at hand and during prior stages,
- but only taking into account the *distribution* of possible outcomes of the stochastic events taking place in any further stage.

Even if model periods of the deterministic model and the stages of the stochastic model coincide, variables with an index into the period set do not have to be associated with the stage corresponding to the value of that index. As discussed above, variables that conceptually take their value at the beginning of a

period, provide a first example of this behavior as they must be associated with the stage corresponding to the previous period within the stochastic model.

Examples

Other examples may arise, for instance, when the monthly productions of January, February and March should be decided upon prior to the beginning of January, regardless of the specific outcomes for the demands during these months. Conversely, if market research has delivered good estimates for the demand in January, February and March, the decisions for the production in these months should take into consideration the demand estimates of all three months. Hence the production variables for January, February and March should be part of the stage associated with March.

Scenarios

A *scenario* for a stochastic model is a collection of outcomes for all the stochastic events taking place in the model, along with the associated probability of the scenario to occur. For the event values associated with each scenario, one could solve a deterministic model, which would yield the optimal decisions for that particular scenario. For different scenarios, however, the decisions resulting from solving such deterministic models individually are, in general, completely unrelated, even if the event outcomes of the scenarios are exactly the same up to a certain stage. To address this problem, the scenarios of a stochastic model must be organized into a scenario tree.

Scenario trees

A single scenario can be graphically represented as a simple tree illustrated in Figure 19.1.

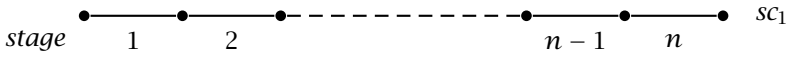


Figure 19.1: A tree representing a single scenario sc_1

For multiple scenarios, the specific outcomes of the stochastic events up to a certain stage usually coincide for a subset of the scenarios. This gives rise to a scenario tree as illustrated in Figure 19.2. In such a scenario tree, the path from the root node of the tree to each of its leaf nodes corresponds to a single scenario, and the event outcomes for scenarios that pass through the same intermediate node are identical for all stages up to that node. If a stage consists of multiple time periods in the deterministic model, this means that the stochastic events taking place during *all* periods associated with the stage should coincide. The solution process of a stochastic model will make sure that the decisions that are to be taken at the end of these stages are identical for all the scenarios passing through the node, as one would intuitively expect.

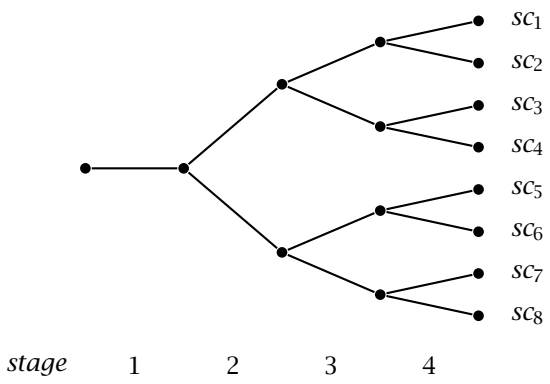


Figure 19.2: A scenario tree with 8 scenarios

Scenario generation

The scenarios and the scenario tree used in a stochastic model are usually generated by using one of the following two techniques

- generate scenarios by incrementally creating a scenario tree according to a given distribution for each stochastic event, or
- given an externally created set of scenarios, create a scenario tree by grouping identical or similar scenarios at every level of the tree.

Distribution-based scenario generation

Given a leaf node in an intermediate scenario tree, for every stochastic event that occurs during the stage directly following that node, a fixed number of values is computed according to a given distribution (each with its own relative probability of taking place). For each of these values a new branch is added to the node. The process starts by adding branches to the root node of the tree and ends when a tree is generated for all stages. The total number of scenarios generated by the process is the final number of leaf nodes generated. The probability of a scenario is the multiplication of the relative probabilities associated with each branch along the path from root to leaf node. The scenario tree in Figure 19.2 could be generated in this way, for example, by choosing, at every intermediate node, a *high* or a *low* level for the demand during the stage following that particular node.

Scenario-based tree generation

Another approach is to start from a given collection of scenarios with probabilities adding up to 1. Such a collection of scenarios can either be randomly generated or be the result of some external process. As a tree, they can be represented as a trivial scenario tree, as illustrated in Figure 19.3. This tree can be transformed into a scenario tree by bundling together identical or similar scenarios into a fixed or dynamic number of branches. The group of scenarios passing through a particular intermediate node in the scenario tree is analyzed and grouped into subgroups of scenarios with similar or identical outcomes of the stochastic events during the stage following that node. For every subgroup, the existing branches are bundled into a single branch, and

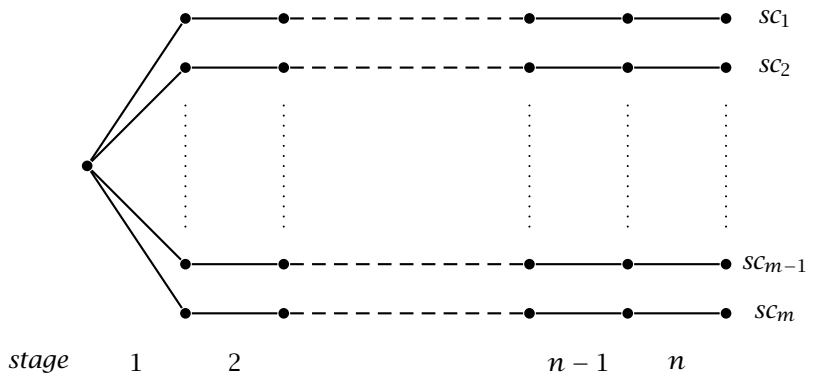


Figure 19.3: An initial disconnected scenario tree

the stochastic event outcomes are made identical for all scenarios in the subgroup. The process starts by analyzing all scenarios at the root node of the tree, and ends when every scenario is associated with a single leaf node.

Basic procedure for solving stochastic models

The implementation of stochastic programming in AIMMS closely follows the concepts described in this section. The basic procedure to create and solve a stochastic model in AIMMS is as follows:

- indicate in your model which parameters and variables are to become stochastic,
- for every stochastic variable in your model specify during which stage of the stochastic model the decision is to be taken,
- generate scenarios, their stochastic data, and a scenario tree, using one of the techniques described above, and
- generate and solve the stochastic model using the special methods available for this purpose in AIMMS.

Each of these steps is explained in more detail in the sections to follow. Note that changing parameters and variables in your model into stochastic parameters and variables, does in no way influence the possibility to solve the underlying deterministic model in its original form. Thus, the stochastic programming facilities in AIMMS always form a true extension of the functionality of the existing AIMMS application.

19.2 Stochastic parameters and variables

The set AllStochasticScenarios

To allow the storage of scenario-dependent parameter and variable data for multiple scenarios in a stochastic model, all such scenarios should be added to the predefined set AllStochasticScenarios. If your application contains multiple stochastic models—each with different scenario sets—the set AllStochasticScenarios should be the union of all these scenario sets. For each stochastic model you can then define an associated subset of AllStochasticScenarios to use with that particular stochastic model.

Stochastic parameters

Stochastic events are modeled in AIMMS as numeric Parameters for which the Stochastic property has been set (see also Section 4.1). For stochastic parameters AIMMS provides an additional .Stochastic suffix, which you can use to store scenario-dependent stochastic event outcomes. The data stored in the suffix is used by AIMMS when generating the stochastic model. The index domain of the .Stochastic suffix is, therefore, the set AllStochasticScenarios plus the original domain of the parameter.

Example

Consider the following declarations

```
Set MyScenarios {
    SubsetOf     : AllStochasticScenarios;
    Index        : sc;
}
Parameter Demand {
    IndexDomain  : (c,t);
    Property     : Stochastic;
}
```

These declarations will cause AIMMS to create a .Stochastic suffix for the parameter Demand(c,t). To use, or assign values to, Demand.Stochastic, you must use an additional index into (a subset of) AllStochasticScenarios. The following statement provides an example of such a statement.

```
Demand.Stochastic(sc,c,t) := Uniform(10,20);
```

If a constraint contains a reference to the parameter Demand, AIMMS will use the data in Demand.Stochastic to generate the appropriate demand constraint for every scenario.

Stochastic variables

By setting the Stochastic property for a Variable in your model, you indicate to AIMMS that this variable may have multiple, scenario-dependent, solutions when used in a stochastic model. Consequently, when generating a matrix for the stochastic model, a column will be generated conceptually for every single scenario.

The Stage attribute

For stochastic variables you must also specify the mandatory Stage attribute. Through the Stage attribute you specify the stage at the end of which the decision corresponding to the stochastic variable is to be taken. The value of the Stage attribute must be an explicit positive integer value, or a parameter reference involving some or all of the indices on the index list of the declared variable.

Non-anticipativity constraints...

As discussed in the previous section, for every scenario s_0, a stochastic variable x gets its value x_{s_0} at the end of stage n as specified in the Stage attribute of the variable. In addition, its value is based on the specific outcomes of the stochastic events for that scenario taking place during stages $1, \ldots, n$, but only on the distribution of the stochastic event outcomes for any further stages. Therefore, the value x_s must be equal to x_{s_0} for every other scenario s that passes through the same node in the scenario tree at the end of stage n as s_0. The constraints enforcing this equality are called *non-anticipativity constraints*—they do not allow the solution to anticipate on stochastic outcomes that lie beyond the stage as specified by the Stage suffix.

... enforced explicitly or implicitly

When generating a stochastic model, AIMMS will automatically enforce the non-anticipativity constraints, either by explicitly adding them to the generated matrix, or implicitly by substituting a single representative x_{s_0} for every other variable x_s. While enforcing non-anticipativity in an implicit manner will drastically reduce the matrix size, an explicit representation may be helpful for solvers able to decompose the generated matrix.

Non-stochastic variables

If a variable in a stochastic model has not been declared stochastic, it is deterministic in the sense that it assumes the same value for every scenario, as is the case with first stage variables.

The .Stochastic suffix for variables

Variables can also have a .Stochastic suffix in AIMMS. It follows the same rules for its index domain as the .Stochastic suffix of parameters. AIMMS uses the .Stochastic suffix of variables to store the solution data of a stochastic model after solving it. However, contrary to stochastic parameters, AIMMS will not only create the .Stochastic suffix for stochastic variables, but for *all* variables that are involved in a stochastic model.

Contents of .Stochastic suffix

The values stored in the .Stochastic suffix after solving a stochastic model for each type of variable are as follows:

- for stochastic variables, the .Stochastic suffix will contain the solution of the variable for each scenario,
- for the objective variable, the .Stochastic suffix will contain the contribution to the objective of each scenario, as well as the weighted objective value of the stochastic model itself,

- for any other non-stochastic variable, the .Stochastic suffix will contain the deterministic solution of that variable for the stochastic model.

As the solution of a stochastic model is entirely stored in the .Stochastic suffix, the solution of the underlying deterministic model remains completely intact after solving the stochastic model. This makes it easy to visually, and/or programmatically, compare the solutions of the deterministic and stochastic model.

Non-stochastic solution data

As the objective value and solution of the non-stochastic variables of the stochastic model cannot be coupled directly with one specific scenario in the scenario set, AIMMS creates an extra element in the set AllStochasticScenarios for this purpose. You must specify the name of this element when solving the stochastic model (see also Section 19.4).

19.3 Scenario generation

Scenario generation

To support you in creating scenarios and a scenario tree, AIMMS provides a system module which provides a customizable scenario generation framework. For each of the two basic methods for scenario generation discussed in Section 19.1, the module contains a generic procedure to implement that method. To use these scenario generation procedures to generate scenarios and/or a scenario tree, you only have to implement some callback procedures to supply the necessary data for your specific stochastic model.

Importing the system module

To import the generation module into your model, select **Install System Module...** from the **Settings** menu, and select the **Scenario Generation Module** from the dialog box that appears. The module will be added at the end of the model tree of your model. By default, the module prefix of the **Scenario Generation Module** is ScenGen.

19.3.1 Distribution-based scenario generation

Distribution-based scenario generation

The basic procedure in the scenario generation module for distribution-based scenario generation is

- CreateScenarioTree(*Stages*, *Scenarios*, *ScenarioProbability*, *ScenarioTreeMapping*).

Input arguments

The procedure has a single input argument:

- the set of *Stages* in your stochastic model. This set must be a subset of the predefined set Integers.

Output arguments

The outputs of this procedure are:

- the set of *Scenarios* (which must be a subset of AllStochasticScenarios) generated by the procedure,
- the *ScenarioProbability*, a one-dimensional parameter indexed over the set *Scenarios*, and
- the *ScenarioTreeMapping*, a two-dimensional element parameter defined over *Scenarios*×*Stages* to *Scenarios*, providing a mapping from every scenario during every stage to a single representative scenario for the scenario bundle in which the given scenario is contained during this stage.

Distribution-based callback functions

The contents of the outputs of the procedure CreateScenarioTree is completely based on the results of the problem-specific callbacks that you have to supply. The following callbacks are expected by CreateScenarioTree:

- InitializeNewScenarioCallback(*CurrentStage*, *Scenario*, *RepresentativeScenario*),
- InitializeStochasticDataCallback(*CurrentStage*, *Scenario*, *ChildBranch*, *ChildBranchName*), and
- InitializeChildBranchesCallback(*CurrentStage*, *Scenario*, *ChildBranches*, *ChildBranchNames*).

Initializing a new scenario

When building up the scenario tree, AIMMS creates new scenarios on the fly. In order for you to refer to data from previous stages for this scenario, AIMMS will call the callback InitializeNewScenarioCallback for every *Stage* prior to the current stage, and supply the *RepresentativeScenario* from the scenario bundle for *CurrentStage* which also contains the newly created *Scenario*. By copying the stochastic data for this stage from this representative scenario, you make it available both to you and AIMMS. To properly generate the stochastic model, AIMMS needs the stochastic parameter values for every stage and every scenario.

Supplying stochastic event data

In the procedure InitializeStochasticDataCallback you can provide values to all stochastic parameter values for the *ChildBranch* during *CurrentStage* for the *Scenario*. Because AIMMS has called the InitializeNewScenarioCallback prior to calling InitializeStochasticDataCallback you also have access to the stochastic parameter values of this scenario prior to the current stage. Based on the value of *ChildBranch* and the prior stochastic parameter values, you should have sufficient information to generate new stochastic parameter values for the current stage. You should pass the relative weight of this branch compared to the other child branches through the return value of the callback. Note that the relative weights you return may, but need not, add up to one. After creating scenarios for all branches, AIMMS will scale the sum of the returned relative weights of all branches to one.

Generating new child branches

Finally, to extend the scenario tree to a next stage, AIMMS calls the callback InitializeChildBranchesCallback. In this callback, you should fill the Integer subset *ChildBranches* with integers $1, 2, \ldots$ for every child branch that you want to add to *Scenario* at *CurrentStage*. Through the element parameter *ChildBranchNames* you should provide a short representative name for every child branch (for instance, "H" and "L" when child branches represent high and low demand). From the branch names you supply, AIMMS will generate the full names of the final element names of the scenarios generated by the scenario generation procedure (for instance '[H,L,H,H,L]' for a scenario with high, low, high, high, and low demand values during the successive stages of the scenario).

Setting the callbacks

The scenario generation module contains templates for each of the callbacks described above. Rather than changing these template callbacks in the module, you are advised to copy the template callbacks to your core model, and change the bodies of the copied callbacks. Finally, you should notify AIMMS of the names of your callback functions by, prior to calling the procedure CreateScenarioTree, assigning the names of your callback procedures to the element parameters

- ScenGen::InitializeNewScenarioCallbackFunction,
- ScenGen::InitializeStochasticDataCallbackFunction, and
- ScenGen::InitializeChildBranchesCallbackFunction.

Example

The following callbacks will cause the procedure CreateScenarioTree to generate a tree with 2 branches "H" and "L" (for high and low demand) at every intermediate node, and initialize Demand.Stochastic for every period. The example assumes the existence of a mapping PeriodToStage(st,t).

Initializing a new scenario

To initialize a new scenario, we have to copy the stochastic demand data for the newly created *Scenario* during *Stage* from the scenario *RepresentativeScenario*. Thus, the body of the InitializeNewScenarioCallback would read

```
for ( t | PeriodToStage(CurrentStage,t) ) do
    Demand.Stochastic(Scenario,t) := Demand.Stochastic(RepresentativeScenario,t);
endfor;
```

Generating new child branches

To generate two child branches to any intermediate node in the scenario tree representing high ("H") and low ("L") demand, the implementation of the InitializeChildBackBranchesCallback should be

```
ChildBranches := { 1, 2 };
ChildBranchNames('1') := "H";
ChildBranchNames('2') := "L";
```

Initializing stochastic demand

For each newly added child branches, the following implementation of `InitializeStochasticDataCallback` assigns a high (20) or low (10) stochastic demand value to the *Scenario* during the *CurrentStage*

```
for ( t | PeriodToStage(CurrentStage,t) ) do
    Demand.Stochastic(Scenario,t) := if ( ChildBranch = 1 ) then 20 else 10 endif;
endfor;

return 1;
```

By returning 1 for all branches, we just indicate that every branch has equal relative weight. For two branches, this will result in a relative probability for each branch of 0.5.

19.3.2 Scenario-based tree generation

Scenario-based tree generation

The basic procedure in the scenario generation module for scenario-based tree generation is

- `CreateScenarioData`(*Stages*, *Scenarios*, *ScenarioProbability*, *ScenarioTreeMapping*).

Input arguments

The procedure has a single input argument:

- the set of *Stages* in your stochastic model. This set must be a subset of the predefined set `Integers`.

Output arguments

The outputs of the procedure are:

- the set of *Scenarios* for which you have provided stochastic parameter values,
- the *ScenarioProbability*, a one-dimensional parameter indexed over the set *Scenarios*, and
- the *ScenarioTreeMapping*, a two-dimensional element parameter defined over *Scenarios*×*Stages* to *Scenarios*, providing a mapping from every scenario during every stage to a single representative scenario for the scenario bundle in which the given scenario is contained during this stage.

Algorithm outline

The procedure `CreateScenarioData` will help you construct a scenario tree as follows:

- initially, AIMMS will request you to generate a set of scenarios with their relative weights,
- next, AIMMS will ask you, to divide a given group of scenarios at the current stage into a number of subgroups of equal or similar scenarios at the next stage,

- AIMMS will request you to reassign a single unique value to each stochastic event parameter for all scenarios in a scenario group (e.g. the mean over all scenarios in the group), and
- finally, AIMMS will remove scenarios which you identify as identical.

Scenario-based callbacks

For each of the steps outlined in the previous paragraph, you must supply a callback procedure:

- `InitializeStochasticScenarioDataCallback`(*Scenario*, *Scenarios*),
- `DetermineScenarioGroupsCallback`(*CurrentStage*, *ScenarioGroup*, *ScenarioGroupOrder*),
- `AssignStochasticDataForScenarioGroupCallback`(*CurrentStage*, *ScenarioGroup*), and
- `CompareScenariosCallback`(*Scenario1*, *Scenario2*, *Stages*, *FirstDifferentStage*)

Initializing scenarios

Through the `InitializeStochasticScenarioDataCallback` you must supply the stochastic event data during all stages for a *Scenario* generated by AIMMS. The function should return the relative weight of the scenario compared to all other scenarios you supply. If you are done adding scenarios, the callback should return the value 0.

Dealing with existing scenario data

If you have already read scenario data from a database, for instance, you can overwrite the generated value of *Scenario* argument with an existing scenario name read from the database. In that case, if you have read the stochastic data directly into the `.Stochastic` suffix of the stochastic parameters in your model, you only have to return the relative weight.

Supplying new scenario data

If you do not have existing scenario data, you should generate stochastic data for the *Scenario* element generated by AIMMS for all stochastic parameters in your model. If you want to change the name of the generated *Scenario*, you can do so using the function `SetElementRename`.

Creating scenario subgroups

In the `DetermineScenarioGroupsCallback`, you must divide the scenarios in *ScenarioGroup* created during a previous stage (or the group of all scenarios to start with during the first stage) into subgroups, based on the equality or similarity of the stochastic event values associated with the scenarios during *CurrentStage*. You must specify the subgroups by assigning a *ScenarioGroupOrder* to every scenario in the *ScenarioGroup*, where scenarios with the same assigned order form a subgroup during the current stage.

Assigning stochastic event values

If the stochastic event parameters in *ScenarioGroup* during *CurrentStage* are similar, but not equal, you must make sure to assign identical event parameter values to every scenario when AIMMS calls the AssignStochasticDataForScenarioGroupCallback. Failure to do so may result in infeasible stochastic models generated by AIMMS.

Removing identical scenarios

Finally, AIMMS will probe for identical scenarios through the CompareScenarioCallback, remove duplicate scenarios when encountered, and adjust the scenario probabilities accordingly. When the stochastic event values of *Scenario1* and *Scenario2* are identical during *Stages*, the callback should return 0. If the scenarios are not identical the callback should have a nonzero return value, and set the output argument *FirstDifferentStage* equal to the first stage during which the event parameters differ for both scenarios.

Setting the callbacks

The scenario generation module contains templates for each of the callbacks described above. Rather than changing these template callbacks in the module, you are advised to copy the template callbacks to your core model, and change the bodies of the copied callbacks. Finally, you should notify AIMMS of the names of your callback functions by, prior to calling the procedure CreateScenarioData, assigning the names of your callback procedures to the element parameters

- ScenGen::InitializeStochasticScenarioDataCallbackFunction,
- ScenGen::DetermineScenarioGroupsCallbackFunction,
- ScenGen::AssignStochasticDataForScenarioGroupCallbackFunction, and
- ScenGen::CompareScenariosCallbackFunction.

Example

The callbacks for scenario-based tree generation, are usually more problem-specific, and hence less instructive, than the callbacks for the tree-based scenario generation scheme. Therefore, rather than including a lengthy example here, we refer to the example models for stochastic programming that come with your AIMMS system.

Scenario generation can be modified

The scenario generation module is completely implemented in the AIMMS language itself, and contains basic implementations of both scenario generation methods, which will provide a good starting point for most stochastic models. If neither of these implementations fits your needs, you can copy the module to your project directory, replace the system module with the copy, and make the algorithms in the copied module more advanced to better fit the needs of your stochastic model.

19.4 Solving stochastic models

Solving stochastic models

After generating stochastic event data and a scenario tree, you can generate and solve the stochastic model by using methods from the GMP library discussed in Chapter 16. AIMMS supports two methods for solving a stochastic model:

- by solving its *deterministic equivalent*, or
- for purely linear mathematical programs only, through the *stochastic Benders algorithm*, or
- using the Benders decomposition algorithm in CPLEX 12.7 or higher.

Both Benders algorithms will decompose the stochastic model into multiple smaller models, and thus is better suited to solve stochastic models where the deterministic equivalent, either by the size of the deterministic model or because of a huge number of scenarios, becomes too big or time-consuming to solve at once. The Benders decomposition algorithm in CPLEX can be used to solve stochastic models with integer variables, as long as all integer variables are assigned to the first stage. For more information see the CPLEX option `Benders_strategy`.

19.4.1 Generating and solving the deterministic equivalent

Generating a stochastic model

The method for generating a stochastic model for a `MathematicalProgram` *MP* is

- `GMP::Instance::GenerateStochasticProgram(`
  MP, *StochasticParameters*, *StochasticVariables*,
  Scenarios, *ScenarioProbability*, *ScenarioTreeMap*,
  DeterministicScenarioName[, *GenerationMode*][, *Name*])

The function returns an element into the set `AllGeneratedMathematicalPrograms`. This generated math program instance contains a memory-efficient representation of the technology matrix of the stochastic model and the stochastic event data, and can be used to create a deterministic equivalent of the stochastic model, as well as the submodels necessary for a stochastic Benders approach.

Specifying stochastic identifiers

Through the arguments *StochasticParameters* and *StochasticVariables* you indicate to AIMMS which stochastic parameters and variables you want to take into consideration when generating this stochastic model. These arguments must be subsets of the predefined sets `AllStochasticParameters` and `AllStochasticVariables`, respectively. You may want to use real subsets, for instance, when your AIMMS project contains multiple stochastic models, each referring only to a subset of the stochastic parameters and variables.

Specifying scenarios

Through the *Scenarios*, *ScenarioProbability* and *ScenarioTreeMap* arguments you specify the set of scenarios, their probabilities and the mapping defining the scenario tree for which you want to generate the stochastic model to AIMMS. Through the string argument *DeterministicScenarioName*, you supply the name of the artificial element that AIMMS will add to the predefined set `AllStochasticScenarios` (if not created already), and use to store the solution of non-stochastic variables in their respective `.Stochastic` suffices as explained in Section 19.2.

Enforcing non-anticipativity constraints

Using the *GenerationMode* argument you can specify whether you want AIMMS to explicitly add the non-anticipativity constraints to your stochastic model, or whether you want non-anticipativity to be enforced implicitly by substituting the representative scenario for every non-representative scenario at every stage. *GenerationMode* is an element parameter into the predefined set `AllStochasticGenerationModes`, with possible values

- `'CreateNonAnticipativityConstraints'`, and
- `'SubstituteStochasticVariables'` (the default value).

Name argument

With the optional *Name* argument you can explicitly specify a name for the generated mathematical program. If you do not choose a name, AIMMS will use the name of the underlying `MathematicalProgram` as the name of the generated mathematical program as well. Please note, that AIMMS will also use this name as the default name for solving the deterministic model. Therefore, if you do not want the generated mathematical program of the deterministic model to be deleted, then you have to choose a non-default name.

Solving the deterministic equivalent of a stochastic model

You can solve a stochastic model by using the regular GMP procedure

- `GMP::Instance::Solve`(*gmp*)

By applying this function to a generated mathematical program associated with a stochastic model, AIMMS will create the deterministic equivalent and pass it to the appropriate LP/MIP solver. The `GMP::Instance::Solve` method is discussed in full detail in Section 16.2.

Changing the model input

Note that, when you adjust the scenario tree map, the stochastic data, the scenario probabilities, or the value of the `Stage` attribute of some variables after you generated the stochastic model, you should regenerate the stochastic model again to reflect these changes.

Example

Consider the following call to `GMP::Instance::GenerateStochasticProgram`

```
GMP::Instance::GenerateStochasticProgram(
    TransportModel, AllStochasticParameters, AllStochasticVariables,
    MyScenarios, MyScenarioProbability, MyScenarioTreeMap,
    "TransportModel", 'SubstituteStochasticVariables', "StochasticTransportModel");
```

After solving the generated stochastic model, its solution will be stored as follows, where sc is an index into MyScenarios

- the per-scenario solution of a stochastic variable Transport(i,j) will be stored in Transport.Stochastic(sc,i,j),
- the deterministic solution of a non-stochastic variable InitialStock(i) will be stored in InitialStock.Stochastic('TransportModel',i),
- the weighted objective value for the objective variable TotalCost will be stored in TotalObjective.Stochastic('TransportModel'), while the contribution by every scenario is available through TotalCost.Stochastic(sc).

19.4.2 Using the stochastic Benders algorithm

Using the stochastic Benders algorithm

Instead of solving the deterministic equivalent of a stochastic model, AIMMS also allows you to solve *linear* stochastic models using a stochastic Benders algorithm. The stochastic Benders algorithm is based on a reformulation of the original model as a sequence of models outlined below. The solution of the original model can be achieved by solving the sequence of models iteratively until a terminating condition is reached. A more detailed discussion of the stochastic Benders algorithm can be found in [De98] or [Al03].

Definitions

All nodes in the scenario tree are numbered starting at 1 (the root node).

Indices:

i	*index for the set of nodes N*
t	*index for the set of stages T*

Parameters:

q_i	*probability belonging to node i*
p_i	*parent of node i*

Sets:

I_i	*set with children of node i*
N_t	*set of nodes belonging to stage t*

Convention

In the algorithmic outline below we identify the problem names with their associated solutions. That is, if a problem is, for instance, identified as $F_i(x_{p_i})$, we will also use this name to denote its solution in other sub- problems.

The original model

The nested Benders algorithm can be used for problems of the form

Minimize:

$$\sum_{t \in T} \sum_{i \in N_t} q_i c_i^T x_i$$

Subject to:

$$\begin{aligned} W_1 x_1 &= h_1 \\ A_i x_{p_i} + W_i x_i &= h_i \qquad \forall\, i \in N_t, t \in T \setminus \{1\} \\ x_i &\geq 0 \qquad \forall\, i \in N_t, t \in T \end{aligned}$$

A reformulation as a sequence of models

This problem corresponds to the following sequence of problems. For node $i = 1$, the problem F_1 is defined as

Minimize:

$$c_1^T x_1 + \sum_{j \in I_1} q_i F_j(x_1)$$

Subject to:

$$W_1 x_1 = h_1$$
$$x_1 \geq 0$$

For all other nodes $i \in N_t$ in stage $t \in T \backslash \{1\}$, the problem $F_i(x_{p_i})$ is defined as follows (note that $\sum_{j \in I_i} q_j = q_i$)

Minimize:

$$c_i^T x_i + \sum_{j \in I_i} \frac{q_j}{q_i} F_j(x_i)$$

Subject to:

$$W_i x_i = h_i - A_i x_{p_i}$$
$$x_i \geq 0$$

For the leaf nodes in the scenario tree, the term $\sum_{j \in I_i} \frac{q_j}{q_i} F_j(x_i)$ is omitted.

Formulated differently

If we now introduce an upper bound θ_i to replace the term $\sum_{j \in I_i} \frac{q_j}{q_i} F_j(x_i)$, we can rewrite the subproblem $F_i(x_{p_i})$ as

Minimize:

$$c_i^T x_i + \theta_i$$

Subject to:

$$W_i x_i = h_i - A_i x_{p_i}$$
$$\theta_i \geq \sum_{j \in I_i} \frac{q_j}{q_i} F_j(x_i)$$
$$x_i \geq 0$$

Because of the linear nature of the original problem, the terms $\sum_{j \in I_i} \frac{q_j}{q_i} F_j(x_i)$ are piecewise linear and convex. Therefore there exists an (a priori unknown) set of equations

$$D_i^l x_i = d_i^l$$

that describes such a term and for which

$$D_i^l x_i + \theta_i \geq d_i^l.$$

Moreover, we are only interested in those x_i such that $F_j(x_i)$ are feasible for all $j \in I_i$. This requirement can be enforced by an (a priori unknown) set of constraints

$$E_i^l x_i \geq e_i^l.$$

By substituting these constraints we obtain the following reformulation of problem $F_i(x_{p_i})$

Minimize:

$$c_i^T x_i + \theta_i$$

Subject to:

$$\begin{aligned} W_i x_i &= h_i - A_i x_{p_i} \\ D_i^l x_i + \theta_i &\geq d_i^l && \forall l \in 1, \ldots, R_i \\ E_i^l x_i &\geq e_i^l && \forall l \in 1, \ldots, S_i \\ x_i &\geq 0 \end{aligned}$$

The relaxed master problem

The actual problem that is solved at node i is the following relaxed master problem $\tilde{N}_i(x_{p_i})$ defined as follows:

Minimize:

$$c_i^T x_i + \theta_i$$

Subject to:

$$\begin{aligned} W_i x_i &= h_i - A_i x_{p_i} \\ D_i^l x_i + \theta_i &\geq d_i^l && \forall l \in 1, \ldots, r_i \\ E_i^l x_i &\geq e_i^l && \forall l \in 1, \ldots, s_i \\ x_i &\geq 0 \end{aligned}$$

At the start of the Benders algorithm r_i and s_i will be 0 for all $i \in N$. The constraints $D_i^l x_i + \theta_i \geq d_i^l$ are optimality cuts obtained from the children. That is, if $\tilde{N}_j(x_i)$ is feasible for all $j \in I_i$ (but not optimal) then an optimality cut is added to $\tilde{N}_i(x_{p_i})$. The optimality cut is constructed by using a combination of the dual solutions of $\tilde{N}_j(x_i)$ for all $j \in I_i$. Adding an optimality cut does not make a feasible relaxed master problem infeasible. The Benders algorithm fails if one of the subproblems is unbounded. This can be avoided by giving all variables, except the objective variable, finite bounds.

Adding feasibility cuts

The constraints $E_i^l x_i \geq e_i^l$ are feasibility cuts obtained from a child. If some child problem $\tilde{N}_j(x_i)$ is not feasible then the following problem $\tilde{E}_j(x_i)$ is solved

Minimize:

$$w_j = e^T u_j^+ + e^T u_j^-$$

Subject to:

$$\begin{aligned} W_j x_j + I u_j^+ - I u_j^- &= h_j - A_j x_i \\ E_j^l x_j &\geq e_j^l && \forall l \in 1, \ldots, s_j \\ x_j &\geq 0 \\ u_j^+ &\geq 0 \\ u_j^- &\geq 0 \end{aligned}$$

This feasibility problem can only be formulated for linear problems, is always feasible, and bounded from below by 0. Its dual solution is used to construct a new feasibility constraint for $\tilde{N}_i(x_{p_i})$. Note that node j in its turn obtains optimality and/or feasibility cuts from its children for $\tilde{N}_j(x_i)$ and $\tilde{E}_j(x_i)$, unless j refers to a leaf node.

Terminating condition

If (x_i, θ_i) is an optimal solution of $\tilde{N}_i(x_{p_i})$ and

$$\theta_i \geq \tilde{N}_i(x_{p_i})$$

then (x_i, θ_i) is an optimal solution of $F_i(x_{p_i})$. If this holds for all non-leaf nodes then we have found an optimal solution of our original problem. For the leaf nodes, x_i only needs to be an optimal solution of $\tilde{N}_i(x_{p_i})$.

Implementation in AIMMS

The stochastic Benders algorithm outlined above is implemented in AIMMS as a system module that you can include into your model, together with a number of supporting functions in the GMP library to perform a number of algorithmic steps that cannot be performed in the AIMMS language itself, for instance, to actually generate the stochastic sub-problems, and to generate feasibility and optimality cuts.

Adding the module

You can add the system module implementing the stochastic Benders algorithm to your model through the **Settings-Install System Module...** menu. By selecting the **Stochastic Decomposition Module** in the **Install System Module** dialog box, AIMMS will add this system module to your model.

Using the stochastic Benders module

The main procedure for using the stochastic Benders algorithm is `DoStochasticDecomposition`. Its inputs are:

- a stochastic GMP,
- the set of stages to consider, and
- the set of scenarios to consider.

The procedure implements the algorithm outlined above. The supporting GMP functions for the stochastic Benders algorithm are described in Section 16.8.

Modifying the algorithm

Because the stochastic Benders algorithm is written in the AIMMS language, you have complete freedom to modify the algorithm in order to tune it for your stochastic programs. Also, the basic algorithm solves all sub-problems sequentially. If your AIMMS license permits parallel solver sessions, you can also speed up the algorithm by solving multiple sub-problems in parallel using the GMP function `GMP::SolverSession::AsynchronousExecute`.

Chapter 20

Robust Optimization

Introduction

Robust optimization is a rather new modeling methodology for decision making under uncertainty. Robust optimization is designed to meet some major challenges associated with uncertainty-affected optimization problems:

- to operate under lack of full information on the nature of uncertainty,
- to model the problem in a form that can be solved efficiently, and
- to provide guarantees about the performance of the solution.

Robustness of decisions is defined in terms of the best performance in the worst case possible state-of-the-world (min-max optimization). A more in-depth discussion of robust optimization can be found, for instance, in [BT09].

Robust optimization in AIMMS

In this chapter, you will find a description of the facilities built into AIMMS for creating and solving robust optimization models. From any existing deterministic linear program (LP) or mixed-integer program (MIP), AIMMS is able to automatically create a robust optimization model as well, *without the need for you to reformulate any of the constraint definitions*. The only steps necessary to create a robust optimization model are

- to indicate which parameters in your deterministic model are to become uncertain in a declarative manner,
- to indicate which variables in your deterministic model are to become adjustable to the uncertain parameters (if any), and
- to specify possible realizations of the uncertain parameters.

Single formulation

Being able to generate both a deterministic and robust optimization model from an identical symbolic formulation allows for any changes you make in the deterministic formulation to automatically propagate to the robust optimization model. This significantly reduces the effort involved with maintaining a robust optimization model associated with a given deterministic model.

Robust Optimization Add-On required

To be able to run an robust optimization model, you need to make sure you have the *Robust Optimization Add-On* licensed. Without the RO Add-On, you can still define your robust optimization models, but will be unable to solve them (an execution error will occur).

Acknowledgements

The Robust Optimization Add-On in AIMMS has been developed in close cooperation with Professor Aharon Ben-Tal and Boris Bachelis of the Technion, Israel Institute of Technology. We would like to express our gratitude for our partnership in developing the Robust Optimization Add-On in AIMMS and for their continuous support to get the details right, which allowed us to make Robust Optimization a natural and intuitive extension to our existing functionality.

This chapter

Section 20.1 discusses a number of basic concepts in robust optimization. These provide a common understanding necessary for the introduction of the robust optimization features of AIMMS discussed in the sections to follow. Section 20.2 describes the facilities available in AIMMS for specifying uncertain parameters, while Section 20.3 discusses chance constraints as another means to introduce uncertainty into your robust optimization model. Section 20.4 discusses the facilities available to declare variables to be adjustable to uncertain parameters. Section 20.5, finally, describes the steps how to actually solve a robust optimization model.

20.1 Basic concepts

Basic concepts

In this section you will find a number of basic concepts that are commonly used in robust optimization. They will help you to unambiguously understand the robust optimization facilities in AIMMS.

Robust counterpart

In robust optimization the model with uncertain data is translated into the so-called *robust counterpart*. Consider the following linear programming problem:

$$\max\{c^T x : A^T x \le b\} \tag{P}$$

in which $c \in \mathbb{R}^m$, $b \in \mathbb{R}^n$ and $A \in \mathbb{R}^{m\times n}$. Suppose that the actual technology matrix A is in fact uncertain and it is only known to belong to a bounded uncertainty set $U_A \subset \mathbb{R}^{m\times n}$. Similarly assume that right hand side b belongs to an uncertainty set $U_b \subset \mathbb{R}^n$, and the objective coefficients c to an uncertainty set $U_c \subset \mathbb{R}^m$. The robust counterpart (RC) for the nominal problem (P) is then defined as follows:

$$\max\{c^T x : A^T x \le b,\ \forall A \in U_A, c \in U_c, b \in U_b\}. \tag{RC}$$

Uncertainty set

The sets U_A, U_c and U_b specify all possible realizations of the uncertain data and are collectively called the *uncertainty set*. The main questions associated with the uncertainty set are:

- When and how can the robust counterpart of an uncertain problem be reformulated as a computationally tractable optimization problem?

- How to specify a reasonable uncertainty set, i.e., meaningful for a particular application and yielding a tractable robust counterpart?

It can be shown that when the uncertainty set is a multi-dimensional interval or described by linear constraints, then the robust counterpart can be reformulated as a linear problem. Furthermore, when the uncertainty set is an ellipsoid, then the robust counterpart is still tractable, i.e., it can be reformulated as a second-order cone program (SOCP), for which efficient (polynomial time) solution methods exist. The reformulation of the robust counterpart is an automated process performed by AIMMS during the generation of your mathematical program.

Integer programming

If the uncertainty set is a multi-dimensional interval or described by linear constraints, then the robust counterpart of a mixed-integer robust optimization problem can also be reformulated as a mixed-integer optimization problem. If the uncertainty set is described by ellipsoidal constraints then the robust counterpart becomes a mixed-integer second-order cone program. This class of problems is more difficult to solve than mixed-integer optimization problems.

Scenarios

A special situation to consider is when the uncertainty set consists of a finite number of points representing a collection of scenarios. This discrete case resembles the situation in multi-stage stochastic programming with discrete data realizations. More precisely, in this case hard constraints are imposed for every scenario s, while the objective is to optimize a worst-case performance measure with respect to the set of scenarios. This performance measure can be, for example, the objective value of the original (deterministic) model associated with an uncertain scenario. Another possibility is to define this performance measure as a deviation of the objective for a decision with respect to the absolutely optimal objective for each scenario. In the latter case, the optimal robust solution will be the one with the minimum maximum deviation across scenarios.

Chance constraints

Another manner to account for uncertainty into your model is by specifying so-called *chance constraints*. In order to introduce chance constraints, you have to declare some of the parameters in your model to take random values from a distribution with given characteristics. Subsequently, you can specify that some of the constraints in your model be satisfied with a given probability with respect to the specified data distributions. For example, if a chance constraint has a probability of 95%, this means that the constraint should be satisified for (at least) 95% of the realizations drawn from specified distributions of the random parameters contained in it. Compared to using uncertain parameters, specifying random parameters with the same range and formulating the existing constraints as chance constraints may lead to solutions that put less emphasis on worst-case scenarios that only occur occasionally.

Multistage optimization

All decision variables in problem (P) represent "here and now" decisions; they get specific numerical values as a result of solving the problem before the actual data "reveals itself" and as such are independent of the actual values of the data. There are situations where this is too restrictive, since "in reality" some of the decision variables can adjust themselves, to some extent, to the true values of the uncertain data.

Adjustable variables

For that reason, it is possible to specify both *non-adjustable* and *adjustable* variables in AIMMS, similar to first-stage and second-stage (or multi-stage) decisions in stochastic programming, where the solution of a variable in stage n depends on the specific solution of variables in stage $n-1$ in a scenario-dependent manner. Please note that, while non-adjustable variables can be integer, adjustable variables *must* be continuous.

Basic procedure for solving robust optimization models

The implementation of robust optimization in AIMMS closely follows the concepts described in this section. The basic procedure to create and solve a robust optimization model in AIMMS is as follows:

- indicate in your model which parameters are to become uncertain or random,
- for every constraint in your model that you want to become a chance constraint, specify the probability with which it must hold,
- for every adjustable variable in your model specify on which uncertain parameters it depends, and
- specify possible realizations of the uncertain parameters in terms of predefined regions or using specialized uncertainty constraints.

Each of these steps is explained in more detail in the sections to follow. Note that changing parameters, variables and constraints in your model into uncertain or random parameters, adjustable variables and chance constraints does in no way influence the possibility to solve the underlying deterministic model in its original form. Thus, the robust optimization facilities in AIMMS always form a true extension of the functionality of the existing AIMMS application. It is even possible to extend an existing deterministic model to both a stochastic model and a robust optimization model, all of which can be solved independently.

20.2 Uncertain parameters and uncertainty constraints

Uncertain parameters

Uncertain parameters are modeled in AIMMS as numeric `Parameters` for which the `Uncertain` property has been set (see also Section 4.1). When a parameter has been declared `Uncertain` AIMMS will create two new attributes `Region` and `Uncertainty`.

The Region attribute

The Region attribute of an uncertain parameter offers an easy way to define the uncertainty set without the need to introduce additional uncertain parameters. AIMMS supports a number of predefined regions which you can enter here:

- Box(l,u),
- Ellipsoid(c,r), and
- ConvexHull$(s,v(s))$.

Box example

If we want to specify that parameter A is uncertain and constrained as follows:

$$l(i,j) \le A(i,j) \le u(i,j)$$

then it suffices to specify the uncertainty set of A using its Region attribute as follows

```
Parameter A {
    IndexDomain  : (i,j);
    Property     : Uncertain;
    Region       : Box( l(i,j), u(i,j) );
}
```

where l(i,j) and u(i,j) are ordinary parameters in your model.

Ellipsoid example

It is also possible to specify the region using an Ellipsoid region

```
Parameter A {
    IndexDomain  : (i,j);
    Property     : Uncertain;
    Region       : Ellipsoid( A.level(i,j), r(i,j) );
}
```

which leads to an uncertainty set for A defined as an ellipsoid around the nominal value of A as follows:

$$\sum_{i,j} \left(\frac{A(i,j) - A.level(i,j)}{r(i,j)} \right)^2 \le 1.$$

ConvexHull example

The region can also be defined as a ConvexHull region

```
Parameter A {
    IndexDomain  : (i,j);
    Property     : Uncertain;
    Region       : ConvexHull( s, A_s(s,i,j) );
}
```

which says that the uncertain parameter A belongs to an uncertainty set that is described by the convex hull of the values of a collection of values A_s for a given set of scenarios, i.e.,

$$A(i,j) = \sum_s \lambda_s A_s(s,i,j)$$
$$1 = \sum_s \lambda_s, \qquad \lambda_s \ge 0.$$

Dependencies

If there are two parameters A and B that both depend on scenario-dependent data, then those scenarios can either be dependent or independent. To differentiate between these two possibilities, AIMMS uses the name of the binding index used in the ConvexHull operator. If the names of the binding indices are identical, then AIMMS assumes that the scenarios are dependent. If the index names are different, *even if they refer to the same scenario set*, AIMMS assumes the scenarios to be independent.

Dependent scenarios example

Consider the following two declarations of uncertain parameters

```
Parameter A {
    IndexDomain  :  (i,j);
    Property     :  Uncertain;
    Region       :  ConvexHull( s, A_s(s,i,j) );
}
Parameter B {
    IndexDomain  :  (i,j);
    Property     :  Uncertain;
    Region       :  ConvexHull( s, B_s(s,i,j) );
}
```

Based on these declarations AIMMS will generate a single convex hull as follows

$$\begin{bmatrix} A(i,j) \\ B(i,j) \end{bmatrix} = \sum_s \lambda_s \begin{bmatrix} A_s(s,i,j) \\ B_s(s,i,j) \end{bmatrix}$$
$$\sum_s \lambda_s = 1, \qquad \lambda_s \geq 0.$$

If A and B consist of a single value each, and there are two scenarios for s, then the combined convex hull for A and B is depicted in Figure 20.1.

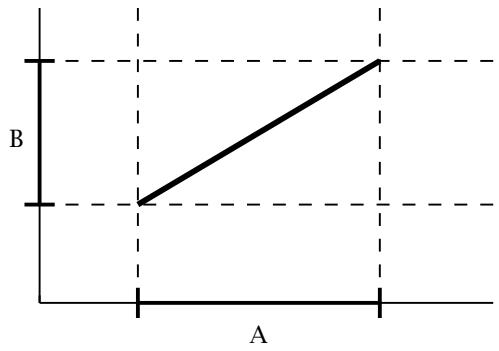


Figure 20.1: Combined convex hull for dependent scenarios

If, on the other hand, both declarations are given as

Independent scenarios example

```
Parameter A {
    IndexDomain  :  (i,j);
    Property     :  Uncertain;
    Region       :  ConvexHull( s, A_s(s,i,j) );
}
Parameter B {
    IndexDomain  :  (i,j);
    Property     :  Uncertain;
    Region       :  ConvexHull( t, B_t(t,i,j) );
}
```

then AIMMS will generate two separate convex hulls as follows

$$\begin{bmatrix} A(i,j) \\ B(i,j) \end{bmatrix} = \begin{bmatrix} \sum_s \lambda_s A_s(s,i,j) \\ \sum_t \mu_t B_t(t,i,j) \end{bmatrix}$$
$$\sum_s \lambda_s = \sum_t \mu_t = 1, \qquad \lambda_s \geq 0, \mu_t \geq 0.$$

If A and B consist of a single value each, and there are two scenarios for s and t each, then the combined convex hull for A and B is depicted in Figure 20.2.

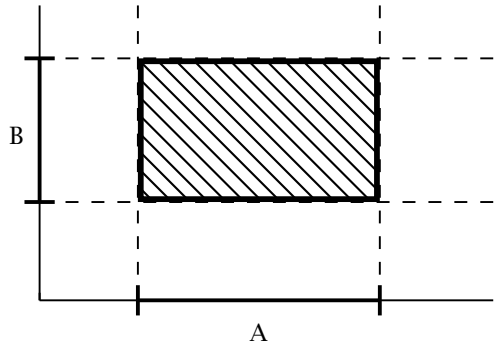


Figure 20.2: Combined convex hull for independent scenarios

ConvexHullEx

The ConvexHull operator AIMMS can be used to express that an uncertain parameter is defined as the convex combination of a certain parameter on some set of scenarios. The ConvexHullEx operator is an extension for which the user explicitly has to define the "lambda" parameter as an uncertain parameter. For example:

```
Parameter A {
    IndexDomain  : (i,j);
    Property     : Uncertain;
    Region       : ConvexHullEx( s, A_s(s,i,j), L(s,i) );
}
```

which says that the uncertain parameter A belongs to an uncertainty set that is described by the convex hull of the values of a collection of values A_s for a

given set of scenarios using the uncertain parameter L, i.e.,

$$A(i,j) = \sum_s L_s(i) A_s(s,i,j)$$
$$1 = \sum_s L_s(i), \qquad L_s(i) \geq 0.$$

More flexibility

The ConvexHullEx operator offers more flexibility as demonstrated by the above example in which the *lambda* parameter L depends on the indices s and i while the implicitly generated *lambda* parameter in case of the ConvexHull operator only depends on the index s. Moreover, the *lambda* parameter can be used in the Dependency attribute of an adjustable variable (see Section 20.4). The same *lambda* parameter can be used in ConvexHullEx in regions of different uncertain parameters to define a dependency between the uncertain parameters. As the *lambda* parameter is not an ordinary uncertainty parameter, it cannot be used in uncertainty constraints.

The Uncertainty attribute

Through the Uncertainty attribute of an uncertain parameter you can define a relation in term of other ordinary and uncertain parameters in your model which must hold for the uncertain value of that parameter.

Example

Consider the following declaration

```
Parameter Demand {
    IndexDomain  : (c,t);
    Property     : Uncertain;
    Uncertainty  : Demand.level(c,t) + Sum[k, D(c,t,k) * xi(k)];
}
```

where D(c,t,k) is an ordinary parameter and xi an uncertain parameter. The reference to Demand.level in the Uncertainty attribute refers to the deterministic (or nominal) value of Demand. The uncertain value of Demand is defined as its nominal value plus a linear combination of some other uncertain parameter xi(k).

Non-exclusive attributes

Note that the Region and Uncertainty attributes are non-exclusive, i.e., you can use them in conjuction to each other. In such a case, AIMMS will make sure that the solution is robust with respect to both relations.

Uncertainty constraints

The Region and the Uncertainty attribute of a uncertain parameter can be used to specify possible realizations of the uncertain parameters. In some cases, however, more flexibility is needed in specifying special relations for one or more uncertain parameters. For this purpose AIMMS allows you to specify UncertaintyConstraints. An UncertaintyConstraint is a constraint that specifies the relation between uncertain parameters. It is similar to an ordinary constraint in which the uncertain parameters play the role for variables; the

definition of an UncertaintyConstraint may only refer to normal and uncertain parameters, and not to variables.

Example

The following example specifies a condition on an uncertain parameter that cannot be expressed through its Region or Uncertainty attributes.

```
Parameter A {
    IndexDomain  : (i,j);
    Property     : Uncertain;
}
UncertaintyConstraint ConditionOnA {
    IndexDomain  : i;
    Definition   : Sum( j, A(i,j) ) <= 1;
}
```

The Constraints attribute

Through the Constraint attribute of an UncertaintyConstraint you can specify to which (normal) constraints the UncertaintyConstraint should apply. In this way it is possible to use different uncertainty sets for different constraints. If the Constraints attribute is empty then the UncertaintyConstraint will be active for all constraints.

Example

Consider the following declarations

```
UncertaintyConstraint ConditionOnA {
    IndexDomain  : i;
    Constraints  : CapacityRestriction(j) : UncertaintyDependency(i,j);
    Definition   : Sum( j, A(i,j) ) <= 1;
}
Constraint CapacityRestriction {
    IndexDomain  : j;
    Definition   : Sum( i, A(i,j) * Transport(i,j) ) <= Capacity(j);
}
Parameter UncertaintyDependency {
    IndexDomain  : (i,j);
    Definition   : 1 $ (i = j);
}
```

These declarations yield that the uncertainty constraint ConditionOnA(i) is only active for constraint CapacityRestriction(j) for all elements j equal to i.

Generalized ellipsoid

Besides linear uncertainty constraints, AIMMS also allows you to formulate the following uncertainty set for a uncertain parameter ξ, that generalizes the ellipsoidal uncertainty sets that can be defined by using the Ellipsoid region:

$$\xi^T Q_0 \xi + \sum_{m=1}^{M} \sqrt{\xi^T Q_m \xi} \leq b,$$

where Q_0 and Q_m should be positive semidefinite matrices. If your model contains an ellipsoidal uncertainty constraint then the robust counterpart will become a second-order cone program, except if the ellipsoidal uncertainty constraints are of the form

$$\sum_i \sqrt{\xi_i^2} \leq b,$$

in which case the robust counterpart will be a linear program.

20.3 Chance constraints

Chance constraints

In the previous sections we assumed that each constraint with uncertain data was satisfied with probability 1. In many situations, however, such a requirement may lead to solutions that over-emphasize the worst-case. In such cases, it is more natural to require that a candidate solution has to satisfy a constraint with uncertain data for "nearly all" realizations of the uncertain data. More specifically, in this approach one requires that the robust solution has to satisfy the constraint with probability at least $1 - \epsilon$, where $\epsilon \in [0, 1]$ is a pre-specified small tolerance. Instead of the deterministic constraint

$$a^T x \leq b$$

we now require that the *chance constraint*

$$\text{Prob}\left(x : a(\xi)^T x \leq b\right) \geq 1 - \epsilon$$

be satisfied, where the probability is associated with the specific distribution of the uncertain parameter(s) ξ.

Approximation

In general, (linear) problems with chance constraints are very hard to solve even if the probability distribution of the uncertain data is completely known. It is, however, possible to construct safe tractable approximations of chance constraints using robust optimization (see, for instance, Chapter 2 of [BT09]). The way a chance constraint is approximated depends merely on the general characteristics of the data distribution, rather than on precise specification of the distribution. If more information is available about the distribution, this will generally result in a tighter approximation. A tighter approximation, however, could result in a more difficult solution process (for instance, requiring second-order cone programming instead of just linear programming).

Chance constraints in AIMMS

The procedure to introduce chance constraints into your robust optimization model is as follows:

- indicate which parameters in your model should become random, and specify the properties of their distributions, and
- specify which constraints should be considered chance constraints, and specify their probability and method of approximation.

Random parameters

A probability distribution is modeled in AIMMS as a numeric Parameter for which the Random property has been set (see also Section 4.1). If the property Random is set, AIMMS will create the mandatory Distribution attribute for this parameter which must be used to specify the characteristics of the distribution to be used for that parameter. All random parameters for which a distribution has been specified are considered to be independent.

Example

Consider the following declaration

```
Parameter Demand {
    IndexDomain  : i;
    Property     : Random;
    Distribution : Bounded(Demand(i).level,0.1);
}
```

This declaration states that parameter Demand corresponds to a bounded probability distribution with a mean equal to the nominal value of Demand and a support of 0.1.

Supported distributions

AIMMS supports the distribution types listed in Table 20.1 All distributions

Distribution	Meaning
Bounded(m,s)	mean m with range $[m-s, m+s]$
Bounded(m,l,u)	range $[l,u]$ and mean m not in the center of the range
Bounded(m_l,m_u,l,u)	range $[l,u]$ and mean in interval $[m_l,m_u]$
Bounded(m_l,m_u,l,u,v)	range $[l,u]$ and mean in interval $[m_l,m_u]$, and variance bounded by v
Unimodal(c,s)	unimodal around c with range $[c-s,c+s]$
Symmetric(c,s)	symmetric around c with range $[c-s,c+s]$
Symmetric(c,s,v)	symmetric around c with range $[c-s,c+s]$, and variance bounded by v
Support(l,u)	range $[l,u]$ (and no information about the mean)
Gaussian(m_l,m_u,v)	Gaussian with mean in interval $[m_l,m_u]$ and variance bounded by v

Table 20.1: Supported distribution types for robust optimization

in this table are bounded except the Gaussian distribution. The distributions Bounded(m_l,m_u,l,u), Bounded(m_l,m_u,l,u,v) and Symmetric(c,s,v) are currently not implemented.

Symmetric unimodal distribution

A distribution is called unimodal if its density function is monotonically increasing up to a certain point c and monotonically decreasing afterwards. For symmetric distribution AIMMS offers the possibility to mark it as unimodal by using the unimodal keyword:

```
Parameter Demand {
    IndexDomain  : i;
    Property     : Random;
    Distribution : Symmetric(Demand(i).level,0.1), unimodal;
}
```

The unimodal keyword can only be used in combination with a symmetric distribution.

Linear relation

In addition to specifying a random parameter using an independent distribution, AIMMS also allows you to define a random parameter as a linear combination of other random parameters (but not as combination of uncertain parameters). For example,

```
Parameter Demand {
    Property     : Random;
    Distribution : Sum( i, xi(i) );
}
```

where xi is an random parameter. To avoid cyclic definitions, AIMMS requires that the distributions of random parameters cannot be specified as an expression of other random parameters which are themselves defined as an expression of random parameters.

Chance constraints

A constraint in your mathematical program becomes a chance constraint in the context of robust optimization by setting its Chance property. The definition of a chance constraint may only contain random parameters, normal parameters and variables. Uncertain parameters are not allowed inside a chance constraint. When setting the Chance property for a constraint, you must specify two new attributes for the constraint, the Probability attribute and the Approximation attribute. It is allowed to use chance constraints in a mixed-integer program.

The Probability attribute

The Probability attribute specifies the probability with which the chance constraint should be satisfied when solving a robust optimization model. The value of the Probability attribute should be a numerical expression in the range $[0,1]$. If the probability is 0, then AIMMS will not generate the chance constraint. If the probability is 1, then AIMMS will generate an uncertainty constraint.

The Approximation attribute

The Approximation attribute is used to define the approximation that should be used to approximate the chance constraint. Its value should be an element expression into the predefined set AllChanceApproximationTypes.

Supported approximation types

The approximations supported by AIMMS are:

- *Ball*,
- *Box*,
- *Ball-box*,
- *Budgeted*, and
- *Automatic*.

A detailed mathematical definition of these approximation types can be found in Chapter 2 of [BT09]. Whether or not a particular approximation type is possible, depends on the characteristics of the distributions used in the chance constraint, as explained below. By specifying approximation type *Automatic* the most accurate approximation possible will be used. In some cases it might be beneficial to use a less tight approximation because it leads to a robust counterpart that is easier to solve.

Example

Consider the declaration

```
Constraint ChanceConstraint {
    IndexDomain   : i;
    Property      : Chance;
    Definition    : Demand(i) * X(i) <= 10;
    Probability   : prob(i);
    Approximation : 'Ball';
}
```

This declaration states that ChanceConstraint is a chance constraint with probability prob(i), and that approximation type *Ball* is used to approximate the chance constraint.

Possible approximations per distribution

Table 20.2 shows for each (supported) distribution which approximation types are possible. It also shows whether the approximation will result in a linear or a second-order cone robust counterpart. For the Bounded(m,s) distribu-

Distribution	Automatic	Ball	Box	Ball-box	Budgeted
Bounded(m,s)	linear	conic	linear	conic	linear
Bounded(m,l,u)	conic		linear		
Unimodal(c,s)	conic		linear		
Symmetric(c,s) (unimodal)	conic	conic	linear	conic	linear
Support(l,u)	linear		linear		
Gaussian(m_l,m_u,v)	conic				

Table 20.2: Allowed approximations and their resulting problem type

tion the automatic approximation equals the *Budgeted* approximation, and the automatic approximation of the Support(l,u) distribution equals the *Box* approximation. The non-unimodal Symmetric(c,s) distribution is treated as a Bounded(m,s) distribution.

Combining distributions

A chance constraint cannot contain both bounded random parameters and Gaussian random parameters. Different types of bounded random parameters can be combined, in which case only a part of the available information will be used. The possible combinations of bounded random parameters are given in Table 20.3.

	Distribution	1	2	3	4	5
1	Bounded(m,s)	1	2	–	1	5
2	Bounded(m,l,u)	2	2	–	2	5
3	Unimodal(c,s)	–	–	3	3	5
4	Symmetric(c,s) (unimodal)	1	2	3	4	5
5	Support(l,u)	5	5	5	5	5

Table 20.3: Resulting distribution type when combining distributions

Explanation

If a random parameter with a Bounded(m,l,u) distribution and a random parameter with a Support(l,u) distribution are used in a single chance constraint, then Table 20.3 states that the Bounded(m,l,u)) distribution of the first random parameter will be treated as a Support(l,u) distribution. Unimodal distributions can only be mixed with unimodal Symmetric(c,s) and Support(l,u) distributions.

20.4 Adjustable variables

Adjustable variables

An *adjustable variable* reflects a decision made after uncertain data has been revealed. In robust optimization this is interpreted as the adjustable variable taking some (explicit or implicit) functional form in terms of the uncertain data on which it depends. In AIMMS, you indicate that a Variable should be treated as adjustable by setting its Adjustable property.

The Dependency attribute

For any adjustable variable, AIMMS will create a Dependency attribute which you can use to specify on which uncertain parameters the variable depends. The attribute value must be a comma-separated list of mappings from an uncertain parameter to a binary parameter, indicating for which combination of indices a dependency exists on that uncertain parameter.

Linear decision rule only

AIMMS currently only supports the *linear decision rule*, which means any adjustable variable will be expressed as an affine relation in terms of the uncertain parameters which it depends on. More explicitly, if an adjustable variable $x(t)$ depends on uncertain parameters d_r, then, under the linear decision rule, AIMMS assumes that $x(t)$ takes the form

$$x(t) = X_0(t) + \sum_r X_r(t) d_r$$

where $X_0(t)$ and $X_r(t)$ are newly introduced intermediate variables, the value of which is determined by solving the robust counterpart. As such, the value of an adjustable variable is not fully determined by the solver. It can be computed afterwards for a given realization of the uncertain parameters. AIMMS will automatically generate the affine relation based on the dependencies you indicated in the Dependency attribute, without the need for you to introduce the appropriate intermediate variables.

Requirements for adjustable variables

In order for AIMMS to be able to generate the robust counterpart of a robust optimization model, the model must satisfy the *fixed recourse* condition, i.e., the coefficients of any adjustable variables in your model must not depend on uncertain parameters. In addition, for AIMMS to be able to generate the robust counterpart, adjustable variables may *not* occur in chance constraints. Also, adjustable variables cannot be integer.

The .Adjustable suffix for variables

The collection of intermediate variables introduced during this process, automatically becomes available through the .Adjustable attribute of the adjustable variable at hand, followed by the name of the uncertain parameter involved. That is, if an adjustable variable x(i) depends on an uncertain parameter a(j), then the corresponding intermediate variable is available as the expression x.Adjustable.a(i,j). In addition, a variable x.Adjustable.Constant(i) will be created to account for the constant part of the affine relation. If necessary, you can bound these variables through the .Lower and .Upper suffices, or you can formulate additional constraints on these variables.

Example

Consider the following declarations

```
Variable Stock {
    IndexDomain  :  t;
    Property     :  Adjustable;
    Dependency   :  Demand(t2) : StockDemandDependency(t,t2);
}
Parameter Demand {
    IndexDomain  :  t;
    Property     :  Uncertain;
}
Parameter StockDemandDependency {
    IndexDomain  :  (t,t2);
    Definition   :  1 $ (t2 < t);
}
```

These declarations yield that the adjustable variable Stock(t) depends on the uncertain parameter Demand(t2) for all elements t2 smaller than t. Given these declarations, AIMMS will generate the following definition for Stock(t)

```
Stock(t) = Stock.Adjustable.Constant(t) +
       sum(t2 | StockDemandDependency(t,t2), Stock.Adjustable.Demand(t,t2)*Demand(t2))
```

If the data for Demand(t) becomes available, you can use the computed values of Stock.Adjustable.Demand(t,t2) and Stock.Adjustable.Constant to compute the value of Stock(t).

Warning: using same indices

You should be aware that using the same indices in the Dependency attribute and the index domain of the adjustable variable will restrict the dependencies that are generated. For example, assume we have the following declarations

```
Variable Stock {
    IndexDomain  :  t;
    Property     :  Adjustable;
    Dependency   :  Demand(t);
}
Parameter Demand {
    IndexDomain  :  t;
    Property     :  Uncertain;
}
```

Given these declarations, AIMMS will generate the following definition for Stock(t)

```
Stock(t) = Stock.Adjustable.Constant(t) + Stock.Adjustable.Demand(t)*Demand(t)
```

If you want Stock(t) to depend on all possible Demand then you should use a different index in the Dependency attribute, e.g.,

```
Variable Stock {
    IndexDomain  :  t;
    Property     :  Adjustable;
    Dependency   :  Demand(t2);
}
```

Evaluating adjustable variables

To compute the values of an adjustable variable for a given realization of the uncertain parameters of the robust optimization model, you do not have to explicitly add the appropriate definitions to your model. AIMMS offers the function GMP::Robust::EvaluateAdjustableVariables, discussed in Section 16.9, to automatically compute these values for you.

20.5 Solving robust optimization models

Solving robust optimization models

After you have specified all uncertain parameters, random parameters, chance constraints and adjustable variables that specify your robust optimization model, your original mathematical program can now be solved as a robust optimization model. It is also still possible to solve it as a *deterministic* model by just calling the SOLVE statement (see also Section 15.3).

Generate robust counterpart

To solve a robust optimization model for a MathematicalProgram *MP*, the first step is to generate its robust counterpart. This can be accomplished by calling the GMP function

- GenerateRobustCounterpart(*MP*,*UncertainParameters*, *UncertaintyConstraints*[,*Name*])

The function returns an element into the set AllGeneratedMathematicalPrograms, i.e., the generated mathematical program representing the robust counterpart of the given robust optimization model.

Specifying uncertain data

Through the *UncertainParameters* and *UncertaintyConstraints* arguments you can specify the collection of uncertain and random parameters, as well as the uncertainty constraints that you want to take into account when generating the robust counterpart. Together, these completely determine the uncertain data which AIMMS will use to translate the uncertain matrix coefficients, chance constraints and adjustable variables into the generated mathematical program representing the robust counterpart.

Name argument

With the optional *Name* argument you can explicitly specify a name for the generated mathematical program. If you do not choose a name, AIMMS will use the name of the underlying MathematicalProgram as the name of the generated mathematical program as well. Please note, that AIMMS will also use this name as the default name for solving the deterministic model. Therefore, if you do not want the generated mathematical program of the deterministic model to be deleted, then you have to choose a non-default name.

Solving the robust counterpart

You can solve the generated mathematical program *gmp* representing the robust counterpart by calling the regular GMP procedure

- GMP::Instance::Solve(*gmp*)

The GMP::Instance::Solve method is discussed in full detail in Section 16.2. Alternatively, you can use any of the other available functions available to solve generated mathematical programs discussed in Chapter 16. Note that AIMMS will not allow you to use the GMP modification functions on any gmp generated by GenerateRobustCounterpart.

The resulting solution

The solution resulting from solving the robust counterpart will satisfy all non-chance constraints in your model for all realizations of the uncertain parameters that you passed to the GenerateRobustCounterPart function, and will satisfy all chance constraints with the given probabilities and approximations, given the random parameters taken into account.

Chapter 21

Automatic Benders' Decomposition

Important note

The solver CPLEX has its own implementation of the Benders' decomposition algorithm. An important difference is that the algorithm in CPLEX supports multiple subproblems. CPLEX allows you to specify the decomposition by assigning the variables to the master problem or a subproblem by using the procedure `GMP::Column::SetDecomposition`. For more information see the CPLEX option `Benders_strategy`.

Introduction

Benders' decomposition, introduced by Jacques F. Benders in 1962 ([Be62]), is an algorithm that decomposes a problem into two simpler parts. The first part is called the master problem and solves a relaxed version of the problem to obtain values for a subset of the variables. The second part is often called the subproblem (or slave problem or auxiliary problem) and finds values for the remaining variables fixing the variables of the master problem. If the problem contains integer variables then typically they become part of the master problem while the continuous variables become part of the subproblem.

Cuts

The solution of the subproblem is used to cut off the solution of the master problem by adding one or more constraints ("cuts") to the master problem. This process of iteratively solving master problems and subproblems is repeated until no more cuts can be generated. The combination of the variables found in the last master problem and subproblem iteration forms the solution to the original problem.

Reduced solution times possible

For particular optimization problems, Benders' decomposition may lead to a good, or even the optimal, solution in relatively few iterations. In such cases, employing Benders' decomposition results in drastically reduced solution times compared to solving the original problem. For other problems, however, the progress per iteration is so small that there is no positive, or even an adversary, effect by applying Benders' decomposition. Upfront, it is hard predict whether or not there will be positive effects for your particular model.

Hard to implement manually

Implementing Benders' decomposition from scratch for a particular problem is a non-trivial and error-prone task. Because duality theory plays an important role, the process often involves explicitly working out the dual formulation of the subproblem—and keeping it up-to-date when you make changes to the original problem. Given the uncertainty whether Benders' decomposition will lead to an improvement in solution times at all, a manual implementation may not be a prospect to look forward to.

Automatic Benders' decomposition in AIMMS

For AIMMS, on the other hand, generating the master and slave problems in an automated fashion is a fairly straightforward task, given a generated mathematical program and the collection of variables that should go into the master problem. With such an automated scheme, verifying whether your particular model will benefit from Benders' decomposition becomes completely trivial. With just a few lines of code, and simply re-solving your model you will get immediate insight into the benefits of Benders' decomposition for your model.

Classical versus modern

The Benders' decomposition module in AIMMS implements both the classical Benders' decomposition algorithm and a modern version. By the classical approach we mean the algorithm described above that solves an alternating sequence of master problems and subproblems, and that, in principle, will work for any problem type. The modern approach will only work for problems containing integer variables. In the modern approach, the algorithm will solve only a *single* master MIP problem, where subproblems are solved whenever the MIP solver finds a solution for the master problem, using callbacks provided by modern MIP solvers.

Several algorithms

Besides the classical and the modern algorithm, the Benders' decomposition module in AIMMS also implements a two phase algorithm that solves a relaxed problem in the first phase and the original problem in the second phase. In addition, the module offers you the flexibility to solve the subproblem as a primal or dual problem, to normalize the subproblem to get better feasibility cuts, and so on.

Limitations of current implementation

Benders' decomposition in AIMMS can be used for solving Mixed-Integer Programming (MIP) problems and Linear Programming (LP) problems. Currently it cannot be used to solve nonlinear problems. Also, the current implementation does not support multiple subproblems which could be efficient in case the subproblem has a block diagonal structure. This implies that the current implementation cannot be used to solve (two stage) stochastic programming problems with a subproblem for each scenario.

Open algorithm

Benders' decomposition in AIMMS is implemented as a system module with the name GMP Benders Decomposition. You can install this module using the **Install System Module** command in the AIMMS **Settings** menu. The Benders' decomposition algorithms are implemented in the AIMMS language. Some supporting functions that are computationally difficult, or hard to express in the AIMMS language, have been added to the GMP library in support of the Benders' decomposition algorithm. Besides this small number of fixed subtasks, the implementation is an open algorithm; you as an algorithmic developer may want to customize the individual steps in order to obtain better performance and/or a better solution for your particular problem.

This chapter

This chapter starts with a quick start for using Benders' decomposition in AIMMS for those already familiar with Benders' decomposition. Following a brief introduction to the problem statement, we discuss the Benders' decomposition algorithm as it can be found in several textbooks. Next we describe the implementation of the classic Benders' decomposition algorithm using procedures in the AIMMS language that are especially designed to support the open approach. This section is important for users that want to modify the algorithm. Next, we discuss in detail the parameters inside the Benders' module that can be used to control the Benders' decomposition algorithm. We continue by describing the implementation of a modern Benders' decomposition algorithm. The chapter ends by introducing a two phase algorithm that solves a problem by using information gathered while solving a relaxed version of the problem, and we also describe its implementation.

21.1 Quick start to using Benders' decomposition

System module

The system module with the name GMP Benders Decomposition implements the Benders' decomposition algorithm. You can add this module to your project using the **Install System Module** command in the AIMMS **Settings** menu. This module contains three procedures that can be called, each implementing a different algorithm.

Classic algorithm

The procedure DoBendersDecompositionClassic inside this module implements the classic version of the Benders' decomposition algorithm, in which the master problem and the subproblem are solved in an alternating sequence.

Modern algorithm

The procedure DoBendersDecompositionSingleMIP inside the module implements the modern approach for MIP problems which solves only a single MIP problem; the subproblem is solved whenever the MIP solver finds a solution for the master problem (using callbacks).

Two phase algorithm

The procedure DoBendersDecompositionTwoPhase inside the module implements a two phase algorithm for MIP problems. In the first phase it solves the relaxed problem (in which the integer variables become continuous) using the classic Benders decomposition algorithm. The Benders' cuts found in the first phase are then added to the master problem in the second phase after which the MIP problem is solved using either the classic or modern approach of the Benders decomposition algorithm.

The procedure DoBenders-Decomposition-Classic

The procedure DoBendersDecompositionClassic has two input arguments:

1. MyGMP, an element parameter with range AllGeneratedMathematicalPrograms, and
2. MyMasterVariables, a subset of the predefined set AllVariables defining the variables in the master problem.

For a MIP problem, the integer variables typically become the variables of the master problem, although it is possible to also include continuous variables in the set of master problem variables. The DoBendersDecompositionClassic procedure is called as follows:

```
generatedMP := GMP::Instance::Generate( SymbolicMP );

GMPBenders::DoBendersDecompositionClassic( generatedMP, AllIntegerVariables );
```

Here SymbolicMP is the symbolic mathematical program containing the MIP model, and generatedMP is an element parameter in the predefined set AllGeneratedMathematicalPrograms. GMPBenders is the prefix of the Benders' module. The implementation of this procedure will be discussed in Section 21.3.

The procedure DoBenders-Decomposition-SingleMIP

The procedure DoBendersDecompositionSingleMIP has the same input arguments as the procedure DoBendersDecompositionClassic. The DoBendersDecomposition-SingleMIP procedure is called as follows:

```
generatedMP := GMP::Instance::Generate( SymbolicMP );

GMPBenders::DoBendersDecompositionSingleMIP( generatedMP, AllIntegerVariables );
```

This procedure can only be used if the original problem contains some integer variables. The implementation of this procedure will be discussed in Section 21.6.

The procedure DoBenders-Decomposition-TwoPhase

The procedure DoBendersDecompositionTwoPhase has one additional argument compared to the procedure DoBendersDecompositionClassic. Namely, the third argument UseSingleMIP is used to indicate whether the second phase should use the classic algorithm (value 0) or the modern algorithm (value 1). The procedure is called as follows if the modern algorithm should be used:

```
generatedMP := GMP::Instance::Generate( SymbolicMP );

GMPBenders::DoBendersDecompositionTwoPhase( generatedMP, AllIntegerVariables, 1 );
```

This procedure should only be used if the original problem contains some integer variables. The implementation of this procedure will be discussed in Section 21.7.

Combining procedure

To make it easier for you to switch between the three algorithms, the module also implements the procedure `DoBendersDecomposition` that calls one of the three procedures above based on the Benders' mode. The first two arguments of this procedure are the same as before, namely `MyGMP` and `MyMasterVariables`. The third argument, `BendersMode`, is an element parameter that defines the Benders' mode and can take value `'Classic'`, `'Modern'`, `'TwoPhaseClassic'` or `'TwoPhaseModern'`. The procedure is called as follows if the two phase algorithm should be used with the modern algorithm for the second phase:

```
generatedMP := GMP::Instance::Generate( SymbolicMP );

GMPBenders::DoBendersDecomposition( generatedMP, AllIntegerVariables,
                                    'TwoPhaseModern' );
```

If the problem contains no integer variables then only mode `'Classic'` can be used.

Control parameters

The Benders' module defines several parameters that influence the Benders' decomposition algorithm. These parameters have a similar functionality as options of a solver, e.g., CPLEX. The most important parameters, with their default setting, are shown in Table 21.1. The parameters that are not self-

Parameter	Default	Range	Subsection
BendersOptimalityTolerance	1e-6	[0,1]	
IterationLimit	1e7	{1,maxint}	
TimeLimit	1e9	[0,inf)	
CreateStatusFile	0	{0,1}	
UseDual	0	{0,1}	21.5.1
FeasibilityOnly	1	{0,1}	21.5.2
NormalizationType	1	{0,1}	21.5.3
UseMinMaxForFeasibilityProblem	1	{0,1}	21.5.4
AddTighteningConstraints	1	{0,1}	21.5.5
UseStartingPointForMaster	0	{0,1}	21.5.6
UsePresolver	0	{0,1}	21.5.7

Table 21.1: Control parameters in the Benders' module

explanatory are explained in Section 21.5; the last column in the table refers to the subsection that discusses the corresponding parameter.

Optimality tolerance

The optimality tolerance, as controlled by the parameter `BendersOptimalityTolerance`, guarantees that a solution returned by the Benders' decomposition algorithm lies within a certain percentage of the optimal solution.

Solver options

The parameters `BendersOptimalityTolerance`, `IterationLimit` and `TimeLimit` are used by the classic algorithm (and the first phase of the two phase algorithm). For the modern algorithm, the corresponding general solver options, `MIP_relative_optimality_tolerance`, `iteration_limit` and `time_limit` respectively, are used.

21.2 Problem statement

MIP

We consider the following generic mixed-integer programming model to explain Benders' Decomposition. (The notation used is similar to [Fi10].)

Minimize:

$$c^T x + d^T y$$

Subject to:

$$\begin{aligned} Ax &\leq b \\ Tx + Qy &\leq r \\ x &\in \mathbb{Z}_+^n \\ y &\in \mathbb{R}_+^m \end{aligned}$$

The variable x is integer and the variable y is continuous. The matrix T may contain empty constraints but we assume that the matrix Q does not contain any empty constraint. So, the model can have constraints that only contain continuous variables.

Limitation

Benders' Decomposition cannot be used if the model contains only integer variables. If the number of continuous variable is small compared to the number of integer variables then Benders' Decomposition will very likely be inefficient.

21.3 Benders' decomposition - Textbook algorithm

Master problem

The basic Benders' decomposition algorithm as explained in several textbooks (e.g., [Ne88], [Ma99]) works as follows. After introducing an artificial variable $\eta = d^T y$, the master problem relaxation becomes:

Minimize:

$$c^T x + \eta$$

Subject to:

$$\begin{aligned} Ax &\leq b \\ \eta &\geq \overline{\eta} \\ x &\in \mathbb{Z}_+^n \end{aligned}$$

Here $\overline{\eta}$ is a lower bound on the variable η that AIMMS will automatically derive. For example, if the vector d is nonnegative then we know that 0 is a lower bound on $d^T y$ since we assumed that the variable y is nonnegative, and therefore we can take $\overline{\eta} = 0$. We assume that the master problem is bounded.

Subproblem

After solving the master problem we obtain an optimal solution, denoted by (x^*, η^*) with x^* integer. This solution is fixed in the subproblem which we denote by $PS(x^*)$:

Minimize:

$$d^T y$$

Subject to:

$$Qy \leq r - Tx^*$$
$$y \in \mathbb{R}^m_+$$

Note that this subproblem is a linear programming problem in which the continuous variable y is the only variable.

Dual subproblem

Textbooks that explain Benders' decomposition often use the dual of this subproblem because duality theory plays an important role, and the Benders' optimality and feasibility cuts can be expressed using the variables of the dual problem. The dual of the subproblem $PS(x^*)$ is given by:

Maximize:

$$r - \pi^T (Tx^*)$$

Subject to:

$$\pi^T Q \geq d^T$$
$$\pi \geq 0$$

We denote this problem by $DS(x^*)$.

Optimality cut

If this subproblem is feasible, let z^* denote the optimal objective value and $\overline{\pi}$ an optimal solution of $DS(x^*)$. If $z^* \leq \eta^*$ then the current solution (x^*, η^*) is a feasible and optimal solution of our original problem, and the Benders' decomposition algorithm only needs to solve $PS(x^*)$ to obtain optimal values for variable y. If $z^* > \eta^*$ then the Benders' optimality cut $\eta \geq \overline{\pi}^T (r - Tx)$ is added to the master problem and the algorithm continues by solving the master problem again.

Feasibility cut

If the dual subproblem is unbounded, implying that the primal subproblem is infeasible, then an unbounded extreme ray $\overline{\pi}$ is selected and the Benders' feasibility cut $\overline{\pi}^T (r - Tx) \leq 0$ is added to the master problem. Modern solvers like CPLEX and GUROBI can provide an unbounded extreme ray in case a LP problem is unbounded. After adding the feasibility cut the Benders' decomposition algorithm continues by solving the master problem.

21.4 Implementation of the classic algorithm

The procedure DoBenders-Decomposition

In this section we show the implementation of the classic Benders' decomposition algorithm. It follows the classic approach of solving the master problem and the subproblem in an alternating sequence. The procedure DoBenders-Decomposition, introduced in Section 21.1, implements the classic algorithm.

Focus on textbook algorithm

The Benders' cuts can be generated in several ways; in this section we focus on the approach used in the textbook algorithm of the previous section (Section 21.3). The textbook algorithm uses the dual formulation of the subproblem and can add both Benders' optimality and feasibility cuts.

Calling DoBendersDecom-positionClassic

We have to change some of the control parameters of Table 21.1 to let the DoBendersDecomposition procedure execute the textbook algorithm. The relevant changes are listed below.

```
generatedMP := GMP::Instance::Generate( SymbolicMP );

! Settings needed to run textbook algorithm:
GMPBenders::FeasibilityOnly := 0;
GMPBenders::AddTighteningConstraints := 0;
GMPBenders::UseDual := 1;
GMPBenders::NormalizationType := 0;

GMPBenders::DoBendersDecompositionClassic( generatedMP, AllIntegerVariables );
```

Implementation of DoBendersDecom-positionClassic

The DoBendersDecompositionClassic procedure starts by making copies of its input arguments. Next the master problem and the subproblem are created. For the subproblem we also create a solver session which gives us more flexibility passing and retrieving subproblem related information. The parameters for the number of optimality and feasibility cuts are reset. Finally, the procedure calls another procedure, namely BendersAlgorithm, and finishes by deleting the master problem and the subproblem. For the sake of brevity and clarity, we leave out parts of the code that handle details like creating a status file; this will also be the case for the other pieces of code shown in this chapter.

```
OriginalGMP := MyGMP ;
VariablesMasterProblem := MyMasterVariables ;

! Create (Relaxed) Master problem.
gmpM := GMP::Benders::CreateMasterProblem( OriginalGMP, VariablesMasterProblem,
                        'BendersMasterProblem',
                        feasibilityOnly : FeasibilityOnly,
                        addConstraints : AddTighteningConstraints ) ;

! Create Subproblem.
gmpS := GMP::Benders::CreateSubProblem( OriginalGMP, gmpM, 'BendersSubProblem',
                        useDual : UseDual,
                        normalizationType : NormalizationType );
```

```
solsesS := GMP::Instance::CreateSolverSession( gmpS ) ;

NumberOfOptimalityCuts  := 0;
NumberOfFeasibilityCuts := 0;

! Start the actual Benders' decomposition algorithm.
BendersAlgorithm;

GMP::Instance::Delete( gmpM );
GMP::Instance::Delete( gmpS );
```

The procedure BendersAlgorithm

The BendersAlgorithm procedure implements the actual Benders' decomposition algorithm. It initializes the algorithm by resetting the parameters for the number of iterations, etc. Next the master problem is solved. The separation step solves the subproblem and checks whether the current solution is optimal. If it is not optimal then the algorithm creates a constraint ("cut") that separates the current solution from the set of feasible solutions. This constraint is added to the master problem enforcing that the current solution of the master problem will not be found again if we solve the master problem once again. This alternating sequence of solving master problems and subproblems is repeated until a stopping criterion is met.

```
InitializeAlgorithm;

while ( not BendersAlgorithmFinished ) do

    NumberOfIterations += 1;

    SolveMasterProblem;

    if ( UseDual ) then
        if ( FeasibilityOnly ) then
            SeparationFeasibilityOnlyDual;
        else
            SeparationOptimalityAndFeasibilityDual;
        endif;
    else
        if ( FeasibilityOnly ) then
            SeparationFeasibilityOnly;
        else
            SeparationOptimalityAndFeasibility;
        endif;
    endif;

endwhile;
```

Separation

The code above shows four possible ways of performing the separation step. The textbook algorithm uses the procedure SeparationOptimalityAndFeasibilityDual which we will discuss below. The other three separation procedures are discussed in Appendix B.

The procedure SolveMaster-Problem

The implementation of the SolveMasterProblem procedure is straightforward. This procedure solves the Benders' master problem and retrieves its objective value after checking the program status. If the program status is infeasible or unbounded then the algorithm terminates.

```
GMP::Instance::Solve( gmpM );

ProgramStatus := GMP::Solution::GetProgramStatus( gmpM, 1 ) ;

if ( ProgramStatus = 'Infeasible' ) then
    return AlgorithmTerminate( 'Infeasible' );
elseif ( ProgramStatus = 'Unbounded' ) then
    return AlgorithmTerminate( 'ProgramNotSolved' );
endif;

ObjectiveMaster := GMP::Instance::GetObjective( gmpM );
```

The procedure SeparationOptimalityAndFeasibilityDual

The procedure SeparationOptimalityAndFeasibilityDual is called by the Benders' decomposition algorithm in case the dual of the Benders' subproblem is used and if both optimality and feasibility cuts can be generated by the algorithm (we will discuss in Section 21.5 the case in which only feasibility cuts are generated). This procedure updates the dual subproblem and solves it. If the dual subproblem is unbounded then a feasibility cut is added to the master problem (using an unbounded extreme ray; see the next paragraph). If the subproblem is bounded and optimal then the objective value of the subproblem is compared to the objective value of the master problem to check whether the algorithm has found an optimal solution for the original problem. If the solution is not optimal yet then an optimality cut is added to the master problem, using the level values of the variables in the solution of the dual subproblem.

```
return when ( BendersAlgorithmFinished );

GMP::Benders::UpdateSubProblem( gmpS, gmpM, 1, round : 1 );

GMP::SolverSession::Execute( solsesS ) ;
GMP::Solution::RetrieveFromSolverSession( solsesS, 1 ) ;

ProgramStatus := GMP::Solution::GetProgramStatus( gmpS, 1 ) ;

if ( ProgramStatus = 'Unbounded' ) then

    ! Add feasibility cut to the Master problem.
    NumberOfFeasibilityCuts += 1;
    GMP::Benders::AddFeasibilityCut( gmpM, gmpS, 1, NumberOfFeasibilityCuts );

else

    ! Check whether optimality condition is satisfied.
    ObjectiveSubProblem := GMP::SolverSession::GetObjective( solsesS );

    if ( SolutionImprovement( ObjectiveSubProblem, BestObjective ) ) then
        BestObjective := ObjectiveSubProblem;
    endif;
```

```
    if ( SolutionIsOptimal( ObjectiveSubProblem, ObjectiveMaster ) ) then
        return AlgorithmTerminate( 'Optimal' );
    endif;

    ! Add optimality cut to the Master problem.
    NumberOfOptimalityCuts += 1;
    GMP::Benders::AddOptimalityCut( gmpM, gmpS, 1, NumberOfOptimalityCuts );

endif;
```

Unbounded extreme ray

In textbooks, if the dual subproblem is unbounded then an unbounded extreme ray is chosen and used to generate a feasibility cut. Choosing such an unbounded extreme ray is not trivial but luckily modern solvers like CPLEX and GUROBI can compute an unbounded extreme ray upon request. It is stored in the `.Level` suffix of the variables. The downside is that preprocessing by CPLEX or GUROBI has to be switched off which can have a negative impact on the performance. So, if the textbook algorithm is selected in which the dual subproblem is used and both optimality and feasibility cuts can be generated by the algorithm, the solver options for switching on the calculation of unbounded extreme ray and for switching off the preprocessor are set during the initialization of the Benders' decomposition algorithm:

```
if ( UseDual and ( not FeasibilityOnly ) ) then
    rval := GMP::SolverSession::SetOptionValue( solsesS, 'unbounded ray', 1 );
    if ( rval = 0 ) then
        halt with "Solver must support unbounded extreme rays.";
        return;
    endif;

    rval := GMP::SolverSession::SetOptionValue( solsesS, 'presolve', 0 );
    if ( rval = 0 ) then
        halt with "Switching off the solver option 'presolve' failed.";
        return;
    endif;
endif;
```

If the solver does not support unbounded extreme rays then the textbook algorithm cannot be used.

The procedure `AlgorithmTerminate`

The procedure `AlgorithmTerminate` is called whenever the Benders' decomposition algorithm is finished. Appropriate values are assigned to the program and solver status of the original problem. If the algorithm has found an optimal solution then the solutions of the last master problem and last subproblem are combined into an optimal solution for the original problem. In the code below, the uncommon situation in which the algorithm terminates after hitting the iteration limit has been omitted.

```
BendersAlgorithmFinished := 1;

if ( ProgrStatus = 'Optimal' ) then
    GMP::Solution::SetProgramStatus( OriginalGMP, 1, 'Optimal' ) ;
    GMP::Solution::SetSolverStatus( OriginalGMP, 1, 'NormalCompletion' ) ;

    GMP::Solution::SendToModel( gmpS, 1 ) ;

    GMP::Solution::SendToModelSelection( gmpM, 1, VariablesMasterProblem,
                                         AllSuffixNames );
    GMP::Solution::RetrieveFromModel( OriginalGMP, 1 );

    GMP::Solution::SetObjective( OriginalGMP, 1, BestObjective );
    GMP::Solution::SendToModel( OriginalGMP, 1 );
elseif ( ProgrStatus = 'Infeasible' ) then
    GMP::Solution::SetProgramStatus( OriginalGMP, 1, 'Infeasible' ) ;
    GMP::Solution::SetSolverStatus( OriginalGMP, 1, 'NormalCompletion' ) ;
elseif ( ProgrStatus = 'Unbounded' ) then
    GMP::Solution::SetProgramStatus( OriginalGMP, 1, 'Unbounded' ) ;
    GMP::Solution::SetSolverStatus( OriginalGMP, 1, 'NormalCompletion' ) ;
else
    GMP::Solution::SetProgramStatus( OriginalGMP, 1, 'ProgramNotSolved' ) ;
    GMP::Solution::SetSolverStatus( OriginalGMP, 1, 'SetupFailure' ) ;
endif;
```

21.5 Control parameters that influence the algorithm

This section

Some of the control parameters of Table 21.1 can be used to influence the behavior of the Benders' decomposition algorithm. We discuss these parameters in this section.

21.5.1 Primal versus dual subproblem

Parameter `UseDual`

In the textbook algorithm the dual of the subproblem is used. It is also possible to use the primal of the subproblem instead. This is controlled by the parameter `UseDual`. By default the Benders' decomposition algorithm uses the primal subproblem.

Dual solution

If the primal subproblem is solved and it appears to be feasible then the dual solution is used to construct an optimality cut. By the dual solution we mean the shadow prices of the constraints and the reduced costs of the variables in the primal subproblem.

Feasibility problem

If the primal subproblem is infeasible then another problem is solved to find a solution of minimum infeasibility (according to some measurement). The *feasibility problem* of $PS(x^*)$ (see Section 21.3) is denoted by $PFS(x^*)$ and defined by:

Minimize:

$$z$$

Subject to:

$$Qy - z \leq r - Tx^*$$
$$y \in \mathbb{R}_+^m$$
$$z \in \mathbb{R}$$

Here z is a scalar variable. The dual solution of this feasibility problem is used to create a feasibility cut which is added to the master problem.

Alternative feasibility problem

The feasibility problem above minimizes the maximum infeasibility among all constraints. It is also possible to minimize the sum of infeasibilities over all constraints; this is controlled by the parameter `UseMinMaxForFeasibilityProblem` which we discuss in Subsection 21.5.4. Also the parameter `NormalizationType` influences the formulation of the feasibility problem; see Subsection 21.5.3. Note that the feasibility problem is always feasible and bounded. Note further that if the optimal objective value of the feasibility problem is 0 or negative then the corresponding subproblem is feasible.

Relationship with parameter `FeasibilityOnly`

In the next subsection we discuss the parameter `FeasibilityOnly`. This parameter has a big influence on how the subproblem is created, for both the primal and dual subproblem. In some cases the subproblem can become a pure feasibility problem.

21.5.2 Subproblem as pure feasibility problem

Until now

By so far we assumed that the Benders' decomposition algorithm first tries to solve the subproblem to optimality to either conclude that the combined solution of the master problem and subproblem forms an optimal solution for the original problem, or to create an optimality cut that is added to the master problem. If the primal or dual subproblem appears to be infeasible or unbounded respectively, then a feasibility problem is solved (if we used the primal subproblem) or an unbounded extreme ray is calculated (if we used the dual subproblem) to create a feasibility cut.

Benders' subproblem always infeasible

For some problems the Benders' subproblem will (almost) always be infeasible unless an optimal solution of the original problem is found. For example, assume that the variables that become part of the subproblem have no objective coefficients. (In the MIP problem of Section 21.2 this is equivalent to the vector d being equal to 0.) In that case the Benders' decomposition algorithm tries to find a solution for the master problem that remains feasible if we also consider the part of the model that became the subproblem. The algorithm is finished if such a solution is found. Until then all subproblems will be infeasible. In

that case it is useless to try to solve the subproblem to optimality (which will always fail) but instead directly solve a feasibility problem for the subproblem.

Reformulation

It is possible to let the AIMMS automatically reformulate the original problem such that the variables that become part of the subproblem have no longer objective coefficients. (This reformulation exists only temporary while the function `GMP::Benders::CreateMasterProblem` is executed; the user will not notice anything inside his project.) For the MIP problem of Section 21.2 the reformulated problem becomes:

Minimize:

$$c^T x + \eta$$

Subject to:

$$\begin{aligned} d^T y - \eta &\le 0 \\ Ax &\le b \\ Tx + Qy &\le r \\ x &\in \mathbb{Z}_+^n \\ y &\in \mathbb{R}_+^m \\ \eta &\in \mathbb{R} \end{aligned}$$

If we assign the new continuous variable η, together with the integer variable x, to the master problem then the subproblem variables no longer have objective coefficients. As a consequence, the subproblem will always be infeasible (unless an optimal solution is found).

Parameter `FeasibilityOnly`

The parameter `FeasibilityOnly` can be used to control whether AIMMS should reformulate the original problem as explained above. AIMMS will do so if the value of this parameter equals 1, which is the default value. Also, if parameter `FeasibilityOnly` equals 1 then the Benders' decomposition algorithm will no longer solve the primal subproblem before solving the feasibility problem. Instead it will directly solve the feasibility problem.

Primal subproblem

After reformulating the original problem, the primal of the subproblem will be different from $PS(x^*)$ of Section 21.3, namely:

Minimize:

$$0$$

Subject to:

$$\begin{aligned} d^T y &\le \eta^* \\ Qy &\le r - Tx^* \\ y &\in \mathbb{R}_+^m \end{aligned}$$

We denote this primal subproblem by $PS'(x^*, \eta^*)$. The feasibility problem will also become slightly different, as compared to $PFS(x^*)$ of Subsection 21.5.1, namely:

Minimize:

$$z$$

Subject to:

$$\begin{aligned} d^T y - z &\leq \eta^* \\ Qy - z &\leq r - Tx^* \\ y &\in \mathbb{R}_+^m \\ z &\in \mathbb{R} \end{aligned}$$

We denote this feasibility problem by $PFS'(x^*, \eta^*)$. If the optimal objective value of this feasibility problem is 0 or negative then we have found an optimal solution for the original problem, and the Benders' decomposition algorithm terminates. Otherwise the dual solution of the feasibility problem is used to add a feasibility cut to the master problem, and the algorithm continues by solving the master problem.

Dual subproblem

We have seen before that if we use the dual of the subproblem and parameter `FeasibilityOnly` equals 0 then the Benders' decomposition algorithm will first solve the dual subproblem and, if that subproblem is infeasible, use an unbounded extreme ray to create a feasibility cut. If parameter `FeasibilityOnly` equals 1 then the algorithm follows a different route. Consider the dual formulation of the above problem, the feasibility problem for $PS'(x^*, \eta^*)$:

Maximize:

$$\pi^T(r - Tx^*) + \pi_0 \eta^*$$

Subject to:

$$\begin{aligned} \pi^T Q + \pi_0 d^T &\geq 0 \\ 1^T \pi + \pi_0 &= 1 \\ \pi, \pi_0 &\geq 0 \end{aligned}$$

Here 1^T denotes a vector of all 1's. We denote this problem by $DS'(x^*, \eta^*)$. This problem is always feasible and bounded. The Benders' decomposition algorithm uses this problem as the (dual) subproblem if the parameters `FeasibilityOnly` and `UseDual` equal 1. If the optimal objective value of this problem is 0 or negative then we have found an optimal solution for the original problem, and the Benders' decomposition algorithm terminates. Otherwise the solution of this problem is used to add a feasibility cut to the master problem, and the algorithm continues by solving the master problem.

Disadvantage

A serious disadvantage of reformulating the problem, as done in this section, is that a first feasible solution (which will be optimal) for the original problem will be found just before the Benders' decomposition algorithm terminates. This means that the "gap" between the lower and upper bound on the objective value is meaningless, and therefore this measurement of progress toward finding and proving optimality by the algorithm is not available. However, this

disadvantage only occurs when using the classic Benders' decomposition algorithm. For the modern approach in which only a single MIP problem is solved, see Section 21.6, the algorithm finds feasible solutions for the original problem during the solution process and therefore the "gap" exists.

21.5.3 Normalization of feasibility problem

Normalization

In the previous subsection we introduced the dual subproblem $DS'(x^*, \eta^*)$ which contains the normalization condition

$$1^T \pi + \pi_0 = 1. \tag{NC1}$$

In order to obtain better feasibility cuts, Fischetti et al. (in [Fi10]) proposed another normalization condition. The matrix T often contains null constraints which correspond to constraints that do not depend on x. These are "static" conditions in the subproblem that are always active. According to Fischetti et al. there is no reason to penalize the corresponding dual multiplier π_i. The new normalization condition then becomes

$$\sum_{i \in I(T)} \pi_i + \pi_0 = 1 \tag{NC2}$$

where $I(T)$ indexes the nonzero constraints of matrix T.

Parameter NormalizationType

The parameter `NormalizationType` controls which normalization condition is used. If it equals 0 then normalization condition (NC1) is used, else (NC2). The Benders' decomposition algorithm uses (NC2) by default because various computational experiments showed a better performance with this normalization condition.

Translation to primal subproblem

We can apply the normalization rule of Fischetti et al. also if we use the primal subproblem. In the corresponding feasibility problem, we then only add variable z for the nonzero rows of T. The relevant constraints in $PFS'(x^*, \eta^*)$ then become:

$$\begin{aligned} (Qy)_i - z_i &\leq r_i - (Tx^*)_i \qquad && i \in I(T) \\ (Qy)_i &\leq r_i && i \notin I(T) \end{aligned}$$

The feasibility problem can be normalized in this way regardless of the setting of parameter `FeasibilityOnly`.

Exception

In case the parameter `UseDual` equals 1 and the parameter `FeasibilityOnly` equals 0 then no feasibility problem is solved to derive a feasibility cut. Instead an unbounded extreme ray for the unbounded dual subproblem is used. Therefore, in that case the parameter `NormalizationType` is ignored.

21.5.4 Feasibility problem mode

Parameter UseMinMaxForFeasibilityProblem

The parameter UseMinMaxForFeasibilityProblem determines what kind of infeasibility is minimized: the maximum infeasibility among all constraints (value 1, the default) or the sum of infeasibilities over all constraints (value 0). If the sum of the infeasibilities over all constraints is used then also the normalization rule of Fischetti et al. can be used, as controlled by the parameter NormalizationType. This parameter is ignored if the parameter UseDual equals 1.

21.5.5 Tightening constraints

Illustrative example

If the Benders' master problem is created, using the function GMP::Benders::CreateMasterProblem, then AIMMS can try to automatically add valid constraints to the master problem that will cut off some infeasible solutions. This is best illustrated by the following MIP example.

Minimize:

$$\sum_i x_i$$

Subject to:

$$\begin{aligned} y_i &\le u_i x_i \qquad \forall i \\ \sum_i y_i &\ge b \\ x &\in \{0,1\} \\ y &\ge 0 \end{aligned}$$

We assume that u and b are strictly positive parameters. The binary variable x is assigned to the master problem and the continuous variable y to the subproblem. For this example, the initial master problem has no constraints (besides the integrality restriction on x) and therefore $x = 0$ is the optimal solution of the initial master problem. Clearly, for $x = 0$ our MIP example has no solution. Adding the constraint

$$\sum_i u_i x_i \ge b$$

to the master problem cuts off the $x = 0$ solution. Note that this constraint is redundant in the original MIP example. By adding these kind of master-problem-tightening constraints we hope that the Benders' decomposition algorithm requires less iterations to find an optimal solution.

Adding tightening constraints to the master problem is controlled by the parameter AddTighteningConstraints. If this parameter equals 1, its default, then AIMMS will try to find and add tightening constraints. Computational experiments indicate that in general the Benders' decomposition algorithm benefits from adding these tightening constraints.

Parameter AddTightening-Constraints

21.5.6 Using a starting point

The parameter UseStartingPointForMaster can be used to let the classic Benders' decomposition algorithm start from a "good" solution. This solution can be obtained from a heuristic and must be a feasible solution for the master problem. The solution should be copied into the level suffix of the problem variables before the Benders' decomposition algorithm is called. If this parameter is set to 1 then the algorithm will skip the solve of the first master problem. Instead, the master problem variable x^* will be fixed in the subproblem $PS(x^*)$ according to the starting point, and the algorithm will continue by solving the subproblem.

Parameter UseStarting-PointForMaster

21.5.7 Using the AIMMS Presolver

The Benders' decomposition algorithm can use the AIMMS Presolver at the start. In that case the algorithm will use the preprocessed model instead of the original model. By preprocessing the model it might become smaller and easier to solve. The parameter UsePresolver can be used to switch on the preprocessing step.

Parameter UsePresolver

21.6 Implementation of the modern algorithm

When solving a MIP problem, the classic Benders' decomposition algorithm often spends a large amount of time in solving the master MIP problems in which a significant amount of rework is done. In the modern approach only one single master MIP problem is solved. Whenever the solver finds a feasible integer solution for the master problem, the subproblem $PS(x^*)$ is solved after fixing the master problem variable x^* according to this integer solution.

Single MIP

Modern MIP solvers like CPLEX and GUROBI allow the user control over the solution process by so-called *callbacks*. Callbacks allow user code in AIMMS to be executed regularly during an optimization process. If the solver finds a new candidate integer solution then the user has the possibility to let the solver call one or more callback procedures. One of these callbacks is the callback for lazy constraints; that callback is used in the modern Benders' decomposition algorithm.

Callbacks

Feasible solutions

If no violated Benders' cut can be generated, after solving the subproblem, then we have found a feasible solution for the original problem and we can accept the current feasible integer solution as a "correct" solution for the master MIP problem. In the classic algorithm we would now be finished because we would know that no better solution of the original problem exists. In the modern algorithm we have to continue solving the master MIP problem because there might still exist a solution to the master MIP problem that results in a better solution for the original problem.

Lazy constraints

If a Benders' optimality or feasibility cut is found then this will be added as a so-called *lazy constraint* to the master MIP problem. Lazy constraints are constraints that represent one part of the model; without them the model would be incomplete. In this case the actual model that we want to solve is the original problem but we are solving the master MIP problem instead. The Benders' cuts represent the subproblem part of the model and we add them whenever we find one that is violated.

Implementation of DoBenders-Decomposition-SingleMIP

In the remainder of this section we show the implementation of the modern Benders' decomposition algorithm as implemented by the procedure DoBendersDecompositionSingleMIP which was introduced in Section 21.1. Similar to the procedure DoBendersDecompositionClassic, the procedure DoBendersDecompositionSingleMIP starts by making copies of its input arguments. Next the master problem and the subproblem are created. The parameters for the number of optimality and feasibility cuts are reset. Finally, the procedure calls another procedure, namely BendersAlgorithmSingleMIP, and finishes by deleting the master problem and the subproblem. As before we leave out parts of the code that handle details like creating a status file, for the sake of brevity and clarity.

```
OriginalGMP := MyGMP ;
VariablesMasterProblem := MyMasterVariables ;

! Create (Relaxed) Master problem.
gmpM := GMP::Benders::CreateMasterProblem( OriginalGMP, VariablesMasterProblem,
                        'BendersMasterProblem',
                        feasibilityOnly : FeasibilityOnly,
                        addConstraints : AddTighteningConstraints ) ;

! Create Subproblem.
gmpS := GMP::Benders::CreateSubProblem( OriginalGMP, gmpM, 'BendersSubProblem',
                        useDual : UseDual,
                        normalizationType : NormalizationType );

solsesS := GMP::Instance::CreateSolverSession( gmpS ) ;

NumberOfOptimalityCuts  := 0;
NumberOfFeasibilityCuts := 0;

! Start the actual Benders' decomposition algorithm.
BendersAlgorithmSingleMIP;
```

```
GMP::Instance::Delete( gmpM );
GMP::Instance::Delete( gmpS );
```

The procedure BendersAlgorithmSingleMIP

The BendersAlgorithmSingleMIP procedure initializes the algorithm by resetting the parameters for the number of iterations, etc. Then it calls the procedure SolveMasterMIP which does the actual work.

```
InitializeAlgorithmSingleMIP;

SolveMasterMIP;
```

The procedure SolveMasterMIP

The SolveMasterMIP procedure implements the actual Benders' decomposition algorithm using the modern approach. It first installs a lazy constraint callback for which the module implements four different versions. We assume that the control parameters have their default settings (see Table 21.1) in which case the procedure BendersCallbackLazyFeasOnlySingleMIP is installed. Next the master problem is solved and if a feasible solution is found, the subproblem is solved one last time to obtain a combined optimal solution for the original problem. Finally the algorithm terminates.

```
if ( UseDual ) then
    if ( FeasibilityOnly ) then
        GMP::Instance::SetCallbackAddLazyConstraint( gmpM,
                        'GMPBenders::BendersCallbackLazyFeasOnlyDualSingleMIP' );
    else
        GMP::Instance::SetCallbackAddLazyConstraint( gmpM,
                        'GMPBenders::BendersCallbackLazyOptAndFeasDualSingleMIP' );
    endif;
else
    if ( FeasibilityOnly ) then
        GMP::Instance::SetCallbackAddLazyConstraint( gmpM,
                        'GMPBenders::BendersCallbackLazyFeasOnlySingleMIP' );
    else
        GMP::Instance::SetCallbackAddLazyConstraint( gmpM,
                        'GMPBenders::BendersCallbackLazyOptAndFeasSingleMIP' );
    endif;
endif;

GMP::Instance::Solve( gmpM );

ProgramStatus := GMP::Solution::GetProgramStatus( gmpM, 1 ) ;

if ( ProgramStatus = 'Infeasible' ) then
    AlgorithmTerminateSingleMIP( 'Infeasible' );
else
    if ( FeasibilityOnly and not UseDual ) then
        ! Solve feasibility problem fixing the optimal solution of the
        ! Master problem.
        GMP::Solution::SendToModel( gmpM, 1 );

        ! Update feasibility problem and solve it.
        GMP::Benders::UpdateSubProblem( gmpF, gmpM, 1, round : 1 );
        GMP::Instance::Solve( gmpF );
```

```
    else
        ! Solve Subproblem fixing the optimal solution of the Master problem.
        GMP::Solution::SendToModel( gmpM, 1 );

        ! Update Subproblem and solve it.
        GMP::Benders::UpdateSubProblem( gmpS, gmpM, 1, round : 1 );
        GMP::Instance::Solve( gmpS );
    endif;

    AlgorithmTerminateSingleMIP( 'Optimal' );
endif;
```

The procedure BendersCallbackLazyFeasOnlySingleMIP

The callback procedure BendersCallbackLazyFeasOnlySingleMIP is called by the MIP solver whenever it finds a candidate integer solution for the master problem. This procedure retrieves the candidate integer solution from the MIP solver. Then it creates the feasibility problem for the (primal) subproblem if it does not exist yet. The feasibility problem is updated and solved. If its optimal objective value is larger than 0, indicating that the subproblem would have been infeasible, we add a feasibility cut as a lazy constraint to the master MIP. The MIP solver will not treat this candidate integer solution as a real solution. If the optimal objective value equals 0 (or is negative) then we do not add a lazy constraint in which case the MIP solver accepts the candidate solution as a real solution. Finally, the callback procedure returns 1 such that the solution process of the master MIP problem continues.

```
! Get MIP incumbent solution.
GMP::Solution::RetrieveFromSolverSession( ThisSession, 1 );
GMP::Solution::SendToModel( gmpM, 1 );

! Create feasibility problem corresponding to Subproblem (if it does not exist yet).
if ( not FeasibilityProblemCreated ) then
    gmpF := GMP::Instance::CreateFeasibility( gmpS, "FeasProb",
                            useMinMax : UseMinMaxForFeasibilityProblem );
    solsesF := GMP::Instance::CreateSolverSession( gmpF ) ;
    FeasibilityProblemCreated := 1;
endif;

! Update feasibility problem corresponding to Subproblem and solve it.
GMP::Benders::UpdateSubProblem( gmpF, gmpM, 1, round : 1 );

GMP::SolverSession::Execute( solsesF ) ;
GMP::Solution::RetrieveFromSolverSession( solsesF, 1 ) ;

! Check whether objective is 0 in which case optimality condition is satisfied.
ObjectiveFeasProblem := GMP::SolverSession::GetObjective( solsesF );

if ( ObjectiveFeasProblem <= BendersOptimalityTolerance ) then
    return 1;
endif;

! Add feasibility cut to the Master problem.
NumberOfFeasibilityCuts += 1;
GMP::SolverSession::AddBendersFeasibilityCut( ThisSession, gmpF, 1 );

return 1;
```

The procedure AlgorithmTerminateSingleMIP

The procedure AlgorithmTerminateSingleMIP is called to finish the Benders' decomposition algorithm. This procedure is called directly after the master MIP problem is solved. Its implementation is similar to that of the procedure AlgorithmTerminate of Section 21.4 and therefore omitted.

21.7 Implementation of the two phase algorithm

First phase

The Benders' module also implements a two phase algorithm for MIP problems. In the first phase it solves the relaxed problem in which the integer variables become continuous. The resulting relaxed MIP problem is then solved using the classic Benders' decomposition algorithm in order to find Benders' cuts.

Second phase

The second phase solves the original MIP problem. The master problem created in the first phase is also used in the second phase but without relaxing the integer variables. The Benders' cuts that were added during the first phase are not removed; these cuts are still valid. In general the relaxed MIP problem can be solved more efficiently than the MIP problem using Benders' decomposition, and the hope is that by adding the Benders' cuts found during the first phase, the Benders' decomposition algorithm needs considerably less iterations in the second phase to solve the original MIP problem.

Implementation of DoBendersDecompositionTwoPhase

The procedure DoBendersDecompositionTwoPhase implements the two phase algorithm. It starts by making copies of its first two input arguments. Next the master problem and the subproblem are created. The parameters for the number of optimality and feasibility cuts are reset. The problem type of the master problem is changed from 'MIP' to 'RMIP' which basically changes the integer variables into continuous variables. The procedure BendersAlgorithm then solves the relaxed problem using the classic Benders' decomposition algorithm; see Section 21.4 for its implementation. After checking the program status of the relaxed master problem the algorithm continues by switching the problem type of the master problem back to 'MIP'. Next the original problem is solved using either procedure BendersAlgorithmSingleMIP (Section 21.6) or BendersAlgorithm (Section 21.4). The algorithm ends by deleting the master problem and the subproblem. As before we leave out parts of the code that handle details like creating a status file, for the sake of brevity and clarity.

```
OriginalGMP := MyGMP ;
VariablesMasterProblem := MyMasterVariables ;

! Create (Relaxed) Master problem.
gmpM := GMP::Benders::CreateMasterProblem( OriginalGMP, VariablesMasterProblem,
                        'BendersMasterProblem',
                        feasibilityOnly : FeasibilityOnly,
                        addConstraints : AddTighteningConstraints ) ;

! Create Subproblem.
gmpS := GMP::Benders::CreateSubProblem( OriginalGMP, gmpM, 'BendersSubProblem',
```

```
                        useDual : UseDual,
                        normalizationType : NormalizationType );

solsesS := GMP::Instance::CreateSolverSession( gmpS ) ;

NumberOfOptimalityCuts  := 0;
NumberOfFeasibilityCuts := 0;

! Start the classic Benders' decomposition algorithm for the relaxed Master
! MIP problem.
GMP::Instance::SetMathematicalProgrammingType( gmpM, 'RMIP' );

IterationLimit := IterationLimitPhaseSingle;

BendersAlgorithm;

ProgramStatus := GMP::Solution::GetProgramStatus( OriginalGMP, 1 );

if ( ProgramStatus = 'Infeasible' or
     ProgramStatus = 'Unbounded'  ) then
    DoPhaseTwo := 0;
endif;

if ( DoPhaseTwo ) then
    ! Switch back math program type.
    GMP::Instance::SetMathematicalProgrammingType( gmpM, 'MIP' );

    if ( UseSingleMIP ) then
        ! Start the Single MIP Tree Benders' decomposition algorithm.
        BendersAlgorithmSingleMIP;
    else
        IterationLimit := IterationLimitPhaseTwo;

        BendersAlgorithm;
    endif;
endif;

GMP::Instance::Delete( gmpM );
GMP::Instance::Delete( gmpS );
```

Iteration and time limit

The section in the Benders' module for the two phase algorithm contains two extra control parameters for setting the iteration and time limit used by the classic Benders' decomposition algorithm in the second phase. These parameters are `IterationLimitPhaseTwo` and `TimeLimitPhaseTwo` respectively. The parameters `IterationLimit` and `TimeLimit` are used in the first phase. In some cases it might be a good strategy to limit the number of iterations (or the running time) during the first phase. The two phase algorithm will then still find a global optimal solution of the original problem as long as the second phase terminates normally. If the modern approach (with a single MIP tree) is used in the second phase then the general solver options `iteration_limit` and `time_limit` are used for the second phase.

Chapter 22

Constraint Programming

Introduction

Constraint Programming is a relatively new paradigm used to model and solve combinatorial optimization problems. It is most effective on highly combinatorial problem domains such as timetabling, sequencing, and resource-constrained scheduling. Successful industrial applications utilizing constraint programming technology include the gate allocation system at Hong Kong airport, the yard planning system at the port of Singapore, and the train timetable generation of Dutch Railways.

Constraint Programming in AIMMS

This chapter discusses the special identifier types and language constructs that AIMMS offers for formulating and solving constraint programming problems. We will see that constraint programming offers a much wider range of modeling constructs than, for example, integer linear programming or nonlinear programming. Different variable types can be used, while restrictions can be formed using arbitrary algebraic and logical expressions or by the use of special constraint types, such as alldifferent. In addition, AIMMS offers a specific syntax to express scheduling problems in an intuitive way, taking advantage of the algorithmic power that underlies constraint-based scheduling.

This chapter

In this chapter, the basic constraint programming concepts are first presented, including different variable types and restrictions in Section 22.1. Section 22.2 discusses the AIMMS syntax for modeling constraint-based scheduling problems. The final section of this chapter discusses issues related to modeling and solving constraint programs in AIMMS.

Literature

An in-depth discussion on constraint programming is given in [Ro06] and more details on constraint-based scheduling can be found in [Ba01].

Online resources

First, the Association for Constraint Programming organizes an annual summer school, the material for which is posted online. This material can be accessed at `http://4c.ucc.ie/a4cp/`. The CPAIOR conference series organizes tutorials alongside each event, the materials of which are posted online. The CPAIOR 2009 tutorial provides an introduction to constraint programming and hybrid methods, and available online at `http://www.tepper.cmu.edu/cpaior09`.

22.1 Constraint Programming essentials

Variables

In constraint programming, models are built using variables and constraints, and as such is similar to integer programming. One fundamental difference is that, in integer programming, the range of a variable is specified and maintained as an interval, while in constraint programming, the variable range is maintained explicitly as a set of elements. Note that in the constraint programming literature, the range of a variable is commonly referred to as its *domain*.

Constraints

As a consequence of this explicit range representation, constraint programming can offer a wide variety of constraint types. Most constraint programming solvers allow constraints to be defined by arbitrary expressions that combine algebraic or logical operators. Moreover, meta-constraints can be formulated by interpreting the logical value of an expression as a boolean value in a logical relation, or as a binary value in an algebraic relation. For example, to express that every two distinct tasks i and j, from a set of tasks T with respective starting times s_i, s_j and durations d_i, d_j, should not overlap in time, we can use logical disjunctions:

$$(s_i + d_i \leq s_j) \vee (s_j + d_j \leq s_i),\ \forall i, j \in T, i \neq j. \tag{22.1}$$

As another example, we can set a restriction such that no more than half the variables from $x_1, \ldots, x_n$ are assigned to a specific value v as $\sum_{i=1}^{n}(x_i = v) \leq 0.5n$.

Global constraints

In addition, constraint programming offers special symbolic constraints that are called *global constraints*. These constraints can be defined over an arbitrary set of variables, and encapsulate a combinatorial structure that is exploited during the solving process. The constraint `cp::AllDifferent`$(x_1 \ldots, x_n)$ is an example of such a global constraint. This global constraint requires the variables $x_1 \ldots, x_n$ to take distinct values.

Solving

The solving process underpinning constraint programming combines systematic search with inference techniques. The systematic search implicitly enumerates all possible variable-value combinations, thus defining a search tree in which the root represents the original problem to be solved. At each node in the search tree, an inference is made by means of *domain filtering* and *constraint propagation*. Each constraint in the model has an associated domain filtering algorithm that removes provably inconsistent values from the variable domains. Here, a domain value is inconsistent if it does not belong to any solution of the constraint. The updated domains are then communicated to the other constraints, whose domain filtering algorithms in turn become active; this is the constraint propagation process. In practice, the most effective filtering algorithms are those associated with global constraints. Therefore,

most practical applications that are to be solved with constraint programming are formulated using global constraints.

Application domains

Constraint programming can be particularly effective with highly combinatorial problem domains, such as timetabling or resource-constrained scheduling. For such problems an integer programming model may be non-intuitive to express. Moreover, the associated continuous relaxation may be quite weak, which makes it much harder to find a provably optimal solution. For example, again consider two tasks A and B that must not overlap in time. In integer programming one can introduce two binary variables, y_{AB} and y_{BA}, representing that task A must be processed either before or after, task B. The non-overlapping constraint can then be expressed as $y_{AB} + y_{BA} = 1$, for which a continuous linear relaxation may assign a solution $y_{AB} = y_{BA} = 0.5$, which is not very informative. In contrast, the non-overlapping requirement can be handled very effectively using a specific 'sequential resource' scheduling constraint in constraint programming that effectively groups together all pairwise logical disjunctions in (22.1) above. Such a constraint is also referred to as a disjunctive or unary constraint in the constraint programming literature.

Designing models

The expressiveness of constraint programming offers a powerful modeling environment, albeit one that comes with a caveat. Namely, that different syntactically equivalent formulations may yield quite different solving times. For example, an alternative to the constraint `cp::AllDifferent`$(x_1, x_2, \ldots, x_n)$ is its decomposition into pairwise not-equal constraints $x_i \neq x_j$ for $1 \leq i < j \leq n$. The domain filtering algorithm for `cp::AllDifferent` provably removes more inconsistent domain values than the individual not-equal constraints, which results in much smaller search trees and faster solution times. Therefore, when designing a constraint programming model, one should be aware of the effect that different formulations can have on the solving time. In most situations, it is advisable to apply global constraints to exploit their algorithmic power.

22.1.1 Variables in constraint programming

Variables of a constraint program

A constraint programming problem is made up of discrete variables and constraints over these discrete variables. A discrete variable is a variable that takes on a discrete value. AIMMS supports two base types of discrete variables for constraint programming. The first type of variable is the integer variable; an ordinary variable with a range formulated such as {`a..b`} where `a` and `b` are numbers or references to parameters (see Chapter 14). Such variables can also be used in `MIP` problems. The second type of variable is the element variable, which will be further detailed in this section. This type of variable can only be used in a constraint programming problem. These two types of variable can be combined in a third type, called an integer element variable, which supports

the operations that are defined for both the integer variable and the element variable.

22.1.1.1 ElementVariable declaration and attributes

An element variable is a variable that takes an element as its value. It can have the attributes specified in Table 22.1. The attributes `IndexDomain`, `Priority`, `NonvarStatus`, `Text`, `Comment` are the same as those for the variables introduced in Chapter 14.

Attribute	Value-type	See also page
`IndexDomain`	*index-domain*	208
`Range`	*named set*	
`Default`	*constant-expression*	44, 210
`Priority`	*expression*	211
`NonvarStatus`	*expression*	212
`Property`	`NoSave`	34
`Text`	*string*	19
`Comment`	*comment string*	19
`Definition`	*expression*	211

Table 22.1: `ElementVariable` attributes

The `Range` attribute

The range of an element variable is a one-dimensional set, similar to the range of an element parameter. This attribute must be a set identifier; and this permits the compiler to verify the semantics when element variables are used in expressions. This attribute is mandatory.

The `Default` attribute

The attribute `default` of an element variable is a quoted element. This attribute is not mandatory.

The `Definition` attribute

The `Definition` attribute of an element variable is similar to the definition attribute of a variable, see also page 211, except that its value is an element and the resulting element must lie inside the range of the element variable. This attribute is not mandatory.

The `Property` attribute

The following properties are available to element variables:

- **Nosave** When set, this property indicates that the element variable is not to be saved in cases.
- **EmptyElementAllowed** When set, this property indicates that in a feasible solution, the value of this variable can, but need not, be the empty element `''`. When the range of the element variable is a subset of the set

Integers, this property is not available. In the following example, for the element variable eV, not selecting an element from S, is a valid choice, but this choice forces the integer variable X to 0.

```
ElementVariable eV {
    Range       :  S;
    Property    :  EmptyElementAllowed;
}
Constraint Force_X_to_zero_when_no_choice_for_eV {
   Definition   :  if eV = '' then X = 0 endif;
}
```

This attribute is not mandatory.

Element translation

Constraint programming solvers only use integer variables, and AIMMS translates an element variable, say eV with range the set S containing n elements into a integer variable, say v, with range $\{0..n-1\}$. By design, this translation leaves no room for the empty element '', and subsequently, in a feasible solution, the empty element is no part of it. In order to permit the explicit consideration of the empty element as part of a solution, the property EmptyElementAllowed can be set for eV. In that case the range of v is $\{0..n\}$ whereby the value 0 corresponds to the empty element.

22.1.1.2 Selecting a variable type

Choosing variable type

When there are multiple types of objects, such as integer variables and element variables in AIMMS, the following two questions naturally arise:

1. How to choose between the various types?
2. Can these types be combined?

The answers to these questions are as follows:

1. You may want to base the choice of types of variables on the operations that can be performed meaningfully on these types. Which operation is appropriate for which variable type is described below.
2. An identifier can have both the 'integer variable' and 'element variable' types and is then called an 'integer element variable'. This is created as an element variable with a named subset of the predeclared set Integers as its range.

Operations on variables

The operations on variables that are interesting in constraint programming are:

- **Numeric** operations, such as multiplication, addition, and taking the absolute value. These operations are applicable to integer variables and to integer element variables.
- **Index** operations; selecting an element of an indexed parameter or variable. An element variable eV can be an argument of a parameter P or

a variable X in expressions such as P(eV) or X(eV). These operations are applicable to all element variables. In the Constraint Programming literature, such operations are often implemented using so-called element constraints.

- **Compare, subtract, min and max** operations. These operations are applicable to all discrete variables, including element variables. For element variables, AIMMS uses the ordering of sets, see Section 3.2.

All of the above operations are available with integer element variables.

Contiguous range

In order to limit an element variable to a contiguous subset of its named range, element valued suffixes .lower and .upper can be used. In the example below, the assignment to eV.Lower restricts the variable eV to the contiguous set {c..e}.

```
Set someLetters {
    Definition   : data { a, b, c, d, e };
}
ElementVariable eV {
    Range        : someLetters;
}
Procedure Restrict_eV {
    Body         : eV.lower := 'c';
}
```

The specification of non-contiguous ranges, informally known as ranges with holes, is detailed in the next subsection.

22.1.2 Constraints in constraint programming

Introduction

The constraints in constraint programming allow a rich variety of restrictions to be placed on the variables in a constraint program, ranging from direct domain restrictions on the variables to global constraints that come with powerful propagation algorithms.

Domain restrictions

A domain restriction restricts the domain of a single variable, or of multiple variables, and is specified using the IN operator. For example, we can restrict the domain of an element variable eV as follows:

```
Constraint DomRestr1 {
    Definition   :  eV in setA;
}
```

When we apply the IN operator to multiple variables, we can define a constraint by explicitly listing all tuples that are allowed. For example:

```
Constraint DomRestr2 {
    Definition   :  (eV1, eV2, eV3) in ThreeDimRelation;
    Comment      : "ThreeDimRelation contains all allowed tuples";
}
```

```
Constraint DomRestr3 {
    Definition   :  not( (eV1, eV2, eV3) in ComplementRelation );
    Comment      : "ComplementRelation contains all forbidden tuples";
}
```

In constraint `DomRestr2` above, the three element variables are restricted to elements from the set of allowed tuples defined by `ThreeDimRelation`. Alternatively, we can define such a restriction using the complement, i.e., a list of forbidden tuples, as with constraint `DomRestr3`. In constraint programming, these constraints are also known as *Table* constraints; the data for these constraints resemble tables in a relational database.

Algebraic restrictions

The following operations are permitted on discrete variables, resulting in expressions that can be used in constraint programming constraints:

1. The binary `min(a,b)`, `max(a,b)` and the iterative `min(i,x(i))`, `max(i,x(i))` can both be used,
2. multiplication `*`, addition `+`, subtraction `-`, absolute value `abs` and square `sqr`,
3. integer division `div(a,b)`, integer modulo `mod(a,b)`,
4. floating point division `/`, and
5. indexing: an element variable is used as an argument of another parameter or variable, `P(eV)`, `V(eV)`,

Note that the operation must be meaningful for the variable type, see page 374. These expressions can be compared, using the operators `<=`, `<`, `=`, `<>`, `>`, and `>=` to create algebraic restrictions. Simple examples of algebraic constraints, taken from Einstein's Logic Puzzle, are presented below.

```
Constraint Clue15 {
    Definition   :  abs( Smoke('Blends') - Drink('Water') ) = 1;
    Comment      :  "The man who smokes Blends has a neighbor who drinks water.";
}
Constraint TheQuestion {
    Definition   :  National(eV)=Pet('Fish');
    Comment      :  "Who owns the pet fish? Result stored in element variable eV";
}
```

Combining restrictions

The constraints above can be combined to create other constraints called meta-constraints. Meta-constraints can be formed by using the scalar operators `AND`, `OR`, `XOR`, `NOT` and `IF-THEN-ELSE-ENDIF`. For example:

```
Constraint OneTaskComesBeforeTheOther {
    Definition   : {
        ( StartA + DurA <= StartB ) or
        ( StartB + DurB <= StartA )
    }
}
```

In addition, restrictions can be combined into meta-constraints using the iterative operators `FORALL` and `EXISTS`. Moreover, restrictions can be counted using

the iterative operator SUM and the result compared with another value. Finally, meta-constraints are restrictions themselves, they can be combined into even more complex meta-constraints. The following example is a variable definition, in which the collection of constraints (Finish(i) > Deadline(i)) is used to form a meta-constraint.

```
Variable TotalTardinessCost {
    Definition  :  Sum( i, TardinessCost(i) | ( Finish(i) > Deadline(i) ) );
}
```

In the following example, the binary variable y gets the value 1 if each X(i) is greater than P(i).

```
Constraint Ydef {
    Definition   :  y = FORALL( i, X(i) > P(i) );
}
```

From the Steel Mill example, we can model that we do not want more than two colors for each slab by the following nested usage of meta-constraints:

```
Constraint EnhancedColorCst {
    IndexDomain  :  (sl);
    Definition   :  sum( c, EXISTS (o in ColorOrders(c), SlabOfOrder(o)=sl)) <= 2;
}
```

Global constraints

AIMMS supports the global constraints presented in Table 22.2. These global constraints come with powerful filtering techniques that may significantly reduce the search tree and thus the time needed to solve a problem.
The example below illustrates the use of the global constraint cp::AllDifferent as used in the Latin square completion problem. A Latin square of order n is an $n \times n$ matrix where the values are in the range $\{1..n\}$ and distinct over each row and column.

```
Constraint RowsAllDifferent {
    IndexDomain  :  r;
    Definition   :  cp::AllDifferent( c, Entry(r, c) );
}
Constraint ColsAllDifferent {
    IndexDomain  :  c;
    Definition   :  cp::AllDifferent( r, Entry(r, c) );
}
```

Additional examples of global constraints are present in the AIMMS Function Reference. Unless stated otherwise in the function reference, global constraints can also be used outside of constraints definitions, for example in assignments or parameter definitions.

Global constraint	Meaning
cp::AllDifferent(i,x_i)	The x_i must have distinct values. $\forall i,j \mid i \neq j : x_i \neq x_j$
cp::Count(i,x_i,c,$\otimes$,y)	The number of x_i related to c is y. $\sum_i (x_i = c) \otimes y$ where $\otimes \in \{\leq, \geq, =, >, <, \neq\}$
cp::Cardinality(i,x_i, j,c_j,y_j)	The number of x_i equal to c_j is y_j. $\forall j : \sum_i (x_i = c_j) = y_j$
cp::Sequence(i,x_i, S,q,l,u)	The number of $x_i \in S$ for each subsequence of length q is between l and u. $\forall i = 1..n-q+1 :$ $l \leq \sum_{j=i}^{i+q-1} (x_j \in S) \leq u$
cp::Channel(i,x_i, j,y_j)	Channel variable $x_i \rightarrow J$ to $y_j \rightarrow I$ $\forall i,j : x_i = j \Leftrightarrow y_j = i$
cp::Lexicographic(i,x_i,y_i)	x is lexicographically before y $\exists i : \forall j < i : x_j = y_j \wedge x_i < y_i$
cp::BinPacking(i,l_i, j,a_j,s_j)	Assign object j of known size s_j to bin $a_j \rightarrow I$. Size of bin $i \in I$ is l_i. $\forall i : \sum_{j \mid a_j = i} s_j \leq l_i$

Table 22.2: Global constraints

Global constraint vector arguments

These global constraints have vectors as arguments. The size of a vector is defined by a preceding index binding argument. Further information on index binding can be found in the Chapter on Index Binding 9. Such a vector can be a vector of elements, for example the fourth argument of cp::Cardinality. In a vector of elements, the empty element '' is not allowed; comparison of an element variable against the empty element is not supported.

Basic scheduling constraints

AIMMS offers support for both basic scheduling and advanced scheduling. Advanced scheduling will be detailed in the next section but, for basic scheduling, AIMMS offers the following two global constraints:

1. The global constraint cp::SequentialSchedule(j, s_j, d_j, e_j) ensures that two distinct jobs do not overlap where job j has start time s_j, duration d_j and end time e_j. This constraint is equivalent to:
   - $\forall i,j,i \neq j : (s_i + d_i \leq s_j) \vee (s_j + d_j \leq s_i)$.
   - $\forall j : s_j + d_j = e_j$

   This and similar constraints are also known as unary or disjunctive constraints within the Constraint Programming literature.
2. The global constraint cp::ParallelSchedule(l, u, j, s_j, d_j, e_j, h_j) allows a single resource to handle multiple jobs, within limits l and u, at the same time. Here job j has start time s_j, duration d_j, end time e_j and

resource consumption (height) h_j. This constraint is equivalent to:

- $\forall t : l \leq \sum_{j|s_j \leq t < e_j} h_j \leq u$.
- $\forall j : s_j + d_j = e_j$

This and similar constraints are also known as `cumulative` constraints within the Constraint Programming literature.

22.2 Scheduling problems

Introduction

Resource-constrained scheduling is a key application area of constraint programming. Most constraint programming systems contain special syntactical constructs to formulate such problems, allowing the use of specialized inference algorithms. In Section 22.1.2 we have already seen two examples of global constraints for scheduling: `cp::SequentialSchedule` and `cp::ParallelSchedule`. For more complex scheduling problems that contain, for example, sequence-dependent setup times between activities, or specific precedence relations, the use of more advanced scheduling algorithms is advisable. These algorithms cannot be offered by the stand-alone global constraints `cp::SequentialSchedule` and `cp::ParallelSchedule`, but can be accessed by formulating the problem using *activities* and *resources* in AIMMS.

Activities

Activities correspond to the execution of objects in scheduling problems, e.g., processing an order, working a shift, or performing a loading operation. They can be viewed as the variables of a scheduling problem, since we must decide on their position in the schedule. Common attributes associated to an activity are its begin, end, length, and size. Further, it is often convenient to distinguish mandatory and optional activities, which allows to consider the presence of an activity. In AIMMS, the properties begin, end, length, size, and presence of an activity can be used as variables in other parts of the model. It is also possible to build models using nested activities, where meta-activities group together a number of sub-activities, for example in the context of project planning.

Resources

Resources correspond to the assets that are available to execute the activities, e.g., the capacity of a machine, the volume of a truck, or the number of available employees. Resources can be viewed as the constraints of a scheduling problem. The main attributes of a resource are its capacity, its activity level, and the set of activities that require the resource in order to be executed. That is, during the execution of the schedule, we must ensure that the resource activity level is always within its capacity. Note that while a resource depends on a set of activities, an activity can impact on one or more resources at the same time.

An activity changes the resource activity level

A resource starts with a default activity level of 0, corresponding to full available capacity, or a user-specified initial value. During the execution of the schedule, activities will influence the resource activity level. The viewpoint chosen in AIMMS is that an activity changes the activity level of a resource when it begins, and/or when it ends. This enables one to model many common situations. For example, when an activity corresponds to a loading operation, and the resource corresponds to a truck load, the activity will change the activity level of the resource with the volume of the load at its start, but there is no change in the resource activity level when finishing this activity. For sequential resources the capacity is 1, and each activity will change the resource activity level by $+1$ when it begins, and by -1 when it ends. For example, when an activity corresponds to a visit operation, and the resource corresponds to a truck, the activity level of the resource will be decreased by 1 at the beginning of the visit, and increased by 1 at the end.

Schedule domains

The timeline on which activities are scheduled, is the so-called *schedule domain*. A schedule domain is a finite set of timeslots. Each activity and resource has its own schedule domain.

The problem schedule domain

The schedule domain of the entire problem, the *problem schedule domain*, is a named superset of each of these schedule domains. Unless overridden, it is based on the schedule domains of the activities and resources.

Handling schedule domains

An activity is only considered active during a timeslot t if t is in the schedule domain of that activity, and it is in the schedule domain of each resource for which it is scheduled. Thus, for each individual activity, AIMMS passes the intersection of these schedule domains to the constraint programming solver.

Additional restrictions

Most scheduling problems contain several side constraints in addition to the resource constraints. Examples include precedence relations between activities, release dates or deadlines, and sequence-dependent setup times. Such constraints can be specified using global scheduling constraints or in the attribute forms of activities and resources. Constraint programming solvers can take an extra algorithmic advantage of such constraints when they are presented in this manner.

22.2.1 Activity

On the one hand, an activity can be seen as consisting of five variables that can be accessed by the suffixes: `.Begin`, `.End`, `.Length`, `.Size` and `.Present`. These variables represent the begin, end, length (difference between end and begin), size (number of active slots) and presence of an activity. These variables can be used inside constraints, for example `myActivity.End <= myDeadLine+1`. On the other hand, an activity is defined using its attributes as presented in Ta-

ble 22.3. We will first discuss the attributes of an activity, and then these suffixes in more detail.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	208
ScheduleDomain	*set range or expression*	
Property	Optional, NoSave	
Length	*expression*	
Size	*expression*	
Priority	*reference*	211
Text	*string*	19
Comment	*comment string*	19

Table 22.3: Activity attributes

The ScheduleDomain attribute

The activity is scheduled in time slots in the ScheduleDomain. This is an expression resulting in a one-dimensional set, or a set-valued range. The resulting set need not be a subset of the predeclared set Integers; it can be any one-dimensional set, for instance a Calendar, see Section 33.2. Consider the following examples of the attribute schedule domain:

```
Activity a {
    ScheduleDomain  :  yearCal;
    Comment         : {
        "a can be scheduled during any period
        in the calendar yearCal"
    }
}
Activity b {
    IndexDomain     :  i;
    ScheduleDomain  :  possiblePeriods(i);
    Comment         : {
        "b(i) can be scheduled only during the
        periods possiblePeriods(i)"
    }
}
Activity c {
    IndexDomain     :  i;
    ScheduleDomain  : {
        {ReleaseDate(i)..PastDeadline(i)}
    }
    Comment         : {
        "c(i) must start on or after ReleaseDate(i)
         c(i) must finish before PastDeadline(i)"
    }
}
```

The ScheduleDomain attribute is mandatory.

Singleton schedule domain

An activity with a singleton schedule domain and a length of 1 can be used to model an event. Such an activity is scheduled during the single element in the schedule domain. Because the schedule domain is a single element, the value of the suffixes `.Begin` and `.End` of the activity will be set to that single element and the element thereafter respectively in a feasible solution. Note that this is possible for all elements except for the last element in the problem schedule domain; a nonzero length would then require the `.End` to be after the problem schedule domain. Consider the following example:

```
Activity weekendActivities {
    IndexDomain     : {
        d | ( TimeslotCharacteristic( d, 'weekday' ) = 6 or
        TimeslotCharacteristic( d, 'weekday' ) = 7    ) and
        d <> last( dayCalendar )
    }
    ScheduleDomain  : {
        { d .. d }
    }
    Length          :  1;
    Comment         :  "d is an index in a calendar";
}
```

Scheduling the activity `weekendActivities` in a sequential resource will block other activities for that resource during the weekend.

The Property attribute

An activity can have the properties `Optional`, `Contiguous` and `NoSave`.

- **`Optional`** When an activity has the property `Optional`, it may or may not be scheduled. If the property `Optional` is not specified, then the activity will always be scheduled.
- **`Contiguous`** When an activity has the property `Contiguous`, all elements from `.Begin` up to but not including `.End` in the problem schedule domain must be in its own schedule domain.
- **`NoSave`** When an activity has the property `NoSave`, it will not be saved in cases.

This attribute is not mandatory.

The Length and Size attributes

When an activity is present, the `Length` attribute defines the length of the activity, and the `Size` attribute defines its size. The length of an activity is the difference between its end and its begin. The size of an activity is the number of periods, in which that activity is active from begin up to but not including its end. For example, a non-contiguous activity which `.Begin`s on Friday, `.End`s on Tuesday, and is not active during the weekend has a

- `.Length` of 4 days, and
- `.Size` of 2 days.

The numeric expressions entered at the `Length` and `Size` attributes may involve other discrete variables. These attributes are not mandatory.

For a contiguous activity we have that the .Length is equal to the .Size. Conversely, with a constraint a.Length=a.Size we have that a is contiguous, but the propagation may be less efficient.

The Priority attribute

The Priority attribute applies to all the discrete variables defined by an activity. To these variables it has the same meaning as for integer variables, see page 211. This attribute is not mandatory.

The suffixes of activities

An activity is made up of the following suffixes .Begin, .End, .Length, .Size and .Present. Each of these suffixes is a discrete variable and can be used in constraints.

The suffixes .Begin and .End

The suffixes .Begin and .End are element valued variables. When scheduled, the activity takes place from period .Begin up to but not including period .End. For a present activity a, in a feasible solution:

- a.Begin is an element in the schedule domain of the activity. The range of this element variable is the *smallest* named set encompassing the activity schedule domain.
- a.End is an element in the schedule domain of the problem, and, depending on the .Length of a, with the following additional requirement:
  - When the length of activity a is zero, a.End=a.Begin holds, and they are both in the activity schedule domain.
  - When the length of activity a is greater than 0, the element before a.End is in the activity schedule domain.

  The range of this element variable is the *root* set of the activity schedule domain.

Comparison of the .Begin and .End suffixes of two activities a and b inside a constraint definition will take place on the problem schedule domain, for instance in a constraint like a.End <= b.Begin. Outside constraint definitions these suffixes follow the rules of element comparison specified in Section 6.2.3.

The suffixes .Length and .Size

The suffixes .Length and .Size are nonnegative integer variables. The .Length of an activity is defined as .End - .Begin. The .Size of an activity is the number of timeslots in the schedule domain of the activity in the range [.Begin, .End). When the attribute Length or Size is non-empty, AIMMS will generate a defining constraint for the suffix .Length resp. .Size like the definition attribute of a variable, see Page 211. When the schedule domain of an activity is a calendar or a subset thereof, the unit of each of the .Length and .Size suffixes is the unit of the calendar.

The suffix .Present

The suffix .Present is a binary variable with default 0. For optional activities this variable is 1 when the activity is scheduled and 0 when it is not. For non-optional activities this variable is initialized with the value 1.

suffixes of absent activities in constraints

The value of one of the suffixes .Begin, .End, .Length, and .Size is not defined when the corresponding activity is absent. However, in order to satisfy constraints where such a suffix is used for an absent activity, a value is chosen: the socalled *absent* value. For the suffixes .Length, and .Size, the absent value is 0. For the suffixes .Begin and .End this depends on the problem schedule domain:

- If the problem schedule domain is a subset of Integers, the absent value is 0.
- Otherwise, the absent value of the suffixes .Begin and .End is ''.

To override the absent value use one of the following functions:

- cp::ActivityBegin,
- cp::ActivityEnd,
- cp::ActivityLength, or
- cp::ActivitySize.

Suffixes of optional activities in constraints

The value of the suffix .present is defined for an absent activity as 0. However the values of the other suffixes of an absent activity are not defined. To enable the constraining of the values of those suffixes in constraints several formulation alternatives are available. As an example of these alternatives, consider an activity act whereby we want to enforce its length to be 7 if it is present.

1. Enforce the length constraint conditionally on the presence of activity act:

```
if  act.present then
    act.length = 7
endif
```

2. The cp::ActivityLength function returns the length of a present activity or its second argument if it is not present:

```
cp::ActivityLength( act, 7 ) = 7
```

3. If we simply want to set the value of the .Length or .Size suffix, we can use the Length or Size attribute as follows.

```
Activity act {
    ScheduleDomain  :  ...;
    Property        :  optional;
    Length          :  7;
}
```

Each of the above formulation alternatives has its own merits.

1. The merit of this alternative is that it is general and can also be used to state for instance that the length of act is 7 or 11 when present:

```
if  act.present then
    act.Length = 7 or act.Length = 11
endif
```

2. The merit of this alternative is that it allows the solver to make stronger propagations and thus potentially reduce solution time.
3. The merit of this alternative is that is does not force the model builder to take the optionality of act into account when defining its length. AIMMS will make sure the length definition is translated to alternative 1 or 2 as appropriate.

Solution values of absent activities

The value of the suffixes .Begin, .End, .Length, and .Size of an absent activity in a feasible schedule are meaningless and should not be used in further computations.

Absent versus 0-length activities

Even though no work is done for both absent and 0-length activities, there is a difference in their usage. Let us consider the following two examples:

- Selection of an activity from alternatives; Consider a collection of activities from which we need to select one. This is easily and efficiently achieved by setting the property Optional to the activity. The ones not selected become absent in a solution.
- Consider two collections of activities, whereby the n activities in the first collection all need to be completed before the m activities in the second collection can start. We can model this directly by $n \times m$ precedence constrains. Another way to model this is by introducing an extra activity, say Milestone, of length zero. With this Milestone we only need $n + m$ precedence constraints.

To facilitate above and other examples of scheduling, the suffixes .Present and .Length are supported independently.

Relation between suffixes of activities

Please note, for an activity act, the following relation is implicitly defined:

```
if act.Present then
   act.Begin + act.Length = act.End
endif
if act.Present then
   act.Size <= act.Length
endif
```

22.2.2 Resource

A resource schedules activities by acting as a constraint on the activities it schedules. A feasible resource requires the above implicit constraints on the suffixes of the activities it schedules and the constraints implied by its attributes as discussed below.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	208
Usage	Parallel or Sequential	
ScheduleDomain	*set range or expression*	381
Activities	*collection of activities*	
Property	NoSave	
GroupSet	*a reference to a set*	
GroupDefinition	*activity : expression*	
GroupTransition	*index domain : expression*	
Transition	*set of reference pair : expression*	
FirstActivity	*reference*	
LastActivity	*reference*	
ComesBefore	*set of reference pairs*	
Precedes	*set of reference pairs*	
Unit	*unit-valued expression*	45, 211
LevelRange	*numeric range*	
InitialLevel	*reference*	
LevelChange	*per activity : expression*	
BeginChange	*per activity : expression*	
EndChange	*per activity : expression*	
Text	*string*	19
Comment	*comment string*	19

Table 22.4: Resource attributes

A resource is defined using the attributes presented in Table 22.4.

The Usage attribute

A resource can be used in two ways: Parallel, and Sequential, of which precisely one must be selected. The resource usage is then as follows

- **Sequential:** Defines the resource to be used sequentially. Such a resource is also known as a unary or disjunctive resource. A sequential resource has the additional attributes Transition, FirstActivity, LastActivity, ComesBefore, and Precedes, see Subsection 22.2.2.1.
- **Parallel:** Defines the resource to be used in parallel. Such a resource is also known as a cumulative resource. A parallel resource has the ad-

ditional attributes LevelRange, InitalLevel, LevelChange, BeginChange, and EndChange, see Subsection 22.2.2.2.

The Usage attribute is mandatory; either Sequential or Parallel must be selected.

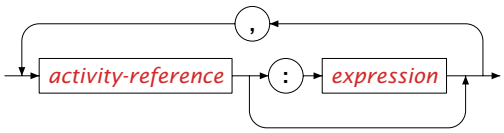
The ScheduleDomain attribute

The resource is affected by activities during the periods set in its schedule domain. This is an expression resulting in a one-dimensional set, or a set-valued range. AIMMS verifies that the schedule domain of the resource matches the schedule domain of all activities it is affected by. Here, two sets match if they have a common super set.
When the intersection of the schedule domain of a resource and the schedule domain of a non-optional activity are empty, the result is an infeasible schedule.

The Activities attribute

The Activities attribute details the activities affecting the resource. This adheres to the syntax:

activity-selection :



as illustrated in the example below:

```
Resource myMachine {
    ScheduleDomain  :  H;
    Usage           :  ...  ! sequential or parallel;
    Activities      : {
        maintenance,  ! Maintenance is scheduled between actual jobs.
        simpleJob(i), ! Every simple job can be done on this machine.
        specialJob(j) : jobpos(j) ! Only selected special jobs are allowed.
    }
}
```

In this example, the activities maintenance and simpleJob can affect the resource myMachine. However, the activity specialJob(j) can only affect the resource when jobpos(j) is non-zero. Only the detailed activities can be used in the attributes that follow. The Activities attribute is mandatory.

The Property attribute

A resource can have the properties: NoSave and TransitionOnlyNext.

- When the property NoSave is set, this indicates that the resource data will not be saved in cases.
- The property TransitionOnlyNext is relevant to the attributes Transition and GroupTransition of sequential resources only, and is discussed after the GroupTransition attribute below.

The attribute Property is not mandatory.

22.2.2.1 Sequential resources

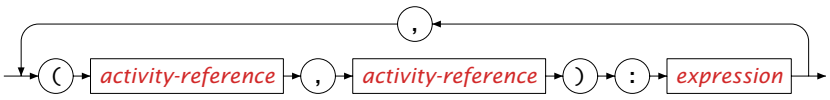
Sequential resources

Sequential resources are used to schedule activities that are not allowed to overlap. Those workers and machines that can only handle one activity at a time are typical examples. A sequential resource has only one suffix, namely `.ActivityLevel`. A sequential resource is active when it is servicing an activity, and then its `.ActivityLevel` is `1`. When a sequential resource is not active, or idle, its `.ActivityLevel` is `0`. The attributes particular to sequential resources are discussed below. The `.ActivityLevel` suffix cannot be used in constraint definitions.

The `Transition` attribute

The `Transition` attribute is only available to sequential resources, and then only if the `GroupSet` attribute has not been specified. This attribute contains a matrix between activities `a` and `b`, specifying the minimal time between `a` and `b` if `a` is scheduled before `b`. One example of using this attribute is to model traveling times, when jobs are executed at different locations. Another example of using this attribute is to model cleaning times of a paint machine, when the cleaning time depends on the color used during the previous job. All entries of this matrix are assumed to be 0 when not specified. If the schedule domain is a calendar, the unit of measurement is the time unit of the schedule domain; otherwise the unit of measurement is unitless. This matrix can, but need not, be symmetric. In the constraint programming literature, this attribute is also called *sequence-dependent setup times* or *changeover times*. The syntax for this attribute is as follows:

activity-transition :



An example of a transition specification is:

```
Resource myMachine {
    ScheduleDomain  :  H;
    Usage           :  sequential;
    Activities      :  acts(a), maintenance;
    Transition      : {
        (acts(a1),acts(a2))    : travelTime(a1,a2),
        (maintenance,acts(a1)) : travelTime('home',a1),
        (acts(a1),maintenance) : travelTime(a1,'home')
    }
    Comment         : {
        "activities acts are executed on location/site; yet
        maintenance is executed at home.  Transitions are
        the travel times between locations."
    }
}
```

The `Transition` attribute is not mandatory.

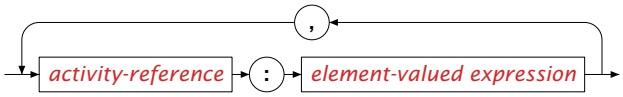
The GroupSet attribute

The GroupSet attribute is only available to sequential resources. The elements of this set name the groups into which the activities can be divided. This attribute is not mandatory.

The GroupDefinition attribute

The GroupDefinition attribute is only available when the GroupSet attribute has been specified. It contains a mapping of activities to group set elements. This mapping is essential for the GroupTransition attribute and for the intrinsic functions cp::GroupOfNext and cp::GroupOfPrevious. The syntax is as follows:

group-definition :



This attribute is mandatory when the GroupSet attribute has been specified.

The GroupTransition attribute

The GroupTransition attribute is used to specify the transition times/sequence dependent setup times between activities in a compressed manner. This attribute is only available when the GroupSet attribute has been specified. The syntax is:

activity-group-transition :

Consider an application where each city has to be visited by a car on multiple occasions, to bring goods being produced in one city to another city where they are consumed. The first product is consumed before the last product is produced:

```
Activity VisitCity {
    IndexDomain      :  (car,city,iter);
    ScheduleDomain   :  Timeline;
    Property         :  Optional;
}
Resource carEnRoute {
    Usage            :  sequential;
    IndexDomain      :  car;
    ScheduleDomain   :  TimeLine;
    Activities       :  VisitCity(car,city,iter);
    GroupSet         :  Cities;
    GroupDefinition  :  VisitCity(car,city,iter) : city;
    GroupTransition  :  (cityFrom,cityTo) : CityDistance(cityFrom,cityTo);
}
```

In this example, the group transition matrix is defined for each combination of cities, which is significantly smaller than an equivalent transition matrix defined for each possible combination of activities would have been. This

not only saves memory, but may also save a significant amount of solution time as some Constraint Programming solvers check whether the triangular inequality holds at the start of the solution process in order to determine the most effective reasoning available to that solver. The GroupTransition attribute is not mandatory.

Property TransitionOnlyNext

The attributes Transition and GroupTransition specify the minimal time between two activities a1 and a2 if a1 *comes before* a2. By specifying the property TransitionOnlyNext, these attributes are limited to specify only the minimal distances between two activities a1 and a2 if a1 *precedes* a2. An activity a1 precedes a2, if there is no other activity b scheduled between a1 and a2. In the example that follows, a, b, and c are all activities of length 1.

```
Resource seqres {
    Usage           :  sequential;
    ScheduleDomain  :  timeline;
    Activities      :  a, b, c;
    Property        :  TransitionOnlyNext;
    Transition      :  (a,b):1, (b,c):1, (a,c):7;
    Precedes        :  (a,b), (b,c);
}
```

Minimizing c.End, the solution is:

```
a.Begin := 0 ; a.End := 1 ;
b.Begin := 2 ; b.End := 3 ;
c.Begin := 4 ; c.End := 5 ;
```

By omitting the TransitionOnlyNext property, the minimal distance between a and c is taken into account, and the solution becomes:

```
a.Begin := 0 ; a.End := 1 ;
b.Begin := 2 ; b.End := 3 ;
c.Begin := 8 ; c.End := 9 ;
```

The Sequencing *attributes*

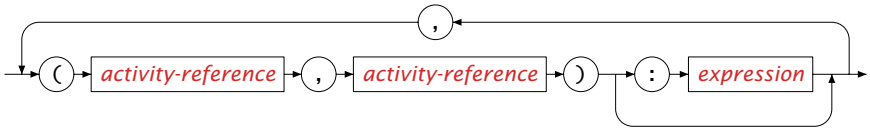
The attributes FirstActivity, LastActivity, ComesBefore, and Precedes are collectively called sequencing attributes. They are used to place restrictions on the sequence in which the activities are scheduled. These attributes are only available to sequential resources.

- **FirstActivity:** When specified, this has to be a reference to a single activity. When this activity is present, it will be the first activity in a feasible solution.
- **LastActivity:** When specified, this has to be a reference to a single activity. When this activity is present, it will be the last activity in a feasible solution.
- **ComesBefore:** This is a list of activity pairs (a,b). A pair (a,b) in this list indicates that activity a comes before activity b in a feasible solution. There may be another activity c that is scheduled between a and b in a feasible solution. This constraint is only enforced when both a and b are present.

- **Precedes:** This is a list of activity pairs (a,b). A pair (a,b) in this list indicates that activity a precedes activity b in a feasible solution. There can be no other activity c scheduled between a and b in a feasible solution, but a gap between a and b is allowed. This constraint is only enforced when both a and b are present.

The syntax of the attributes FirstActivity and LastActivity is simply a reference to a single activity and so the syntax diagram is omitted here. The syntax diagram for the attributes ComesBefore and Precedes is more interesting:

activity-sequence :



If, following the above syntax diagram, an expression is omitted, it is taken to be 1. An example illustrating all the sequencing attributes is presented below:

```
Resource myMachine {
    ScheduleDomain  :  H;
    Usage           :  sequential;
    Activities      :  setup(a), finish(a);
    FirstActivity   :  setup('warmingUp');
    LastActivity    :  finish('Cleaning');
    ComesBefore     :  (setup(a1),setup(a2)) : taskbefore(a1,a2);
    Precedes        :  (setup(a),finish(a));
}
```

None of the sequencing attributes are mandatory.

22.2.2.2 Parallel resources

Parallel resources model and limit the resource consumption and resource production of activities that take place in parallel. Examples of parallel resources could be monetary budget and truck load.

.ActivityLevel suffix

A parallel resource has only one suffix, namely .ActivityLevel. This suffix is only affected by scheduled activities. The limits on the .ActivityLevel suffix, its initialization, and how it is affected by executed activities is discussed below in the parallel resource specific attributes.

The LevelRange attribute

The LevelRange attribute states the range for the activity level of a parallel resource. The maximum value represents the capacity of the resource. It cannot be specified per element in the schedule domain of the resource. The syntax of this attribute is similar to the syntax of the Range attribute for bounded integer variables.

```
Resource myMachine {
    IndexDomain     :  m;
    ScheduleDomain  :  h;
```

```
    Usage           :  parallel;
    Activities      :  act(a);
    LevelRange      :  {
        {minlev(m) .. maxlev(m)}
    }
}
```

The LevelRange attribute is only applicable for parallel resources, and for such a resource it is mandatory.

The InitialLevel attribute

The InitialLevel attribute defines the initial value of the .ActivityLevel suffix. if it is not specified, the .ActivityLevel suffix is initialized to 0. The InitialLevel attribute is not mandatory.

```
Resource AvailableBudget {
    ScheduleDomain  :  Hor;
    Usage           :  parallel;
    Activities      :  act(a);
    LevelRange      :  {0 .. 10000};
    InitialLevel    :  5000;
    Comment         :  "we have a starting budget of 5000";
}
```

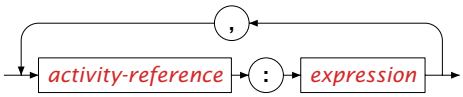
The .ActivityLevel modification attributes

The attributes LevelChange, BeginChange, and EndChange are collectively called *.ActivityLevel modification* attributes.

- An activity in the LevelChange attribute generates a pulse: at the .Begin of the activity the .ActivityLevel of the resource is increased by the indicated amount; at the .End of the activity that suffix is decreased by the same amount.
- An activity in the BeginChange attribute increases the .ActivityLevel of the resource at the .Begin of the activity by the indicated amount.
- An activity in the EndChange attribute increases the .ActivityLevel of the resource at the .End of the activity by the indicated amount.

Note that not only can the indicated amount be a positive or negative integer, it can also be an integer variable. The effect of an activity on the .ActivityLevel is illustrated in the Figure 22.1. The syntax of these attributes is as follows:

level-modification :



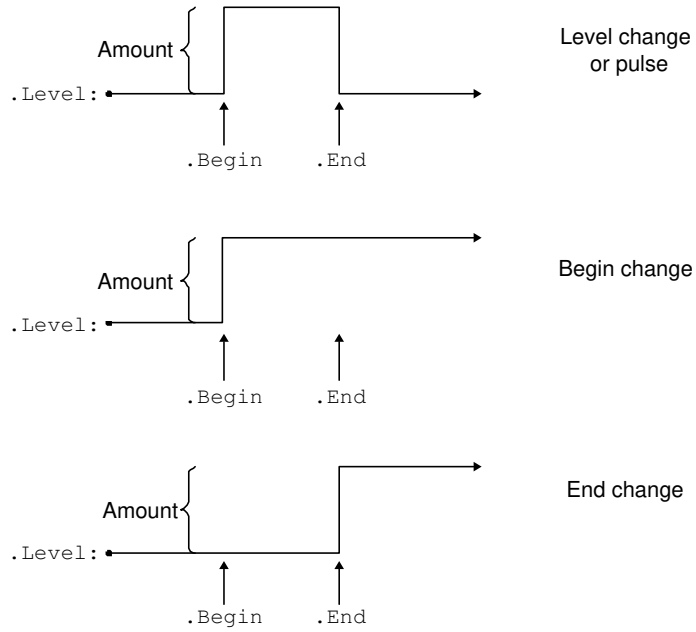


Figure 22.1: Changes to the suffix `.ActivityLevel` of a resource

The next example illustrates the use of the `.ActivityLevel` modification attributes:

```
Resource Budget {
    ScheduleDomain  :  Days;
    Usage           :  parallel;
    Activities      :  Act(i), Alt_Act(j), Deposit_Act(d);
    LevelRange      :  [0, 100];
    LevelChange     :  Alt_Act(i)    : -alt_act_budget(i);
    BeginChange     : {
        Deposit_Act(d): Deposit(d),
        Act(i)        : -ActCost(i)
    }
    EndChange       :  Act(i)        : Profit(i);
}
```

In this example, `Deposit_Act` can be modeled as an activity with a schedule domain containing only one element (an event), see Page 382. None of the `.ActivityLevel` modification attributes are mandatory, but when none of them is specified the resource is either infeasible or ineffective. When the `.ActivityLevel` is outside the range of a parallel resource, that resource is infeasible.

Activity level and schedule domain

The `.ActivityLevel` suffix is not affected by holes in the schedule domain of scheduled activities. Figure 22.2 illustrates the effect of activities `A` and `B` with a level change of `1` on the resource `cash`. The activity `A` has its `.Begin` set to Friday, its `.End` set to Tuesday and it is not scheduled in the weekend. The activity `B` is scheduled in the weekend.

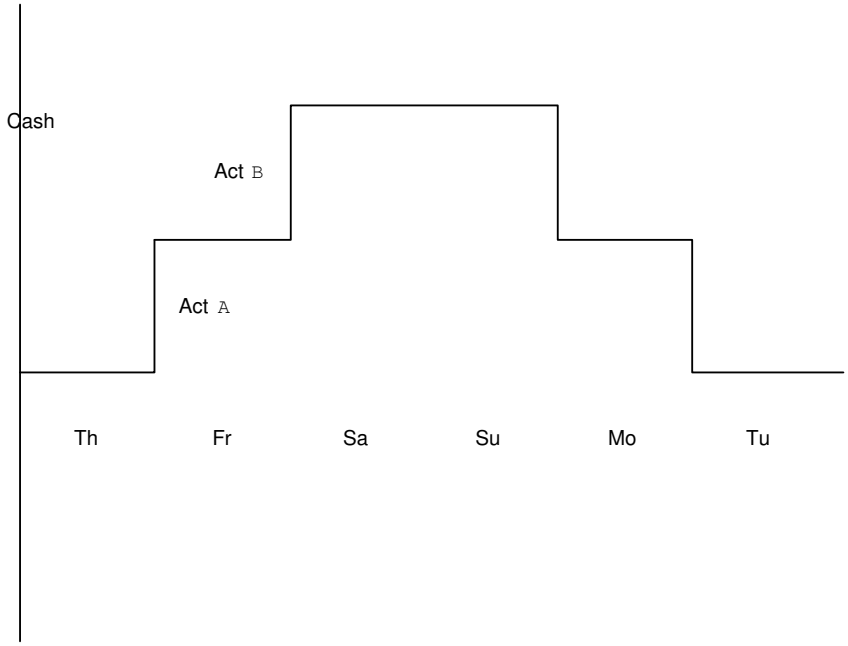


Figure 22.2: Two activities scheduled on a parallel resource

22.2.3 Functions on Activities and Scheduling constraints

Precedence constraints

The suffixes of an activity are variables, and they can be used in the formulation of constraints. Below there follows an example of a simple linear constraint which states that at least a pause of length `restTime` should be observed after activity `a` is completed before activity `b` can start.

```
a.End + restTime <= b.Begin
```

Precedence on optional activities

Consider again the inequality above, but now for optional activities `a` and `b`. When `a` is absent, the minimum value of `a.End` is meaningless but its minimum is 0 and `b` is present, this will enforce `b` to start after `restTime`. This may or may not be the intended effect of the constraint. Enforcing such constraints *only* when both activities `a` and `b` are present, the scheduling constraint `cp::EndAtStart(a,b,restTime)` can be used. This constraint is semantically equivalent to:

```
if a.Present and b.Present then
   a.End + restTime = b.Begin
endif
```

Here `restTime` is an integer valued expression that may involve variables. Note that the scheduling constraint can be exploited more effectively during the

solving process than the equivalent algebraic formulation. A list of available scheduling constraints for precedence relations is given in Table 22.5.

Precedence Relations	
	When activities a and b are present and for a non-negative integer delay d
cp::BeginBeforeBegin(a,b,d)	a.Begin + $d \leq b$.Begin
cp::BeginBeforeEnd(a,b,d)	a.Begin + $d \leq b$.End
cp::EndBeforeBegin(a,b,d)	a.End + $d \leq b$.Begin
cp::EndBeforeEnd(a,b,d)	a.End + $d \leq b$.End
cp::BeginAtBegin(a,b,d)	a.Begin + $d = b$.Begin
cp::BeginAtEnd(a,b,d)	a.Begin + $d = b$.End
cp::EndAtBegin(a,b,d)	a.End + $d = b$.Begin
cp::EndAtEnd(a,b,d)	a.End + $d = b$.End
Scheduling Constraints	**Interpretation**
cp::Span(g,i,a_i)	The activity g spans the activities a_i g.Begin = $\min_i a_i$.Begin$\wedge$ g.End = $\max_i a_i$.End
cp::Alternative(g,i,a_i)	Activity g is the single selected activity a_i $\exists j : g = a_j \wedge \forall k, j \neq k : a_k$.present = 0
cp::Synchronize(g,i,a_i)	If g is present, all present activities a_i are scheduled at the same time. g.present $\Rightarrow (\forall i : a_i$.present $\Rightarrow g = a_i)$

Table 22.5: Constraints for scheduling

Global scheduling constraints

In addition to these precedence constraints and the constraints that are defined by resources, AIMMS offers several other global constraints that are helpful in modeling scheduling problems. Table 22.5 presents the global scheduling constraints and functions available in AIMMS. These constraints are based on activities and can be used to represent hierarchical planning problems (cp::Span), to schedule activities over alternative resources (cp::Alternative), and to synchronize the execution of multiple activities (cp::Synchronize).

Functions on activities

There are several functions available that provide control over the value to be used for the suffixes of activities in the case of absence. In addition, there are functions available for relating adjacent activities on a resource. Table 22.6 lists the functions available that operate on activities. As an example, consider a model whereby the length of two adjacent jobs is limited:

```
Set Timeline {
    Index           :  tl;
}
Set Jobs {
    Index           :  j;
}
```

Limiting activity suffixes taking absence into account	
	a is the activity d the absence value
cp::ActivityBegin(a,d)	Return begin of activity
cp::ActivityEnd(a,d)	Return end of activity
cp::ActivityLength(a,d)	Return length of activity
cp::ActivitySize(a,d)	Return size of activity
Adjacent Activity	
	r is the resource s is the scheduled activity e is extreme value (when s is first or last) a is absent value (s is not scheduled)
cp::BeginOfNext(r,s,e,a)	Beginning of next activity
cp::BeginOfPrevious(r,s,e,a)	Beginning of previous activity
cp::EndOfNext(r,s,e,a)	End of next activity
cp::EndOfPrevious(r,s,e,a)	End of previous activity
cp::GroupOfNext(r,s,e,a)	Group of next activity, see also page 389
cp::GroupOfPrevious(r,s,e,a)	Group of previous activity
cp::LengthOfNext(r,s,e,a)	Length of next activity
cp::LengthOfPrevious(r,s,e,a)	Length of previous activity
cp::SizeOfNext(r,s,e,a)	Size of next activity
cp::SizeOfPrevious(r,s,e,a)	Size of previous activity

Table 22.6: Functions for scheduling

```
Parameter JobLen {
    IndexDomain     :  j;
}
Activity doJob {
    IndexDomain     :  j;
    ScheduleDomain  :  Timeline;
    Length          :  Joblen(j);
}
Resource aWorker {
    Usage           :  sequential;
    ScheduleDomain  :  Timeline;
    Activities      :  doJob(j);
}
Constraint LimitLengthTwoAdjacentJobs {
    IndexDomain     :  j;
    Definition      : {
        cp::ActivityLength(doJob(j),0) +
        cp::LengthOfNext(aWorker,doJob(j)) <= 8
    }
}
```

In the constraint `LimitLengthTwoAdjacentJobs` we take the length of job via the function `cp::ActivityLength` and the length of the next job for resource `aWorker` via the function `cp::LengthOfNext`. In the above constraint, the use of the func-

tion cp::ActivityLength is not essential because activity doJob is not optional. We can use the suffix notation instead and the constraint becomes:

```
Constraint LimitLengthTwoAdjacentJobs {
    IndexDomain     :  j;
    Definition      : {
        doJob(j).Length +
        cp::LengthOfNext(aWorker,doJob(j)) <= 8
    }
}
```

22.2.4 Problem schedule domain

The problem schedule domain of a mathematical program is a single named set containing all timeslots referenced in the activities, resources and timeslot valued element variables of that mathematical program. For the activities to be scheduled, we are usually interested in when they take place in real time; the mapping to real time is an ingredient of the solution. An exception might be if we are interested in the process of scheduling instead of its results. In that case, a contiguous subset of the Integers suffices. Contiguous subsets of Integers are supported by AIMMS, but not considered in the remainder of this subsection.

Real world representations

The problem schedule domain represents the period of time on which activities are to be scheduled. This portion of time is discretized into timeslots. Consider the following three use cases for a problem schedule domain.

1. The first use case is probably the most common one; the distance in time between two consecutive timeslots is constant. This distance is equal to a single unit of time. As an example, consider an application that constructs a maintenance scheme for playing fields. Two consecutive line painting activities should not be scheduled too close or too far from each other. Similarly for other consecutive activities of the same type such as garbage pickup. In this use case the distance in time between two consecutive timeslots is meaningful. A Calendar is practical for this use case.
2. The second use case is the one in which the distance in time between two consecutive timeslots is not constant. As an example, consider an application that constructs a sequence of practice meetings for teams with a limited number of playing fields available. The set of available dates is based on the team member availability, and the distance in time between two consecutive time slots may vary. An important restriction for this application is that two meetings with the same type of practice should be some number of meetings apart. In addition, we want to avoid, for each team, peaks and big holes in exercise dates by limiting the number of exercise dates skipped between two consecutive exercises.

3. The third use case is a combination of the other two use cases. The problem schedule domain is again one whereby the distance in time between two consecutive timeslots is constant. In addition, there are subsets of this problem schedule domain which apply to selected activities and resources. As an example, consider an application that schedules both maintenance activities and team practice sessions on a set of playing fields.

The above use cases are illustrated in AIMMS below.

Use case 1: constant distance

In the first use case as illustrated by the example below, the problem schedule domain is the calendar yearCal. Two consecutive timeslots in that calendar have the fixed distance of 1 day. The activity FieldMaintenance models that there are various types of maintenance activities to be scheduled for the various fields, and each type of maintenance may occur more than once. The two constraints in this example restrict the minimal and maximal distance between consecutive maintenance activities of the same type on the same field.

```
Calendar yearCal {
    Index           :  d;
    Unit            :  day;
    BeginDate       :  "2012-01-01";
    EndDate         :  "2012-12-31";
    TimeslotFormat  :  "%c%y-%sm-%sd";
}
Activity FieldMaintenance {
    IndexDomain     :  (pf, mt, occ);
    ScheduleDomain  :  yearCal;
    Property        :  Optional;
    Length          :  1[day];
    Comment         : {
        "Maintenance on
        playing field     pf
        maintenance type  mt
        occurrence       occ"
    }
}
Constraint maintenanceMinimalDistance {
    IndexDomain     :  (pf, mt, occ) | occ <> first( occurrences );
    Text            :  "at least 7 days apart.";
    Definition      : {
        cp::BeginBeforeBegin( FieldMaintenance(pf, mt, occ-1),
        FieldMaintenance(pf, mt, occ), 7 )
    }
}
Constraint maintenanceMaximalDistance {
    IndexDomain     :  (pf, mt, occ) | occ <> first( occurrences );
    Text            :  "at most 14 days apart.";
    Definition      : {
        cp::BeginBeforeBegin( FieldMaintenance(pf, mt, occ),
        FieldMaintenance(pf, mt, occ-1), -14 )
    }
}
```

Use case 2: varying distance

For the second use case, the same calendar `yearCal` is declared as in the first use case, in order to relate to real time. We are interested in only a selection of the dates available, and a subset `exerciseCal` is created from this calendar. In order to remove the fixed distance from the timeslots, the elements in `exerciseCal` are copied to `exerciseDates`.

```
Calendar yearCal {
    ...
}
Set  exerciseCal {
    SubsetOf        :  yearCal;
    Index           :  yc;
}
Set exerciseDates {
    Index           :  ed;
    Comment         :  "Constructed in ...";
}
```

A simple way of copying the elements with their names from `exerciseCal` to `exerciseDates` is in the code fragment below.

```
        Empty exerciseDates ;
        for yc do
            exerciseDates += StringToElement(exerciseDates,
             formatString("%e",yc), create:1 );
        endfor ;
```

Now that we have the set of exercise dates without a time unit associated, we can use it to declare exercise activities for each team and enforce a minimal distance between exercises of the same type as illustrated below. This minimal distance will now be enforced using a sequential resource and counting the number of times a particular exercise type occurred.

```
Set exerciseTypes {
    Index           :  et;
}
Activity gExerciseTeam {
    IndexDomain     :  (tm,occ);
    ScheduleDomain  :  exerciseDates;
    Length          :  1;
    Comment         : {
        "occ in Occurrencs, defining the number of times
        team tm has to exercise"
    }
}
Resource teamExercises {
    Usage           :  sequential;
    IndexDomain     :  tm;
    ScheduleDomain  :  exerciseDates;
    Activities      :  gExerciseTeam(tm, occ);
    Precedes        :  (gExerciseTeam(tm, occ1),gExerciseTeam(tm, occ2)):occ1=occ2-1;
    Comment         :  "Purpose: determine when a team may exercise";
}
Activity exerciseTeam {
    IndexDomain     :  (tm,et,occ);
    ScheduleDomain  :  exerciseDates;
    Property        :  optional;
    Length          :  1;
}
```

```
Constraint oneExerciseType {
    IndexDomain     :  (tm,occ);
    Definition      : {
        cp::Alternative(
        globalActivity  :  gExerciseTeam(tm, occ),
        activityBinding :  et,
        subActivity     :  ExerciseTeam(tm, et, occ))
    }
    Comment         :  "Purpose: select a single type of exercise";
}
Constraint doingAnExerciseTypeAtMostOnceOverOccurrences {
    IndexDomain     :  (tm,et,occ) | occ <> first( occurrences );
    Definition      : {
        sum( occ1 | occ1 <= occ and occ1 < occ + 2,
        ExerciseTeam(tm, et, occ).Present ) <= 1
    }
    Comment         : {
        "Purpose: the same type of exercise should be some
        exercises apart"
    }
}
Constraint avoidSmallHolesBetweenExercises {
    IndexDomain     :  (tm,occ) | occ <> first( occurrences );
    Text            :  "at least minHoleSize exercise dates apart.";
    Definition      : {
        cp::EndBeforeBegin( gExerciseTeam(tm, occ-1),
        gExerciseTeam(tm, occ), minHoleSize )
    }
}
Constraint avoidBigHolesBetweenExercises {
    IndexDomain    :  (tm,occ) | occ <> first( occurrences );
    Text            :  "at most maxHoleSize exercise dates apart.":
    Definition      : {
        cp::BeginBeforeEnd( gExerciseTeam(tm, occ),
        gExerciseTeam(tm, occ-1), -maxHoleSize )
    }
}
```

Use case 3: combination

There can be only one problem schedule domain per mathematical program; we use the one from the first use case as it encompasses the one from the second use case. Thus we need to adapt the schedule domains of selected activities and resources to the following:

```
Activity gExerciseTeam {
    IndexDomain     :  (tm,occ);
    ScheduleDomain  :  exerciseCal ! modified;
    Length          :  1[day]      ! modified;
}
Resource OneExercise {
    Usage           :  sequential;
    IndexDomain     :  tm;
    ScheduleDomain  :  exerciseCal ! modified;
    Activities      :  gExerciseTeam(tm, occ);
    Precedes        :  (gExerciseTeam(tm, occ1),gExerciseTeam(tm, occ2)):occ1=occ2-1;
    Comment         :  "Purpose: determine when a team may exercise";
}
Activity exerciseTeam {
    IndexDomain    :  (tm,et,occ);
    ScheduleDomain  :  exerciseCal ! modified;
```

```
    Property        :  optional;
    Length          :  1[day]      ! modified;
}
```

The constraints oneExerciseType and doingAnExerciseTypeAtMostOnceOverOccurrences can be left unchanged as they are not dependent on the distance between timeslots. The constraints avoidSmallHolesBetweenExercises and avoidBigHolesBetweenExercises is dependent on the distance between elements of exerciseDates and will now have to be remodeled. By using the fact that the .Size refers to the number of elements in the schedule domain of an activity between its .Begin and .End and that we can define an activity that encompasses a hole by spanning the previous exercise and the current one, the remodeling is done as follows:

```
Activity encompassHoleBetweenExercises {
    IndexDomain     :  (tm,occ)|occ <> first(occurrences);
    ScheduleDomain  :  exerciseCal;
}
Constraint defineEncompassHoleBetweenExercises {
    IndexDomain     :  (tm,occ)|occ <> first(Occurrences);
    Definition      : {
        cp::Span(
        globalActivity  :  encompassHoleBetweenExercises(tm, occ),
        activityBinding :  occ1 | occ1 = occ or occ1 = occ-1,
        subActivity     :  gExerciseTeam(tm, occ1))
    }
}
Constraint maxSizeEncompassingActivity {
    IndexDomain     :  (tm,occ)|occ <> first(Occurrences);
    Definition      : {
        encompassHoleBetweenExercises(tm,occ).size <=
        (maxHoleSize + 2)[day]
    }
}
Constraint minSizeEncompassingActivity {
    IndexDomain    :  (tm,occ)|occ <> first(Occurrences);
    Definition     : {
        encompassHoleBetweenExercises(tm,occ).size >=
        (minHoleSize + 2)[day]
    }
}
```

comparing use cases 2 and 3

Only when the distance in real time is not relevant to an application, the representation chosen for use case 2 has the following advantages over the representation chosen for use case 3:

- The limiting of the size of a hole is formulated more directly using the cp::EndBeforeBegin and cp::BeginBeforeEnd than one using an additional activity leading to a formulation that is easier to understand.
- We not only lose the power of propagation provided by the global constraints cp::EndBeforeBegin and cp::BeginBeforeEnd in use case 3, but we also introduce another activity, and thus additional variables, which increases the search space thereby decreasing the perfomance further.

- The set exerciseDates is smaller than the set yearCal. It is generally a good idea to keep the sets used as ranges for element variables small including the schedule domain to be considered.

This concludes the discussion on multiple use cases for problem schedule domains.

The attribute ScheduleDomain of a mathematical program

When an application contains activities, the attribute ScheduleDomain becomes available to a mathematical program declared in that application. Unlike the attribute with the same name for activities and resources, only a single named one-dimensional set can be filled in here. If this attribute is filled in, with, say Timeline, then AIMMS will ensure that each activity and resource has a schedule domain that is a subset of Timeline. In addition, for each element variable with a range say, selectedMoments, where Timeline and selectedMoments have the same root set, AIMMS will verify that selectedMoments is a subset of Timeline. In addition, the problem schedule domain, say Timeline, has to meet one of the following three requirements:

- Timeline is a true subset of the set Integers. In order to make references forward and backward in time unambiguous, it is required that Timeline is contiguous.
- Timeline is a subset of a calendar. Again, in order to make references forward and backward in time unambiguous, it is required that Timeline is contiguous.
- Timeline is not a subset of Integers nor a subset of a calendar. No additional requirements are placed on Timeline.

This attribute is not mandatory.

Deriving the problem schedule domain

If this attribute is not filled in, then AIMMS will derive the problem schedule domain by finding the smallest common superset of the schedule domains of the activities and resources of the mathematical program. If this set is not a calendar, but a subset thereof, AIMMS chooses to use the calendar instead. This is motivated by the assumption that the length between timeslots is usually relevant in scheduling applications. In scheduling applications in which the subset is more appropriate, copying the elements of a calendar provides an alternative, as illustrated above.

22.3 Modeling, solving and searching

Introduction

In this section we explain how constraint programming models are formulated and solved in AIMMS. We start by explaining how constraint programming formulations are fit into the paradigms of AIMMS like the free objective and units of measurement. We then explain the different mathematical programming types associated to constraint programming, and finally we discuss how a user can modify the search procedure in AIMMS.

The free objective

By design, the objective of any mathematical program in AIMMS is a *free* variable, even when it can be deduced that the objective function is always integer valued for a feasible solution. However, for an infeasible solution, AIMMS will assign the special value NA to the objective variable, in order to emphasize that a feasible solution is not available. Having a single variable for the objective function is convenient when communicating its value in the progress window and other places of AIMMS.

22.3.1 Constraint programming and units of measurement

Integer coefficients only

Constraint programming solvers require the coefficients used in constraints to be integer; fractional values or special values such as -inf, zero, na, undf, or inf are not allowed. In applications where the choice of base units is free, fractional values are easily avoided by choosing the base unit of quantities, and the units of variables and constraints such that all amounts to be considered are integer multiples of those base units, as detailed in Section 32.5.1. As an example, consider a simple constraint stating that the integer variable y is 0.9[m] away from the integer variable x. Both variables are multiples of the derived unit dm, i.e., 0.1[m]. The variables and constraints are declared as follows:

```
Variable x {
    Range       :  integer;
    Unit        :  dm;
}
Variable y {
    Range       :  integer;
    Unit        :  dm;
}
Constraint y_away_from_x {
    Unit        :  dm;
    Definition  :  y = abs( x - 0.9[m] );
}
```

Using the unit m as a base unit, this will lead to fractional values, as can be seen in the constraint listing:

```
y_away_from_x .. [ 1 | 1 | before ]

    y =
    abs((x-0.9))  ****

    name            lower           level           upper           scale
    x                   0               0 21474836470.000           0.100
    y                   0               0 21474836470.000           0.100
```

The coefficient for distance, 0.9[m], is in meters and the variables x and y have the same units inside the row. This row is scaled back to dm afterwards. As a result of all this scaling, the computations go from the domain of integer arithmetic into the domain of floating point arithmetic. For constraint programming, we need to avoid the domain of floating point arithmetic.

Adapted base unit

We will continue the above example, by adapting the base unit such that all amounts are integral multiples of that base unit; we select dm as a base unit. This will lead to the following row communicated to the solver:

```
y_away_from_x .. [ 1 | 1 | before ]

    y =
    abs((x-9))  ****

    name       lower      level      upper
    x              0          0 2147483647
    y              0          0 2147483647
```

Observe that scaling is not needed anymore. Using the scaling based on dm instead of m will keep all computations during the solution process in the domain of integer arithmetic.

Quantity based scaling

In the example of the previous paragraph, the base unit was adapted to the needs of constraint programming; namely to stay within the domain of integer arithmetic. For multi-purpose applications, freedom of choosing the base units of quantities according to the needs of a constraint programming problem is not always available. In order to stay within the domain of integer arithmetic, we can associate a convention with the mathematical program and filling in the per quantity attribute, see also Section 32.8. By filling in the per quantity attribute, AIMMS will generate the mathematical program where all coefficients are scaled with respect to the units specified in the per quantity attribute. Let us continue the example of the previous paragraph using m as the base unit and adding a convention to the mathematical program.

```
Quantity SI_Length {
    BaseUnit     :  m;
    Conversions  : {
        dm -> m : # -> # / 10,
        cm -> m : # -> # / 100
    }
    Comment      :  "Expresses the value of a distance.";
}
Parameter LengthGranul {
    InitialData  :  10;
}
Convention solveConv {
    PerQuantity  :  SI_Length : LengthGranul * cm;
}
MathematicalProgram myCP {
    Direction    :  minimize;
    Constraints  :  AllConstraints;
    Variables    :  AllVariables;
    Type         :  Automatic;
    Convention   :  solveConv;
}
```

Again, AIMMS will generate the constraint such that only integer arithmetic is needed. The constraint listing of that constraint is similar to the constraint listing presented in the paragraph *adapted base unit* above and not repeated.

Note also that with conventions, we can now use parameters to further control the scaling; if we want to change the model such that we can use multiples of 20[cm] instead of multiples of 10[cm], we only need to change the value of LengthGranul.

Calendar used for timeline

Scheduling applications in which the schedule domain is based on a calendar, the length of a timeslot is equal to the unit of the calendar, see Section 33.2. The time quantity is overridden, as if an entry in the per quantity attribute of the associated convention is given, selecting the calendar unit. Even if no convention was associated with the mathematical program. In short, for scheduling applications, AIMMS will scale time based data according to the length of a timeslot.

22.3.2 Solving a constraint program

Mathematical Programming types

AIMMS distinguishes two types of mathematical programs that are associated with constraint programming models: COP for constraint optimization problems, and CSP for constraint satisfaction problems. Both COP and CSP are exact in that COP provides a proven optimal solution while CSP provides a solution, or proves that none exist, if time permits.

Limiting solution time

Constraint programming problems are combinatorial problems and therefore may take a long time to solve, especially when trying to prove optimality. In order to avoid unexpectedly long solution times, you can limit the amount of time allocated to the solver for solving your problem as follows:

```
solve myCOP where time_limit := pMaxSolutionTime ;
! pMaxSolutionTime is in seconds.
```

Alternatively, when you are satisfied with the current objective, as presented in the progress window, and not want to wait on further improvements, you can interrupt the solution process by using the key-stroke *ctrl-shift-s*.

22.3.3 Search Heuristics

Search heuristics

During the solving process, constraint programming employs search heuristics that define the shape of the search tree, and the order in which the search tree nodes are visited. The shape of the search tree is typically defined by the order of the variables to branch on, and the corresponding value assignment. AIMMS allows the user to specify which variable and value selection heuristics are to be used. For example, to decide the next variable on which to branch, a commonly used search heuristic is to choose a non-fixed variable with the minimum domain size, and assign its minimum domain value.

Search phases

The first method offered by AIMMS to influence the search process is through using the Priority attributes of the variables. AIMMS will group together all variables that have the same priority value, and each block of variables will define a *search phase*. That is, the solver will first assign the variables in the block with the highest priority, then choose the next block, and so on. As discussed in Section 14.1.1, the highest priority is the one with the highest positive value. Defining search phases can be very useful. For example, when scheduling activities to various alternative resources, it is natural to first assign an activity to its resource before assigning its begin.

Variable and value selection

The variable and value selection heuristics offered by AIMMS are presented in Table 22.7. They can be accessed through the 'solver options' configuration window. As an example, we can define a 'constructive' scheduling heuristic that builds up the schedule from the begin of the schedule domain by using MinValue as the variable selection, and Min as the value selection. Indeed, this heuristic will attempt to greedily schedule the activities as early as possible. Note that both these variable and value heuristics apply to the entire search process. If no variable priorities are specified, the variable selection heuristic will consider all variables at a time. Otherwise, the variable selection heuristic is applied to each block individually.

Heuristic	**Interpretation**
Variable selection:	choose the non-fixed variable with:
Automatic	use the solver's default heuristic
MinSize	the smallest domain size
MaxSize	the largest domain size
MinValue	the smallest domain value
MaxValue	the largest domain value
Value selection:	assign:
Automatic	use the solver's default heuristic
Min	the smallest domain value
Max	the largest domain value
Random	a uniform-random domain value

Table 22.7: Search heuristics

Chapter 23

Mixed Complementarity Problems

This chapter

This chapter discusses the special identifier types and language constructs that AIMMS offers to allow you to formulate mixed complementarity problems. Although mixed complementarity problems do not involve optimization, they are specified through variables which are linked to constraints in terms of these variables, and thus fall into the common framework of a `Mathematical-Program`. AIMMS also supports nonlinear optimization problems with additional complementarity constraints (also known as MPCC or MPEC problems)

23.1 Complementarity problems

Complementarity problems

Complementarity relations arise in a variety of engineering and economics applications, most commonly to express an equilibrium of quantities such as forces or prices. One standard application in engineering arises in contact mechanics, where complementarity expresses the fact that friction occurs only when two bodies are in contact. Other applications are found in structural mechanics, structural design, traffic equilibrium and optimal control.

Economic models

Interest among economists in solving complementarity problems is due in part to increased use of computational general equilibrium models, where for instance complementarity is used to express Walras' Law, and in part to the equivalence of various games to complementarity problems.

Nonlinear optimization

Some generalizations of nonlinear programming, such as multi-level optimization—in which auxiliary objectives are to be minimized—may be reformulated as problems with complementarity conditions. Also, by formulating the Kuhn-Tucker conditions of a nonlinear optimization model one obtains a complementarity problem, which could be solved by a complementarity solver. In the latter case, however, one requires second-order derivative information of all constraints in the original optimization model.

Complementarity conditions

Complementarity problems are, in general, systems of nonlinear constraints where variables in the system are linked to constraints in the form of complementarity conditions. There are two forms of complementarity conditions, the classical complementarity condition, and its generalization, the mixed complementarity condition.

Classical complementarity conditions

The classical form of a complementarity condition involves a nonnegative variable x_i and an associated function $f_i(x)$. It requires that

$$\begin{aligned} x_i = 0 \quad &\text{and} \quad f_i(x) \geq 0, \text{ or} \\ x_i > 0 \quad &\text{and} \quad f_i(x) = 0. \end{aligned}$$

This condition states that of both inequalities, at least one must reach its bound. Alternatively, one can formulate this complementarity condition as $x_i \geq 0$, $f_i(x) \geq 0$ and $x_i \cdot f_i(x) = 0$.

Mixed complementarity condition

The mixed form of a complementarity condition involves a bounded variable $l_i \leq x_i \leq u_i$ with an associated function $f_i(x)$. It requires that

$$\begin{aligned} x_i = l_i \quad &\text{and} \quad f_i(x) \geq 0, \text{ or} \\ x_i = u_i \quad &\text{and} \quad f_i(x) \leq 0, \text{ or} \\ l_i < x_i < u_i \quad &\text{and} \quad f_i(x) = 0. \end{aligned}$$

This condition states that either x_i must reach one of its bounds, or the function $f_i(x)$ must be zero. A mixed complementarity condition can be split into two classical complementarity conditions (albeit by introducing auxiliary variables). The classical complementarity condition, on the other hand, is a special case of the mixed complementarity condition by choosing $l_i = 0$ and $u_i = \infty$.

Special cases

By choosing $l_i = -\infty$ and $u_i = \infty$, the mixed complementarity condition reduces to the special case

$$x_i \text{ is “free”} \quad \text{and} \quad f_i(x) = 0.$$

Another special case is obtained when $l_i = u_i$. The mixed complementarity condition then reduces to

$$x_i = l_i \quad \text{and} \quad f_i(x) \text{ is “free”}$$

Supported complementarity conditions in AIMMS

All complementarity conditions described above can be represented by associating a variable with a single constraint, which will form the basis for representing complementarity conditions in AIMMS. Consider the inequalities

$$l_{x_i} \leq x_i \leq u_{x_i} \quad \text{and} \quad l_{f_i} \leq f_i(x) \leq u_{f_i}$$

where $f_i(x)$ is a nonlinear expression, and *exactly* two of the constants l_{x_i}, u_{x_i}, l_{f_i} and u_{f_i} are finite. The six possible cases are enumerated in Table 23.1, and are discussed below.

Case	l_{x_i}	u_{x_i}	l_{f_i}	u_{f_i}
1	finite	finite	$-\infty$	∞
2	finite	∞	finite	∞
3	finite	∞	$-\infty$	finite
4	$-\infty$	finite	finite	∞
5	$-\infty$	finite	$-\infty$	finite
6	$-\infty$	∞	finite	finite

Table 23.1: Six allowed cases with exactly two finite bounds

Case 1

The case $l_{x_i} \le x_i \le u_{x_i}$ and $-\infty \le f_i(x) \le \infty$ corresponds to the mixed complementarity condition already discussed above:

$$\begin{aligned} x_i = l_{x_i} \quad &\text{and} \quad f_i(x) \ge 0, \text{ or} \\ x_i = u_{x_i} \quad &\text{and} \quad f_i(x) \le 0, \text{ or} \\ l_{x_i} < x_i < u_{x_i} \quad &\text{and} \quad f_i(x) = 0. \end{aligned}$$

Case 2

The case $l_{x_i} \le x_i \le \infty$ and $l_{f_i} \le f_i(x) \le \infty$ corresponds to the classical complementarity condition

$$\begin{aligned} \hat{x}_i = 0 \quad &\text{and} \quad \hat{f}_i(x) \ge 0, \text{ or} \\ \hat{x}_i > 0 \quad &\text{and} \quad \hat{f}_i(x) = 0. \end{aligned}$$

where $\hat{x}_i = x_i - l_{x_i}$ and $\hat{f}_i(x) = f_i(x) - l_{f_i}$.

Case 3

The case $l_{x_i} \le x_i \le \infty$ and $-\infty \le f_i(x) \le u_{f_i}$ corresponds to the classical complementarity condition

$$\begin{aligned} \hat{x}_i = 0 \quad &\text{and} \quad \hat{f}_i(x) \ge 0, \text{ or} \\ \hat{x}_i > 0 \quad &\text{and} \quad \hat{f}_i(x) = 0. \end{aligned}$$

where $\hat{x}_i = x_i - l_{x_i}$ and $\hat{f}_i(x) = u_{f_i} - f_i(x)$.

Case 4

The case $-\infty \le x_i \le u_{x_i}$ and $l_{f_i} \le f_i(x) \le \infty$ corresponds to the classical complementarity condition

$$\begin{aligned} \hat{x}_i = 0 \quad &\text{and} \quad \hat{f}_i(x) \ge 0, \text{ or} \\ \hat{x}_i > 0 \quad &\text{and} \quad \hat{f}_i(x) = 0. \end{aligned}$$

where $\hat{x}_i = u_{x_i} - x_i$ and $\hat{f}_i(x) = f_i(x) - l_{f_i}$.

Case 5

The case $-\infty \le x_i \le u_{x_i}$ and $-\infty \le f_i(x) \le u_{f_i}$ corresponds to the classical complementarity condition

$$\begin{aligned} \hat{x}_i = 0 \quad &\text{and} \quad \hat{f}_i(x) \ge 0, \text{ or} \\ \hat{x}_i > 0 \quad &\text{and} \quad \hat{f}_i(x) = 0. \end{aligned}$$

where $\hat{x}_i = u_{x_i} - x_i$ and $\hat{f}_i(x) = u_{f_i} - f_i(x)$.

Case 6: $l_{f_i} = u_{f_i}$

The case $-\infty \le x_i \le \infty$ and $l_{f_i} \le f_i(x) \le u_{f_i}$ with $l_{f_i} = u_{f_i}$ corresponds to the first special case of the mixed complementarity condition

$$x_i \text{ is “free”} \quad \text{and} \quad \hat{f}_i(x) = 0.$$

where $\hat{f}(x) = f_i(x) - l_{f_i}$.

Case 6: $l_{f_i} < u_{f_i}$

After the introduction of variables $x_i^+, x_i^- \ge 0$ and functions

$$\begin{aligned} f_i^x(x) &= x_i - x_i^+ - x_i^- \\ f_i^+(x) &= f_i(x) - l_{f_i} \\ f_i^-(x) &= u_{f_i} - f_i(x) \end{aligned}$$

the case $-\infty \le x_i \le \infty$ and $l_{f_i} \le f_i(x) \le u_{f_i}$ with $l_{f_i} < u_{f_i}$ corresponds to a system of three simultaneous complementarity conditions

$$x_i \text{ is “free”} \quad \text{and} \quad f_i^x(x) = 0$$

$$\begin{aligned} x_i^+ = 0 \quad &\text{and} \quad f_i^+(x) \ge 0, \text{ or} \\ x_i^+ > 0 \quad &\text{and} \quad f_i^+(x) = 0 \end{aligned}$$

$$\begin{aligned} x_i^- = 0 \quad &\text{and} \quad f_i^-(x) \ge 0, \text{ or} \\ x_i^- > 0 \quad &\text{and} \quad f_i^-(x) = 0. \end{aligned}$$

AIMMS support

AIMMS supports the variable-constraint couples with two finite bounds, as discussed above, through the special `ComplementaryVariable` data type. The declaration and attributes of this data type are discussed in the next section, while section 23.3 describes the declaration of mixed complementarity models through the common `MathematicalProgram` declaration.

Well-behaved systems

Like with nonlinear optimization models, not all mixed complementarity systems that can be formulated are well-behaved. For instance, a variable $x \ge 0$ with an associated constraint $1 - x \ge 0$, only admits the solutions 0 and 1, which would destroy the continuous character of complementarity problems. For systems of complementarity conditions that are not well-behaved, the solution process may produce no, or unexpected results.

23.2 ComplementaryVariable declaration and attributes

Complementarity variables

To support you in formulating a complementarity model, AIMMS provides a special type of variable, the ComplementaryVariable. The attributes of a complementarity variable allow you to declare an (indexed) class of variables in a complementarity model along with their associated constraints. The attributes of a ComplementaryVariable are listed in Table 23.2.

Automatic sanity checks

By construction, this new variable type automatically ensures that every variable in a complementarity model is associated with a single constraint. Also, when AIMMS detects that the total number of (finite) bounds on both the complementarity variable and its associated constraint is not equal to two (as required above), a compilation error will result. Thus, ComplementaryVariable will help to reduce the most common declaration errors for this type of model.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	42, 208, 208
Range	*range*	208
Unit	*unit-valued expression*	45, 211
Text	*string*	19, 45
Comment	*comment string*	19
Complement	*expression*	217
NonvarStatus	*reference*	212
Property	NoSave, Complement	

Table 23.2: ComplementaryVariable attributes

The IndexDomain attribute

Through the IndexDomain attribute of a complementarity variable you can specify domain of tuples for which you want AIMMS to generate a variable and its associated constraint. During generation, AIMMS will only generate a variable for all tuples that satisfy all domain restrictions that you have imposed on the domain.

The Range attribute

In the Range attribute you can specify the lower and upper bound of a complementarity variable, in a similar manner for ordinary Variables (see also Section 14.1). During generation, AIMMS will perform a runtime check, for every individual tuple in the index domain, whether the number of finite bounds specified here, plus the number of finite bounds in the constraint specified in the Complement attribute, exactly equals two.

The Complement attribute

The Complement attribute allows you to specify the constraint that must be associated with the complementarity variable at hand. With $f(x,\dots)$ a general nonlinear function, the following types of expressions are allowed

- $f(x,\dots) \geq a$ (variable must have a single-sided Range),
- $f(x,\dots) \leq a$ (variable must have a single-sided Range),
- $a \leq f(x,\dots) \leq b$ (variable must be free),
- $f(x,\dots) = a$ (variable must be free), or
- $f(x,\dots)$ (variable must be bounded).

In addition, the Complement attribute can refer to an existing Constraint in your model, which then should hold a definition as one of the cases above. The Complement attribute can also hold a scalar element parameter into the set AllConstraints, which offers the possibility to assign different constraints to the complementarity variable in sequential solves.

Constraint listing

In the constraint listing, the constraints associated with a complementarity variable will be listed with a generated name consisting of the name of the ComplementarityVariable with an additional suffix "_complement".

The NonvarStatus attribute

With the NonvarStatus attribute you can indicate for which tuples you want AIMMS to consider the complementarity variable as a parameter, i.e. with the lower and upper bound set equal to the level value prior to solving the model (see also Section 14.1.1). From the mixed complementarity condition it follows that the function in the corresponding constraint is then allowed to assume arbitrary values, whence there is no strict need to generate the variable and constraint for the solver.

Positive and negative values

The value of the NonvarStatus attribute must be an expression in some or all of the indices in the index list of the variable, allowing you to change the nonvariable status of individual elements or groups of elements at once. When the NonvarStatus assumes a positive value, AIMMS will not generate the variable and its associated constraint. For negative values, the variable and constraint will be generated, but reduces to the second special case of the mixed complementarity condition

$$\hat{x}_i = x_i - x_i^0 = 0 \quad \text{and} \quad f_i(x) \text{ is "free"},$$

i.e. the function in the constraint will be allowed to assume arbitrary values.

The Unit attribute

Providing a Unit for a complementarity variable will help you in a number of ways.

- AIMMS will help you to check the consistency of all the constraints and assignments in your model (including the expression in the Complement attribute), and
- AIMMS will use the units to scale the model that is sent to the solver.

Proper scaling of a model will generally result in a more accurate and robust solution process. You can find more information on the definition and use of units to scale mathematical programs in Chapter 32.

The Property attribute

Complementarity variables support the properties NoSave and Complement. With the property NoSave you indicate that you do not want to store data associated with this variable in a case. The Complement property indicates that you are interested in the level values of the constraint defined in the Complement attribute. When this property is set, AIMMS will make the level value of this constraint available through the .Complement suffix of the complementarity variable at hand.

Example

The declaration of the complementarity variable MembraneHeight expresses a complementarity condition for the height of a membrane in a rectangular (x, y)-grid, with a uniform external force acting on each cell in the grid.

```
ComplementaryVariable MembraneHeight {
    IndexDomain  : (x,y);
    Range        : [MembraneLowerBound(x,y), MembraneUpperBound(x,y)];
    Complement   : {
        4*MembraneHeight(x,y)
        - MembraneHeight(x+1,y) - MembraneHeight(x-1,y)
        - MembraneHeight(x,y+1) - MembraneHeight(x,y-1)
        - CellForce
    }
}
```

The complementarity condition expresses that either the membrane reaches one its given bounds (for instance, an obstacle placed in the way of the membrane), or the external force on the cell must be equal to the internal forces acting on the cell caused by differences in height with neighboring cells.

23.3 Declaration of mixed complementarity models

Mixed complementarity models

To define a pure mixed complementarity model, you must declare a MathematicalProgram (see also Section 15.1) and specify mcp as the Type attribute of the MathematicalProgram. In the Variables attribute you can specify a subset of the set of all ComplementarityVariables to be included in the mixed complementarity model at hand. Based on this specification, AIMMS will automatically generate all constraints associated with these complementarity variables, resulting in a square system.

Additional variables and constraints

In addition, AIMMS allows you to add ordinary variables to the Variables attribute, and to specify additional constraints in the Constraints attribute of the MathematicalProgram that must be satisfied as well. If the solver used to solve the mixed complementarity model requires a square system, AIMMS will automatically add auxiliary constraints or variables to the generated system,

and provide the linkages with the ordinary variables and constraints you have added to the system.

No optimization

For a mixed complementarity problem you should not specify the Objective and Direction attributes, as a complementarity solver will only compute a feasible solution that satisfies all the complementarity conditions specified. If these attributes are not empty, AIMMS will produce a runtime error when you apply the SOLVE statement the corresponding MathematicalProgram (see also Section 15.3).

Example

A mixed complementarity model containing the declaration of the complementarity variable MembraneHeight declared in the previous section, is defined by the following declaration.

```
MathematicalProgram Membrane {
    Variables   : AllVariables;
    Type        : mcp;
}
```

As usual, you can solve the Membrane through the statement

```
solve Membrane;
```

which will generate the mixed complementarity model and invoke a suitable solver for mcp problem type.

23.4 Declaration of MPCC models

MPCC models

Through the KNITRO solver, AIMMS also supports mathematical programs with complementarity constraints (MPCC models). MPCC models are also more commonly denoted by other modeling languages as MPEC models, which form a more general, and much more difficult, class of optimization problems. A MPCC model is an ordinary NLP model with additional complementarity constraints that have to be satisfied.

Declaring MPCC models

To define a MPCC model, you must declare a MathematicalProgram (see also Section 15.1) and specify mpcc as the Type attribute of the MathematicalProgram. The variable set of a MathematicalProgram of a MPCC model can contain ordinary variables as well as complementarity variables. Contrary to pure mixed complementary models, a MPCC model has an objective function.

Solving MPCC models

To solve MPCC models in AIMMS, you need a license for the KNITRO solver. If you do not have a license for the KNITRO solver, AIMMS will return an error that it has no suitable solver available for the mpcc class, whenever you try to solve a MPCC model. The KNITRO solver can also be used for solving pure mixed complementarity problems, but is, in general, far less efficient in that case than dedicated mcp solvers.

Chapter 24

Node and Arc Declaration

This chapter

This chapter discusses the special identifier types and language constructs that AIMMS offers to allow you to formulate network optimization problems in terms of nodes and arcs. In addition, it is illustrated how you can formulate an optimization problem that consists of a network combined with ordinary variables and constraints.

24.1 Networks

Networks

There are several model-based applications which contain networks and flows. Typical examples are applications for the distribution of electricity, water, materials, etc. AIMMS offers two special constructs, `Arcs` and `Nodes`, to formulate flows and flow balances as an alternative to the usual algebraic constructs. Specialized algorithms exist for pure network problems.

Mixed formulations

It is possible to intermingle network constructs with ordinary variables and constraints. As a result, the choice between `Arcs` and `Variables` on the one hand, and `Nodes` and `Constraints` on the other, becomes a matter of convenience. For instance, in the formulation of a flow balance at a node in the network you can refer to flows along arcs as well as to variables that represent import from outside the network. Similarly, you can formulate an ordinary capacity constraint involving both network flows and ordinary variables.

Flow keywords

It is assumed here that you know the basics of network flow formulations. Following are three flow-related keywords which can be used to specify a network flow model:

- `NetInflow`—the total flow into a node minus the total flow out of that node,
- `NetOutflow`—the total flow out of a node minus the total flow into that node, and
- `FlowCost`—the cost function representing the total flow cost built up from individual cost components specified for each arc.

The first two are always used in the context of a node declaration, while the third may be used for the network model declaration.

24.2 Node declaration and attributes

Node attributes

Each node in a network has a number of associated incoming and outgoing flows. Unless stated otherwise, these flows should be in balance. Based on the flows specified in the model, AIMMS will automatically generate a balancing constraint for every node. The possible attributes of a Node declaration are given in Table 24.1.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	42, 208, 216
Unit	*unit-valued expression*	45, 211
Text	*string*	19, 45
Comment	*comment string*	19
Definition	*expression*	217
Property	NoSave, Sos1, Sos2, Level, Bound, ShadowPrice, RightHandSideRange, ShadowPriceRange	45, 213, 218

Table 24.1: Node attributes

Nodes are like constraints

Nodes are a special kind of constraint. Therefore, the remarks in Section 14.2 that apply to the attributes of constraints are also valid for nodes. The only difference between constraints and nodes is that in the definition attribute of a node you can use one of the keywords NetInflow and NetOutflow.

NetInflow and NetOutflow

The keywords NetInflow and NetOutflow denote the net input or net output flow for the node. The expressions represented by NetInflow and NetOutflow are computed by AIMMS on the basis of all arcs that depart from and arrive at the declared node. Since these keywords are opposites, you should choose the keyword that makes most sense for a particular node.

Example

The following two Node declarations show natural applications of the keywords NetInflow and NetOutflow.

```
Node CustomerDemandNode {
    IndexDomain  : (j in Customers, p in Products);
    Definition   : {
        NetInflow >= ProductDemanded(j,p)
    }
}
```

```
Node DepotStockSupplyNode {
    IndexDomain  : (i in Depots, p in Products);
    Definition   : {
        NetOutflow <= StockAvailable(i,p) + ProductImport(i,p)
    }
}
```

The declaration of CustomerDemandNode(c,p) only involves network flows, while the flow balance of DepotStockSupplyNode(d,p) also uses a variable ProductImport(d,p).

24.3 Arc declaration and attributes

Arc attributes

Arcs are used to represent the possible flows between nodes in a network. From these flows, balancing constraints can be generated by AIMMS for every node in the network. The possible attributes of an arc are given in Table 24.2.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	42
Range	*range*	208
Default	*constant-expression*	44, 210
From	*node-reference*	
FromMultiplier	*expression*	
To	*node-reference*	
ToMultiplier	*expression*	
Cost	*expression*	
Unit	*unit-valued expression*	211
Priority	*expression*	211
NonvarStatus	*expression*	212
RelaxStatus	*expression*	213
Property	NoSave, *numeric-storage-property*, Inline, SemiContinuous, ReducedCost, ValueRange, CoefficientRange	34, 45, 213
Text	*string*	19, 45
Comment	*comment string*	19

Table 24.2: Arc attributes

Arcs are like variables

Arcs play the role of variables in a network problem, but have some extra attributes compared to ordinary variables, namely the From, To, FromMultiplier, ToMultiplier, and Cost attributes. Arcs do not have a Definition attribute because they are implicitly defined by the From and To attributes.

The From and To attributes

For each arc, the From attribute is used to specify the starting node, and the To attribute to specify the end node. The value of both attributes must be a reference to a declared node.

The Multiplier attributes

With the FromMultiplier and ToMultiplier attributes you can specify whether the flow along an arc has a gain or loss factor. Their value must be an expression defined over some or all of the indices of the index domain of the arc. The result of the expression must be positive. If you do not specify a Multiplier attribute, AIMMS assumes a default of one. Network problems with non unit-valued Multipliers are called *generalized networks*.

FromMultiplier and ToMultiplier

The FromMultiplier is the conversion factor of the flow at the source node, while the ToMultiplier is the conversion factor at the destination node. Having both multipliers offers you the freedom to specify the network in its most natural way.

The Cost attribute

You can use the Cost attribute to specify the cost associated with the transport of one unit of flow across the arc. Its value is used in the computation of the special variable FlowCost, which is the accumulated cost over all arcs. In the computation of the FlowCost variable the component of an arc is computed as the product of the unit cost and the level value of the flow.

Graphically illustrated

In the presence of FromMultiplier and ToMultipliers, the drawing in Figure 24.1 illustrates

- the level value of the flow,
- its associated cost component in the predefined FlowCost variable, and
- the flows as they enter into the flow balances at the source and destination nodes (denoted by SBF and DBF, respectively).

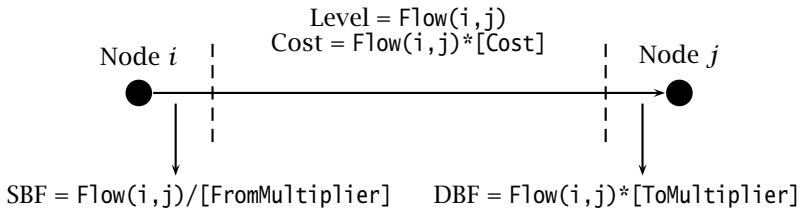


Figure 24.1: Flow levels and cost from node i to node j

Semi-continuous arcs

You can only use the SemiContinuous property for arcs if you use an LP solver to find the solution. If you use the pure network solver integrated in AIMMS, AIMMS will issue an error message.

Example

Using the declaration of nodes from the previous section, an example of a valid arc declaration is given by

```
Arc Transport {
    IndexDomain  :  (i,j,p) | Distance(i,j);
    Range        :  nonnegative;
    From         :  DepotStockSupplyNode(i,p);
    To           :  CustomerDemandNode(j,p);
    Cost         :  UnitTransportCost(i,j);
}
```

Note that this arc declaration declares flows between nodes i and j for multiple products p.

24.4 Declaration of network-based mathematical programs

The FlowCost variable

If your model contains arcs and nodes, the special variable FlowCost can be used in the definition of the objective of your mathematical program. During the model generation phase, AIMMS will generate an expression for this variable based on the associated unit cost for each of the arcs in your mathematical program.

Pure network models

AIMMS will mark your mathematical program as a pure network, if the following conditions are met:

- your mathematical program consists of arcs and nodes only,
- all arcs are continuous and do not have one of the SOS or the SemiContinuous properties,
- the value of the Objective attribute equals the variable FlowCost, and
- all Multiplier attributes assume the default value of one,

For pure network models you can specify network as its Type.

Network versus LP solver

If your mathematical program is a pure network model, AIMMS will pass the model to a special network solver. If your mathematical program is a generalized network or a mixed network-LP problem, AIMMS will generate the constraints associated with the nodes in your network as linear constraints and use an LP solver to solve the problem. AIMMS will also use an LP solver if you have specified its type to be lp. You may assert that your mathematical program is a pure network model by specifying network as its type.

Example

A pure network model containing the arc and node declarations of the previous sections, but without the additional term `ProductImport(d,p)` in the node `DepotStockSupplyNode(d,p)`, is defined by the following declaration.

```
MathematicalProgram ProductFlowDecisionModel {
    Objective   : FlowCost;
    Direction   : minimize;
    Constraints : AllConstraints;
    Variables   : AllVariables;
    Type        : network;
}
```

If the arc `Transport(i,j)` declared in the previous section is the only arc, then the variable `FlowCost` can be represented by the expression

```
    sum [(i,j,p), UnitTransportCost(i,j) * Transport(i,j,p)]
```

Note that the addition of the term `ProductImport(i,p)` in `DepotStockSupplyNode(i,p)` would result in a mixed network/linear program formulation, which requires an LP solver.

Part VI

Data Communication Components

Chapter 25

Data Initialization, Verification and Control

Aspects of the use of data

Data initialization, verification and control are important aspects of modeling applications. In general, verification of initialized data is required to check for input and consistency errors. When handling multiple data input sets, data control helps you to clean and maintain your internal data.

This chapter

This chapter describes how AIMMS implements data initialization, as well as the `Assert` mechanisms that you can use to verify the validity of the data of your model. In addition, this chapter describes the data control statements that you can use to maintain the data of your model in good order. All explicit forms of data communication with text files, cases and external databases are discussed in subsequent chapters.

25.1 Model initialization and termination

Separation of model and data

In general, it is a good strategy to separate the initialization of data from the specification of your model structure. This is particularly true for large models. The separation improves the clarity of the model text, but more importantly, it allows you to use the same model structure with various data sets.

Supplying initial data

There are several methods to input the initial data of the identifiers in your model. AIMMS allows you:

- to supply initial data for a particular identifier as part of its declaration,
- to read in data from various external data sources, such as text data files, AIMMS cases and databases, and
- to initialize data by means of algebraic statements.

Interactive initialization

In an interactive application the end-user often has to enter additional data or modify existing data before the core of the model can be executed. Thus, proper data initialization in most cases consists of more steps than just reading data from external sources. It is the responsibility of the modeler to make sure that an end-user is guided through all necessary initialization steps and that the sequence is completed before the model is executed.

The attribute InitialData

Both sets and parameters can have an InitialData attribute. You can use it to supply the initial data of a set or parameter, but only when the set or parameter does not have a definition as well. In general, the InitialData attribute is not recommended when different data sets are used. However, it can be useful for initializing those identifiers in your model that are likely to remain constant for all data sets. The contents of the InitialData attribute must be a *constant expression* (i.e. a constant, a constant enumerated set or a constant list expression) or a DataTable. The table format is explained in Section 28.2.

The MainInitialization and PostMainInitialization procedures

AIMMS will add the procedures MainInitialization and PostMainInitialization to a new project automatically. Initially these are empty, leaving the (optional) specification of their bodies to you. You can use these procedures to read in data from external sources and to specify AIMMS statements to compute your model's initial data in terms of other data. The latter step may even include solving a mathematical program. Both the MainInitialization and PostMainInitialization procedure are aimed at initializing your model. The distinction between the two becomes apparent in the presence of libraries in your model (cf. Section 35.5).

Library initialization

Each library can provide LibraryInitialization and PostLibraryInitialization procedures. The LibraryInitialization procedure is aimed at initializing the state of each library, *regardless of the state of other libraries*, such as sets and parameters that represent the internal state of the library, or, when the library uses an external DLL, initializing such a DLL. The PostLibraryInitialization procedures are executed after all LibraryInitialization procedures have been executed, and thus, can rely on the internal state of all other libraries already being initialized to perform tasks on its behalf.

Model initialization sequence

To initialize the data in your model, AIMMS performs the following actions directly after compiling the model:

- AIMMS fills the contents of any global set or parameter with the contents of its InitialData attribute,
- aimms executes the predefined procedures MainInitialization,
- AIMMS executes the predefined procedure LibraryInitialization for each library,
- AIMMS executes the predefined procedure PostMainInitialization, and
- finally AIMMS executes the predefined procedure PostLibraryInitialization for each library.

Thus, as a guideline, any model initialization that depends on (other) libraries should go into a PostLibraryInitialization or the PostMainInitialization procedure, to make sure that it can be executed successfully.

Library termination

Similarly to the situation of library initialization, each library can provide PreLibraryTermination and LibraryTermination procedures. The PreLibrary-Initialization procedures are executed before all LibraryTermination procedures have been executed, and are aimed at at library termination steps that may still need other libraries to be functioning. The LibraryTermination procedures are aimed at terminating the state of each library individually, for instance, to deinitialize any external DLLs the library may depend upon.

Model termination sequence

To terminate your model, AIMMS performs the following actions directly prior to closing the project:

- AIMMS executes the predefined procedure PreMainTermination,
- AIMMS executes the predefined procedure PreLibraryTermination for each library,
- aimms executes the predefined procedures MainTermination, and
- finally AIMMS executes the predefined procedure LibraryTermination for each library.

25.1.1 Reading data from external sources

The READ statement

You can use the READ statement to initialize data from the following external data sources:

- user-supplied text files containing constant lists and tables,
- AIMMS-generated binary case files, and
- external ODBC-compliant databases.

Reading from text data files

With the READ statement you can initialize selected model input data from text files containing explicit data assignments. Only DataTables and constant expressions (i.e. a constant, a constant enumerated set or a constant list expression) are allowed. Since the format of these AIMMS data assignments is simple, the corresponding files are easily generated by external programs or by using the AIMMS DISPLAY statement.

When useful

Reading from text files is especially useful when

- the data must come directly from your end-users, but is not contained in a formal database,
- the data is produced by external programs that are not linked or cannot be linked directly to AIMMS

Reading from binary case files

The READ statement can also initialize data from an AIMMS case file. You can instruct AIMMS to read either selected identifiers or all identifiers. The case file data is already in an appropriate format, and therefore provides a fast medium for data storage and retrieval inside your application.

When useful

Reading from case files is especially useful when

- you want to start up your AIMMS application in the same state as you left it when you last used it,
- you want to read from different data sources captured inside different cases making up your own internal database.

Reading from databases

A third (and powerful) application of the READ statement is the retrieval of data from any ODBC-compliant database. This form of data initialization gives you direct access to up-to-date corporate databases.

When useful

Reading from databases is especially useful when

- data is shared by several users or applications inside an organization,
- data integrity over time in a database plays a crucial role during the lifetime of your application.

Computing initial data

After reading initial data from internal and external sources, AIMMS allows you to compute other identifiers not yet initialized. This feature is very useful when the external data sources of your model supply only partial initial data. For instance, after reading in event data which represent tank actions (when and at what rate do charges and discharges take place), all stock levels at distinct model time instances can be computed.

25.2 Assertions

Data validity is important

In almost all modeling applications it is important to check the validity of input data prior to its use. For instance, in a transportation model it makes no sense if the total demand exceeds the total supply. In general, data consistency checks guard against unexplainable or even infeasible model results. As a result, these checks are essential to obtain customer acceptance of your application. In rigorous model-based applications it is not uncommon that the error consistency checks form a significant part of the total model text.

Assertion declaration and attributes

To provide you with a mechanism to implement data validity checks, AIMMS offers a special Assertion data type. With it, you can easily specify and verify logical conditions for all elements in a particular domain, and take appropriate action when you find an inconsistency. Assertions can be verified from within the model through the ASSERT statement, or automatically upon data changes

by the user from within the graphical user interface. The attributes of the Assertion type are given in Table 25.1.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	42, 208
Text	*string*	19, 45
Property	WarnOnly	
AssertLimit	*integer*	
Definition	*logical-expression*	34, 44
Action	*statements*	
Comment	*comment string*	19

Table 25.1: Assertion attributes

The Definition attribute

The Definition attribute of an Assertion contains the logical expression that must be satisfied by every element in the index domain. If the logical expression is not true for a particular element in the index domain, the specified action will be undertaken. Examples follow.

Examples

```
Assertion SupplyExceedsDemand {
    Text         :  Error: Total demand exceeds total supply;
    Definition   : {
        Sum( i in Cities, Supply(i) ) >=
        Sum( i in Cities, Demand(i) )
    }
}
Assertion CheckTransportData {
    IndexDomain  :  (i,j) | Distance(i,j);
    Text         :  Please supply proper transport data for transport (i,j);
    AssertLimit  :  3;
    Definition   : {
        UnitTransportCost(i,j) and
        MinShipment(i,j) <= MaxShipment(i,j)
    }
}
```

The assertion SupplyExceedsDemand is a global check. The assertion CheckTransportData(i,j) is verified for every pair of cities i and j for which Distance(i,j) assumes a nonzero value. AIMMS will terminate further verification when the assertion fails for the third time.

The Text attribute

The Text attribute of an Assertion is the text that is used as warning or error message when the assertion fails for an element in its domain. If the text contains indices from the assertions index domain, these are expanded to identify the elements for which the assertion failed. If you have overridden the default response by means of the Action attribute (see below), then the text attribute is ignored.

The Property attribute

The Property attribute of an assertion can only assume the value WarnOnly. With it you indicate that a failed assertion should only result in a warning being triggered, instead of an error. This attribute is also ignored if the Action is overridden.

The AssertLimit attribute

By default, AIMMS will verify an assertion for every element in its index domain, and call the (default) action for every element for which the assertion fails. With the AssertLimit attribute you can limit the number of verifications that are made. When the number of failed assertions reaches the AssertLimit, AIMMS will stop the verification of any further elemens in the index domain. By default, the AssertLimit is set to 1.

The Action attribute

The default response to a failing assertion is that either an error or a warning is raised, based on the Property setting. You can use the Action attribute if you want to specify a nondefault response to a failed assertion. Like the body of a procedure, the Action attribute can contain multiple statements which together implement the appropriate response. During the execution of the statements in the Action attribute, the indices occurring in the index domain of the assertion are bound to the currently offending element. This allows you to control the interaction with the end-user. For instance, you can request that all detected errors in the index domain are changed appropriately, or perhaps implement an auto-correct on invalid values.

The FailCount operator

If you raise an error or call the HALT statement during the execution of an Action attribute, the current model execution will terminate. When you use it in conjunction with the predefined FailCount operator, you can implement a more sophisticated version of the AssertLimit. The FailCount operator evaluates to the number of failures encountered during the current execution of the assertion. It cannot be referenced outside the context of an assertion.

Verifying assertions

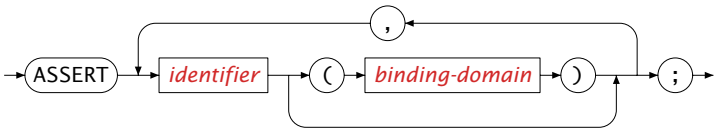
Assertions can be verified in two ways:

- by explicitly calling the ASSERT statement during the execution of your model, or
- automatically, from within the graphical user interface, when the end-user of your application changes input values in particular graphical objects.

The ASSERT statement

With the ASSERT statement you verify assertions at specific places during the execution of your model. Thus, you can use it, for instance, during the execution of the MainInitialization procedure, to verify the consistency of data that you have read from a database. Or, just prior to solving a mathematical program, to verify that all currently accrued data modifications do not result in data inconsistencies. The syntax of the ASSERT statement is simple.

assert-statement : *Syntax*



The following statement illustrates a basic use of the ASSERT statement. *Example*

```
assert SupplyExceedsDemand, CheckTransportData;
```

It will verify the assertion SupplyExceedsDemand, as well as the *complete* assertion CheckTransportData, i.e. checks are performed for every element (i,j) in its domain.

AIMMS allows you to explicitly supply a binding domain for an indexed assertion. By doing so, you can limit the assertion verification to the elements in that binding domain. This is useful when you know a priori that the data for only a small subset of the elements in a large index domain has changed. You can use such sliced verification, for instance, during the execution of a procedure that is called upon a single data change in a graphical object on a page. *Sliced verification*

Assume that CurrentCity takes the value of the city for which an end-user has made a specific data change in the graphical user interface. Then the following ASSERT statement will verify the assertion CheckTransportData for only this specific city. *Example*

```
assert CheckTransportData(CurrentCity,j),
       CheckTransportData(i,CurrentCity);
```

25.3 Data control

The contents of domain sets in your model may change through running procedures or performing other actions from within the graphical user interface. When elements are removed from sets, there may be data for domain elements that are no longer in the domain sets. In addition, data may exist for intermediate parameters, which is no longer used in the remainder of your model session. For these situations, AIMMS offers facilities to eliminate or activate data elements that fall outside their current domain of definition. This section provides you with housekeeping data control statements, which can be combined with ordinary assignments to keep your model data consistent and maintained. *Why data control?*

Facilities

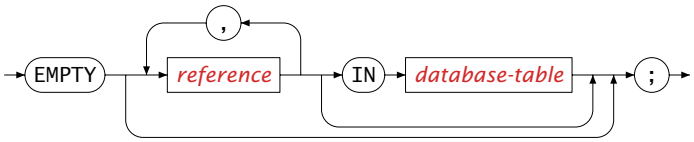
AIMMS offers the following data control tools:

- the EMPTY statement to remove the contents from all or a selected number of identifiers,
- the CLEANUP and CLEANDEPENDENTS statements to clean up all, or a selected number of identifiers,
- the REBUILD statement to manually instruct AIMMS to reclaim unused memory from the internal data structures used to the store the data of a selected number of identifiers,
- the procedure FindUsedElements to find all elements of a particular set that are in use in a given collection of indexed model identifiers, and
- the procedure RestoreInactiveElements to find and restore all inactive elements of a particular set for which inactive data exists in a given collection of indexed model identifiers.

The EMPTY statement

The EMPTY statement can be used to discard the complete contents of all or selected identifiers in your model. Its syntax follows.

empty-statement :



Empty AIMMS *identifiers*

The EMPTY operator operates on a list of references to AIMMS identifiers and takes the following actions.

- For parameters, variables (arcs) and constraints (nodes) AIMMS discards their values plus the contents of all their suffices.
- For sets, AIMMS will discard their contents plus the contents of all corresponding subsets. If a set is a domain set, AIMMS will remove the data from *all* parameters and variables that are defined over this set or any of its subsets.
- For slices of an identifier, AIMMS will discard all values associated with the slice.
- For sections in your model text, AIMMS will discard the contents of all sets, parameters and variables declared in this section.
- For a subset of the predefined set AllIdentifiers, AIMMS will discard the contents of all identifiers contained in this subset.

Use in databases

You can also use the EMPTY statement in conjunction with databases. With the EMPTY statement you can either empty single columns in a database table, or discard the contents of an entire table. This use is discussed in detail in Section 27.4. You should note, however, that applying the EMPTY statement to

a subset of AllIdentifiers does *not* apply to any database table contained in the subset to avoid inadvertent deletion of data.

Examples

The following statements illustrate the use of the EMPTY operator.

- Remove all data of the variable Transport.

```
empty Transport ;
```

- Remove all data in the set Cities, but also all data depending on Cities, like e.g. Transport.

```
empty Cities ;
```

- Remove all the data of the indicated slice of the variable Transport

```
empty Transport(DiscardedCity, j);
```

- Remove all data of all identifiers in the model tree node CityData.

```
empty CityData ;
```

Inactive data

When you remove some but not all elements from a domain set, AIMMS will not automatically discard the data associated with those elements for every identifier defined over the particular domain set. AIMMS will also not automatically discard data that does not satisfy the current domain restriction of a given identifier. Instead, it will consider such data as *inactive*. During the execution of your model, no reference will be made to inactive data, but such data may still be visible in the user interface. In addition, AIMMS will not directly reclaim the memory that is freed up when the cardinality of a multidimensional identifier in your model decreases.

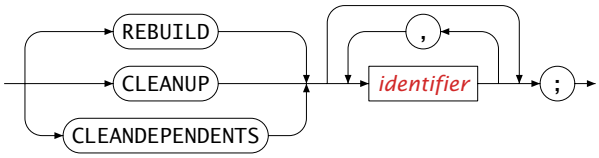
When useful

The facility to create inactive data in AIMMS allows you to temporarily remove elements from domain sets when this is required by your model. You can then restore the data after the relevant parts of the model have been executed.

Discard inactive data

If you want to discard inactive data that has been introduced in a particular data set, you can apply the CLEANUP statement to parameters and variables, or the CLEANDEPENDENTS statement to root sets in your model. Through the REBUILD statement you can instruct AIMMS to reclaim the unused memory associated with one or more identifiers in your model. The syntax follows.

cleanup-statement :



Rules

The following rules apply when you call the CLEANUP statement.

- When you apply the CLEANDEPENDENTS statement to a set, all inactive elements are discarded from the set itself and from all of its subsets. In addition, AIMMS will discard all inactive data throughout the model caused by the changes to the set.
- When you apply the CLEANUP statement to a parameter or variable, all inactive data associated with the identifier is removed. This includes inactive data that is caused by changes in domain and range sets, as well as data that has become inactive by changes in the domain condition of the identifier.
- When you apply the CLEANDEPENDENTS, CLEANUP, or REBUILD statement to a section, AIMMS will remove the inactive data of all sets, or parameters and variables declared in it, respectively.

After using the CLEANUP or CLEANDEPENDENTS statement for a particular identifier, all its associated inactive data is permanently lost.

Resorting root set elements

In addition to discarding inactive data from your model that is caused by the existence of inactive elements in a root set, the CLEANDEPENDENTS operator will also completely resort a root set and all data defined of it whenever possible and necessary. The following rules apply.

- Resorting will only take place if the current storage order of a root set differs from its current ordering principle.
- AIMMS will not resort sets for which explicit elements are used in the model formulation.

As a call to CLEANDEPENDENTS requires a complete rebuild of all identifiers defined over the root sets involved, the CLEANDEPENDENTS statement may take a relatively long time to complete. For a more detailed description of the precise manner in which root set elements and multidimensional data is stored in AIMMS refer to Section 13.2.7. This section also explains the benefits of resorting a root set.

Generated Mathematical Programs

The CLEANDEPENDENTS statement will also check whether any variable or constraint is affected; and if so will remove any generated mathematical program that is generated from such a variable or constraint.

Restricted usage in AIMMS GUI

You should not call the CLEANDEPENDENTS statement in procedures that have been linked to edit actions in graphical objects in an AIMMS end-user GUI via the **Procedures** tab of the object **Properties** dialog box. During these actions, AIMMS does not expect the element numbering to change.

Efficiency considerations

If you want to apply the CLEANDEPENDENTS statement to multiple sets, applying the operation to all sets in a single call of the CLEANDEPENDENTS statement will, in general, be more efficient than using a separate call for every single set. If an identifier depends on two or more of the sets to which you want to apply the CLEANDEPENDENTS operation, the data of such an identifier will only be traversed and/or rebuild once, rather than multiple times.

Examples

- The following CLEANDEPENDENTS statement will remove all data from your application that depends on the removed element 'Amsterdam', including, for instance, all previously assigned values to Transport departing from or arriving at 'Amsterdam'.

  ```
  Cities -= 'Amsterdam' ;

  cleandependents Cities ;
  ```

- The following CLEANUP statement will remove the data of the identifier Transport for all tuples that either lie outside the current contents of Cities, or do not satisfy the domain restriction.

  ```
  cleanup Transport;
  ```

- Consider a parameter A(i,j) where i is an index into a set S and j an index into a set T, then

  ```
  cleandependents S,T;
  ```

  will be more efficient than

  ```
  cleandependents S;
  cleandependents T;
  ```

  because the latter may require A(i,j) to be rebuilt twice.

Finding used elements

When you want to remove the elements in a set that are no longer used in your application, you first have to make sure which elements are currently in use. To find these elements easily, AIMMS provides the procedure FindUsedElements. It has the following three arguments:

- a set *SearchSet* for which you want to find the used elements,
- a subset *SearchIdentifiers* of the predefined set AllIdentifiers consisting of all identifiers that you want to be investigated, and
- a subset *UsedElements* of the set *SearchSet* containing the result of the search.

Upon execution, AIMMS will return that subset of *SearchSet* for which the elements are used in the combined data of the identifiers contained in *SearchIdentifiers*. When the identifiers *SearchSet* and UsedElements are contained in SearchIdentifiers they are ignored.

Example

The following call to FindUsedElements will find the elements of the set Cities that are used in the identifiers Supply, Demand, and Distance, and store the result in the set UsedCities.

```
SearchIdentifiers := DATA { Supply, Demand, Distance };

FindUsedElements( Cities, SearchIdentifiers, UsedCities );
```

If these cities are the only ones of interest, you can place them into the set Cities, and thereby overwrite its previous contents. After that you can cleanup your entire dataset by eliminating data dependent on cities other than the ones currently contained in the set Cities. This process is accomplished through the following two statements.

```
Cities := UsedCities;

cleandependents Cities;
```

Finding and restoring inactive elements

Inactive data in AIMMS results when elements are removed from (domain) sets. Such data will be inaccessible, unless the corresponding set elements are brought back into the set. When this is necessary, you can use the procedure RestoreInactiveElements provided by AIMMS. This procedure has the following three arguments:

- a set *SearchSet* for which you want to verify whether inactive data exists,
- a subset *SearchIdentifiers* of the predefined set AllIdentifiers consisting of those identifiers that you want to be investigated, and
- a subset *InactiveElements* of the set *SearchSet* containing the result of the search.

Upon execution AIMMS will find all elements for which inactive data exists in the identifiers in *SearchIdentifiers*. The elements found will not only be placed in the result set *InactiveElements*, but also be added to the search set. This latter extension of *SearchSet* implies that the corresponding inactive data is restored.

Example

The following call to RestoreInactiveElements will verify whether inactive data exists for the set Cities in AllIdentifiers.

```
RestoreInactiveElements( Cities, AllIdentifiers, InactiveCities );
```

After such a call the set InactiveCities could contain the element 'Amsterdam'. In this case, the set Cities has been extended with 'Amsterdam' as well. If you subsequently decide that cleaning up the set Cities is harmless, the following two statements will do the trick.

```
Cities -= InactiveCities;

cleandependents Cities;
```

Reclaiming memory

If the cardinality of a multidimensional identifier in your model decreases, AIMMS will not automatically reclaim the memory that is freed up because of the decreased amount of data to store. Instead, it will keep the memory available to store additional data that is associated with subsequent changes to the identifier. If the cardinality of an identifier decreases dramatically during a run the of a model, this may lead to a huge amount of memory getting stuck up with a single identifier in your model.

Memory fragmentation

In addition, if a model is running for a prolonged period of time, and an identifier has undergone huge amounts of structural changes during that time, the memory associated with that identifier may become heavily fragmented. In the long run, memory fragmentation may lead to decreased performance of your model. Rebuilding the internal data structures associated with such an identifier will resolve the fragmentation problem.

Automatic reclamation

Prior to solving a mathematical program, AIMMS will perform a quick check comparing the total amount of memory used by an identifier to the amount of unused memory associated with that identifier. By adding to and removing elements from identifiers, memory may become fragmented and the fraction of unused memory may grow. If the fraction of unused memory compared to the total amount of memory in use becomes too large, AIMMS will automatically rebuild such an identifier in order to reclaim the unused memory. AIMMS will also reclaim the memory of an identifier whenever it becomes empty during the run of a model.

the REBUILD statement

Through the REBUILD statement you can manually instruct AIMMS to rebuild the internal data structures associated with one or more identifiers. During the REBUILD statement AIMMS uses a more thorough check to verify whether a rebuild of an identifier is worthwhile prior to solving a mathematical program.

25.4 Working with the set AllIdentifiers

Working with AllIdentifiers

Throughout your model you can use the predefined set AllIdentifiers to construct and work with dynamic collections of identifiers in your model. Several operators in AIMMS support the use of a subset of AllIdentifiers instead of an explicit list of identifier names, while other operators support the use of an index into AllIdentifiers instead of a single explicit identifier name.

Constructing identifier sets

AIMMS offers a number of constructs that can help you to construct a meaningful subset of AllIdentifiers. They are:

- set algebra with other predefined identifier subsets, and
- dynamic selection based on model query functions.

Predefined identifier sets

When compiling your model, AIMMS automatically creates an identifier set for every section in your model. Each such set contains all the identifier names that are declared in the corresponding section. In addition, for every identifier type, AIMMS fills a predeclared set `All`*IdentifierType* (e.g. `AllParameters`, `AllSets`) with all the identifiers of that type. The complete list of identifier type related sets defined by AIMMS can be found in the AIMMS Function Reference. You can use both type of sets to perform set algebra to construct particular identifier subsets of interest to your model.

Example

If your model contains a section `Unit Model`, you can assign the collection of all parameters in that section to a subset `UnitModelParameters` of `AllIdentifiers` through the assignment

```
UnitModelParameters := Unit_Model * AllParameters;
```

Model query functions

Another method to construct meaningful subsets of `AllIdentifiers` consists of using the functions provided to query aspects of those identifiers. Selected examples are:

- the function `IdentifierDimension` returning the dimension of the identifier,
- the function `IdentifierType` returning the type of the identifier as an element of `AllIdentifierTypes`,
- the function `IdentifierText` returning the contents of the `TEXT` attribute, and
- the function `IdentifierUnit` returning the contents of the `UNIT` attribute.

These functions take as argument an element in the set `AllIdentifiers`.

Functions accepting identifier index

In addition to the functions lists above, the functions `Card` and `ActiveCard` also accept an index into the set `AllIdentifiers`. They will then return the cardinality of the identifier represented by the index, or the cardinality of the active elements of that identifier, respectively. You can also use these functions to dynamically construct a subset of `AllIdentifiers`.

Example

The set expression

```
{ IndexIdentifiers in UnitModelParameters |
       IdentifierDimension( IndexIdentifier ) = 3 }
```

refers to the collection of all 3-dimensional parameter in the section `Unit Model`.

Working with identifier sets

The following operators in AIMMS support identifier subsets to represent a collection of individual identifiers:

- the READ and WRITE operators,
- the EMPTY, CLEANUP, CLEANDEPENDENTS, and REBUILD operators.

If you are interested in the contents of an identifier subset, you can use the DISPLAY operator, which will just print the identifier names contained in the set, rather than the contents of the identifiers referred to in the identifier set as is the case for the WRITE statement.

Functions accepting identifier sets

In addition to the operators above, the following AIMMS functions also operate on subsets of AllIdentifiers:

- GenerateXML,
- CaseCompareIdentifier,
- CaseCreateDifferenceFile,
- IdentifierMemory,
- GMP::Solution::SendToModelSelection,
- VariableConstraints,
- ConstraintVariables,
- ScalarValue,
- SectionIdentifiers,
- AttributeToString,
- IdentifierAttributes.

See also Section "Model Query Functions" on page **??** of AIMMS the Function Reference.

Chapter 26

The READ and WRITE Statements

Dynamic data transfer

In order to help you separate the model description and its input and output data, AIMMS offers the READ and WRITE statements for dynamic data transfer between your modeling application and external data sources such as

- *text data files*, and
- *database tables* in external ODBC-compliant databases.

This chapter

This chapter first introduces the READ and WRITE statements in the form of an extended example. Subsequently, their semantics are presented in full detail including issues such as filtering, domain checking, and slicing.

26.1 A basic example

Getting started

The aim of this section is to give you an overview of the READ and WRITE statements through a short illustrative example. It shows how to read data from and write data to text files and database tables. It is based on the familiar transport problem with the following input data:

- the set Cities,
- the relation Routes from Cities to Cities,
- the parameters Supply(i) and Demand(i) for each city i, and
- the parameters Distance(i,j) and TransportCost(i,j) for each route between two cities i and j.

For the sake of simplicity, it is assumed that there is only a single output, the actual Transport(i,j) along each route.

Format of input data

The input data can be conveniently given in the form of tables. One for the identifiers defined over a single city like Supply and Demand, and the other for the identifiers defined over a tuple of cities like Distance and TransportCost. These tables can be provided in the form of text files as in Table 26.1 (format explained in Section 28.3). Alternatively, the data can be obtained from particular tables in a database. This example assumes the following database tables exist:

- CityData for the one-dimensional parameters, and

- RouteData for the two-dimensional parameters.

```
COMPOSITE TABLE
    Cities      Supply  Demand
!   ----------  ------  ------
    Amsterdam     50
    Rotterdam    100
    Antwerp       75      25
    Berlin                125
    Paris                  75
;
```

```
COMPOSITE TABLE
    i          j          Distance  TransportCost
!   ---------  ---------  --------  -------------
    Amsterdam  Rotterdam        85         1.00
    Amsterdam  Antwerp         170         2.50
    Amsterdam  Berlin          660        10.00
    Amsterdam  Paris           510         8.25
    Rotterdam  Antwerp         100         1.20
    Rotterdam  Berlin          700        10.00
    Rotterdam  Paris           440         7.50
    Antwerp    Berlin          725        11.00
    Antwerp    Paris           340         5.00
    Berlin     Paris          1050        17.50
;
```

Table 26.1: Example data set for the transport model

26.1.1 Simple data transfer

Simple data initialization

The simplest use of the READ statement is to initialize data from a fixed name text data file, or a database table. To read all the data from each source, the following groups of statements will suffice

```
read from file  "transport.inp" ;

read from table CityData;
read from table RouteData;
```

Such statements are typically found in the body of the predefined procedure MainInitialization.

Reading identifier selections

When a data source also contains data for identifiers that are of no interest to your particular application (but may be to others), AIMMS allows you to restrict the data transfer to a specific selection of identifiers in that data source. For instance, the following READ statement will only read the identifiers Distance and TransportCost, not changing the current contents of the AIMMS identifiers Supply and Demand.

```
read Distance, TransportCost from file "transport.inp" ;
```

Similar identifier selections are possible when reading from a database table.

Writing the solution

After your model has computed the optimal transport, you may want to write the solution Transport(i,j) to an text output file for future reference. You can do this by calling the WRITE statement, which has equivalent syntax to the READ statement. The transfer of Transport(i,j) to the file transport.out is accomplished by the following WRITE statement.

```
write Transport to file "transport.out" ;
```

If you omit an identifier selection, AIMMS will write all model data to the file. When writing to a database table, AIMMS can of course only transfer data for those identifiers that are known in the table that you are writing to.

File name need not be explicit

File data transfer is not restricted to files with a fixed name. To choose the name of the data file either during execution or from within the end-user interface, you have several options:

- replace the filename string in the READ and WRITE statements with a string-valued parameter holding the filename, or
- use a File identifier (for text files only).

26.1.2 Set initialization and domain checking

Domain restrictions

When you are reading the initial data of the transport model from an external data source several situations can occur:

- you just want to initialize the set Cities from the data source,
- the set Cities has already been initialized, and you want to retrieve the parametric data for existing cities only, or
- the set Cities has already been initialized, but you want to extend it on the basis of the data read from the external data source.

The READ statements

The following statements impose domain restrictions on the READ statement.

```
read Cities
     from file "transport.inp" ;

read Supply, Demand
     from file "transport.inp"
     filtering i ;

read Supply, Demand
     from file "transport.inp" ;
```

Initializing sets

The first READ statement is a straightforward initialization of the set Cities. By default, AIMMS reads in replace mode, which implies that any previous contents of the set Cities is overwritten.

The second READ statement assumes that the set Cities has already been initialized. From all entries of the identifiers Supply and Demand it will only read those which correspond to existing elements in the set Cities, and skip over the data from the remaining entries.

Domain checking

The third READ statement differs from the second in that the clause 'FILTERING i' has been omitted. As a result, AIMMS will not reject data that does not correspond to an existing label in the set Cities, but will read all available Supply and Demand data, and extend the set Cities accordingly.

Extending domain sets

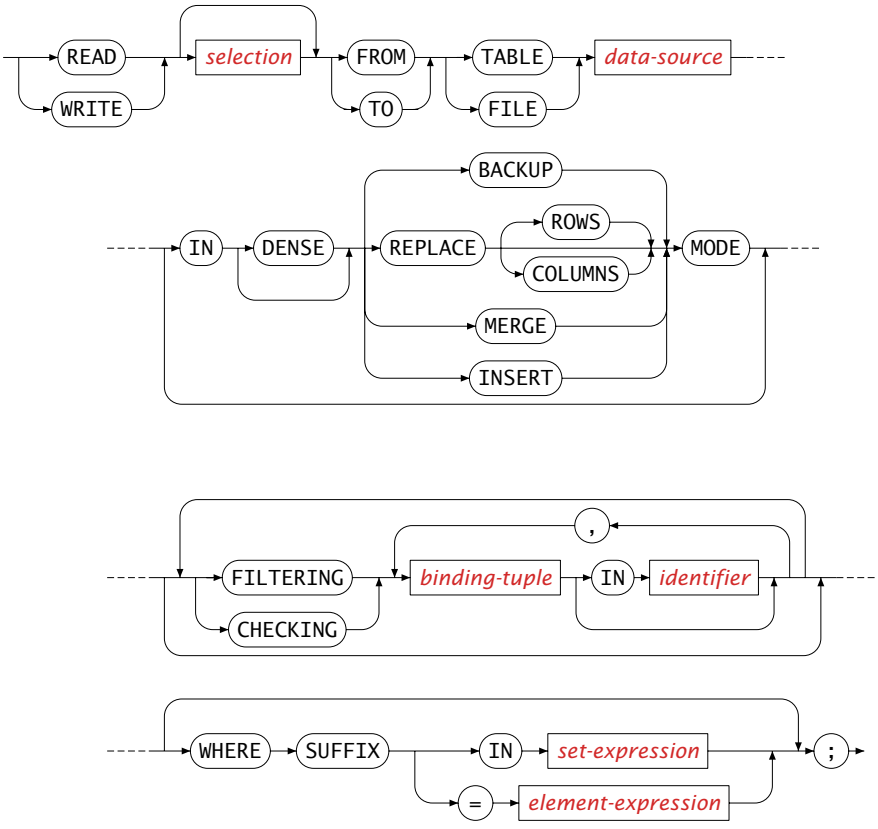
26.2 Syntax of the READ and WRITE statements

In READ and WRITE statement you can specify the data source type, what data will be transferred, and in what mode. The syntax of the statements reflect these aspects.

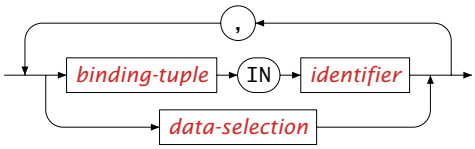
READ and WRITE statements

read-write-statement :

Syntax



selection :



Data sources

The data source of a READ or WRITE statement in AIMMS can be either

- a File represented by either
  - a File identifier,
  - a string constant, or
  - a scalar string reference,
- a TABLE represented by either
  - a DatabaseTable identifier,
  - an element parameter with a range that is a subset of the predeclared set AllDatabaseTables

Strings for file data sources refer either to an absolute path or to a relative path. All relative paths are taken relative to the project directory.

Examples

Assuming that UserSelectedFile is a File identifier, and UserFilename a string parameter, then the following statements illustrate the use of strings and File identifiers.

```
read from file "C:\Data\Transport\initial.dat" ;
read from file "data\initial.dat" ;
read from file UserFileName ;
read from file UserSelectedFile ;
```

Specifying a selection

The *selection* in a READ or WRITE statement determines which data you want to transfer from or to a text file, or database table. A selection is a list of references to sets, parameters, variables and constraints. During a WRITE statement, AIMMS accepts certain restrictions on each reference to restrict the amount of data written (as explained below). Note, however, that AIMMS does not accept all types of restrictions which are syntactically allowed by the syntax diagram of the READ and WRITE statements.

Default selection

If you do not specify a selection during a READ statement, AIMMS will transfer the data of all identifiers stored in the table or file that can be mapped onto identifiers in your model. If you do not specify a selection for a WRITE statement to a text file, all identifiers declared in your model will be written. When writing to a database table, AIMMS will write data for all columns in the table as long as they can be mapped onto AIMMS identifiers.

Filtering the selection

You can apply the following filtering qualifiers on READ and WRITE statements to restrict the data selection:

- the FILTERING or CHECKING clauses restrict the domain of all transferred data in both the READ and WRITE statements, and
- an arbitrary logical condition can be imposed on each individual parameter and variable in a WRITE statement.

FILTERING versus CHECKING

You can use both the FILTERING and CHECKING clause to restrict the tuples for which data is transferred between a data source and AIMMS. During a WRITE statement there is no difference in semantics, and you can use both clauses interchangeably. During a READ statement, however, the FILTERING clause will skip over all data outside of the filtering domain, whereas the CHECKING clause will issue a runtime error when the data source contains data outside of the filtering domain. This is useful feature for catching typing errors in text data files.

Examples

The following examples illustrate filtering and the use of logical conditions imposed on index domains.

```
read Distance(i,j) from table RouteTable
     filtering i in SourceCities, (i,j) in Routes;

write Transport( (i,j) | Sum(k, Transport(i,k)) > MinimumTransport )
      to table RouteTable ;
```

Advanced filtering on records

If you need more advanced filtering on the records in a database table, you can use the database to perform this for you. You can

- define *views* to create temporary tables when the filtering is based on a non-parameterized condition, or
- use *stored procedures* with arguments to create temporary tables when the filtering is based on a parameterized condition.

The resulting tables can then be read using a simple form of the READ statement.

Merge, replace or backup mode

AIMMS allows you to transfer data from and to a file or a database table in *merge* mode, *replace* mode or *insert* mode. If you have not selected a mode in either a READ or WRITE statement, AIMMS will transfer the data in replace mode by default, with one exception: when reading from a case difference file that was generated by CaseCreateDifferenceFile function with diffTypes argument equal to elementReplacement, elementAddition or elementMultiplication, the file is always read in merge mode, so that the diffTypes can be applied in a sensible way.
When you are writing data to a text data file, AIMMS also supports a *backup* mode. The *insert* mode can speed up writing to databases.

Reading in merge mode

When AIMMS reads data in merge mode, it will overwrite existing elements for all read identifiers, and add new elements as necessary. It is important to remember that in this mode, if there is no data read for some of the existing elements, they keep their current value.

Writing in merge mode

When AIMMS writes data in merge mode, the semantics is dependent on the type of the data source.

- If the data source is a text file, AIMMS will *append* the newly written data to the end of the file.
- If the data source is a database table, AIMMS will merge the new values into the existing values, creating new records as necessary.

Reading in replace mode

When AIMMS reads data in replace mode, it will empty the existing data of all identifiers in the identifier selection, and then read in the new data.

Writing in replace mode

When AIMMS writes data in replace mode, the semantics is again dependent on the type of the data source.

- If the data source is a text file, AIMMS will *overwrite the entire contents* of the file with the newly written data. Thus, if the file also contained data for identifiers that are not part of the current identifier selection, their data is lost by the `WRITE` statement.
- If the data source is a database table, AIMMS will either empty all columns in the table that are mapped onto identifiers in the identifier selection (default, `REPLACE COLUMNS` mode), or will remove all records in the table not written by this write statement (`REPLACE ROWS` mode). The `REPLACE COLUMNS` and `REPLACE ROWS` modes are discussed in more detail in Section 27.3).

Writing in insert mode

Writing in insert mode is only applicable when writing to databases. Essentially, what it does is writing the selected data to a database table using SQL *INSERT* statements. In other words, it expects that the selection of the data that you write to the table doesn't match any existing primary keys in the database table. If it does, AIMMS will raise an error message about duplicate keys being written. Functionally, the insert mode is equivalent to the replace rows mode, with the non-existing primary keys restriction. Especially when writing to database tables which already contain a lot of rows, the speed advantage of the insert mode becomes more visible.

Writing in backup mode

When you are transferring data to a text file, AIMMS supports writing in backup mode in addition to the merge and replace modes. The backup mode lets you write out files which can serve as a text backup to a (binary) AIMMS case file. When writing in backup mode, AIMMS

- skips all identifiers on the identifier list which possess a nonempty definition (and, consequently, cannot be read in from a datafile),
- skips all identifiers for which the property NoSave has been set, and
- writes the contents of all remaining identifiers in such an order that, upon reading the data from the file, all domain sets are read before any identifiers defined over such domain sets.

Backup mode is not supported during a READ statement, or when writing to a database.

Writing data in a dense mode

Writing in dense mode is only applicable when writing to databases. Data in AIMMS is stored for non-default values only, and, by default, AIMMS only writes these non-default values to a database. In order to write the default values as well to the database table at hand, you can add the *dense* keyword before most of the WRITE modes discussed above. This will cause AIMMS to write all possible values, including the defaults, for all tuple combinations considered in the WRITE statement. Care should be taken that writing in *dense* mode does not lead to an excessive amount of records being stored in the database. The mode combination *merge* and *dense* is not allowed, because it is ambiguous whether or not a non-default entry in the database should be overwritten by a default value of AIMMS.

Replacing sets

Whenever elements in a domain set have been removed by a READ statement in replace mode, AIMMS will *not* cleanup all identifiers defined over that domain. Instead, it will leave it up to you to use the CLEANUP statement to remove the inactive data that may have been created.

Domain filtering

For every READ and WRITE statement you can indicate whether or not you want domain filtering to take place during the data transfer. If you want domain filtering to be active, you must indicate the list of indices, or domain conditions to be filtered in either a FILTERING of CHECKING clause. In case of ambiguity which index position in a parameter you want to have filtered you must specify indices in the set or parameter reference.

Example

The following READ statements are not accepted because both Routes and Distance are defined over Cities $\times$ Cities, and it is unclear to which position the filtered index i refers.

```
read Routes   from table RouteTable filtering i ;
read Distance from table RouteTable filtering i ;
```

This ambiguity can be resolved by explicitly adding the relevant indices as follows.

```
read (i,j) in Routes from table RouteTable filtering i ;
read Distance(i,j)   from table RouteTable filtering i ;
```

Semantics of domain filtering

When you have activated domain filtering on an index or index tuple, AIMMS will limit the transfer of data dependent on further index restrictions.

- During a READ statement only the data elements for which the value of the given index (tuple) lies within the specified set are transfered. If no further index restriction has been specified, transfer will take place for all elements of the corresponding domain set.
- During a WRITE statement only those data elements are transferred for which the index (tuple) is contained in the AIMMS set given in the (optional) IN clause. If no set has been specified, and the data source is a database table, the transfer is restricted to only those tuples that are already present in the table. When the data source is a text file the latter type of domain filtering is not meaningful and therefore ignored by AIMMS.

READ example

In the following two READ statements the data transfer for elements associated with i and (i,j), respectively, is further restricted through the use of the sets SourceCities and Routes.

```
read Distance(i,j) from table RouteTable filtering i in SourceCities ;
read Distance(i,j) from table RouteTable filtering (i,j) in Routes ;
```

WRITE example

In the following two WRITE statements, the values of the variable Transport(i,j) are written to the database table RouteTable for those tuples that lie in the AIMMS set SelectedRoutes, or for which records in the table RouteTable are already present, respectively.

```
write Transport(i,j) to table RouteTable filtering (i,j) in SelectedRoutes ;
write Transport(i,j) to table RouteTable filtering (i,j) ;
```

The FILTERING clause in the latter WRITE statement would have been ignored by AIMMS when the data source was a text data file.

Writing selected suffices using the WHERE clause

Using the WHERE clause of the WRITE statement you can instruct AIMMS, for all identifiers in the identifier selection, to write the data of either a specified suffix or a set of suffices to file, rather than their level values. The WHERE clause can only be specified during a WRITE statement to a FILE, and the corresponding set or element expression must refer to a subset of, or element in, the predefined set AllSuffixNames.

Example

The following WRITE statement will write the values of the .Violation suffix of to the file ViolationsReport.txt for all variables in the project.

```
write AllVariables to file "ViolationsReport.txt" where suffix = 'Violation';
```

Chapter 27

Communicating With Databases

Communicating with databases

One of the most important capabilities of the READ and WRITE statements in AIMMS is its ability to transfer data with ODBC-compliant databases. Although there are similarities between the basic concepts of data storage in databases and those in AIMMS, they are sufficiently different to justify a separate chapter in this manual.

This chapter

This chapter deals with the intricacies of data transfer from and to databases. It first discusses the link between data in AIMMS and a table in a database. Then it explains the database-specific requirements regarding the READ and WRITE statements. Next comes a discussion on how to access stored procedures, followed by a description how to send SQL statements directly to a particular data source.

27.1 The DatabaseTable declaration

Database tables

You can make a database table known to AIMMS by means of a DatabaseTable declaration in your application. Inside this declaration you can specify the ODBC data source name of the database and the name of the database table from which you want to read, or to which you want to write. The list of attributes of a DatabaseTable is given in Table 27.1.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	42
DataSource	*string-expression*	
TableName	*string-expression*	
Owner	*string-expression*	
Property	ReadOnly	
Mapping	*mapping-list*	
Text	*string*	19
Comment	*comment string*	19
Convention	*convention*	534

Table 27.1: DatabaseTable attributes

The DataSource attribute

The mandatory DataSource attribute specifies the ODBC data source name of the database you want to link with. Its value must be a string or a string parameter. If you are unsure about the data source name by which a particular database is known, AIMMS will help you. While completing the declaration form of a database table, AIMMS will automatically let you choose from the available data sources on your system using the DataSource wizard. AIMMS supports the following data source types:

- ODBC file data sources (.dsn extension, only available on Windows), and
- ODBC user and system data sources (no extension), and
- ODBC connection string.

In addition, you can specify the name of an AIMMS string parameter, holding the name of any of the above data source types. If the data source you are looking for is not available in this list, you can set up a link to that database from within the wizard.

The TableName attribute

With the TableName attribute you must specify the name of the table or view within the data source to which the DatabaseTable is mapped. Once you have provided the DataSource attribute, the TableName wizard will let you select any table or view available in the specified data source.

Example

The following declaration illustrates the simplest possible DatabaseTable declaration.

```
DatabaseTable RouteData {
    DataSource  :  "Topological Data";
    TableName   :  "Route Definition";
}
```

It will connect to an ODBC user or system data source called "Topological Data", and in that data source search for a table named "Route Definition".

The Owner attribute

The Owner attribute is for advanced use only. By default, when connecting to a database server, you will have access to all tables and stored procedures which are visible to you. In case a table name appears more than once, but is owned by different users, by default a connection is made to the table instance owned by yourself. By specifying the Owner attribute you can gain access to the table instance owned by the indicated user.

The Property attribute

With the Property attribute of a DatabaseTable you can specify whether the declared table is ReadOnly. Specifying a database table as ReadOnly will prevent you from inadvertently modifying its content. If you do not provide this property, the database table will default to read-write permissions unless the server does not allow write access.

The Mapping attribute

By default, AIMMS tries to map the column names used in a database table onto the AIMMS identifiers of the same name. Such an implicit mapping is, of course, not always possible. When you link to an existing database that was not specifically designed for your AIMMS application, it is very likely that the column names do not correspond to the names of your AIMMS identifiers. Therefore, the Mapping attribute lets you override this default. The database columns explicitly mapped through the Mapping attribute are added to the set of implicit mappings constructed by AIMMS. The column names from the database table used in a mapping list must be quoted. If the implicit mapping is not desirable you can provide the property No Implicit Mapping.

Example

The following declarations demonstrate the use of mappings in a DatabaseTable declaration. This example assumes the set and parameter declarations of Section 26.1 plus the existence of the relation Routes given by

```
Set Routes {
    SubsetOf     : (Cities, Cities);
}
```

The following mapped database declaration will take care of the necessary column to identifier mapping.

```
DatabaseTable RouteData {
    DataSource   :  "Topological Data";
    TableName    :  "Route Definition";
    Mapping      : {
        "from"        --> i,                              ! name substitution
        "to"          --> j,
        "dist"        --> Distance(i,j),

        "fcost"       --> TransportCost(i,j,'fixed'),     ! slicing
        "vcost"       --> TransportCost(i,j,'variable'),

        ("from","to") --> Routes                          ! mapping to relation
    }
}
```

Name substitution

The first three lines of the Mapping attribute provide a simple name translation from a column in the database table to an AIMMS identifier. You can only use this type of mapping if the structural form of the database table (i.e. the primary key) coincides with the domain of the AIMMS identifier.

Mapping columns to slices

If the number of attributes in the primary key of a database table is lower than the dimension of the intended AIMMS identifier, you can also map a column name to a *slice* of an AIMMS identifier of the proper dimension, as shown in the fcost and vcost mapping. You can do this by replacing one or more of the indices in the identifier's index space with a reference to a fixed element.

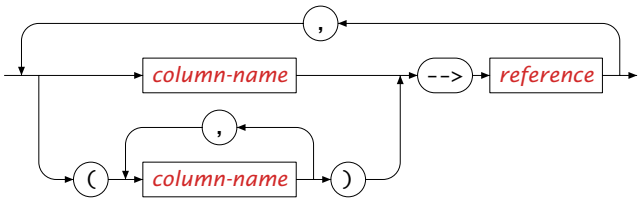
Mapping primary key to relation

As shown in the last line of the Mapping attribute, you can let the complete primary key in a database table correspond with a simple set, or with a relation (see Section 3.2.3) in AIMMS. This correspondence is specified by mapping the tuple of primary attributes of the table onto the AIMMS set itself, or onto an index into this set. The primary attributes in the tuple are mapped in a one-to-one fashion onto the indices in the relation.

Syntax

The syntax of the Mapping attribute is given by the following diagram.

mapping-list :



The Convention attribute

With the Convention attribute you can indicate to AIMMS that the external data is stored with the units provided in the specified convention. If the unit specified in the convention differs from the unit that AIMMS uses to store its data internally, the data is scaled at the time of transfer. For the declaration of Conventions you are referred to Section 32.8.

Date conversions

In addition, you can use Conventions to convert calendar data from the calendar slot format used within your model to the format expected by the database and vice versa. The use of Conventions for this purpose is discussed in full detail in Section 33.10. For non-calendar related date-time values you can use the predefined identifier OBDCDateTimeFormat to accomplish this (see Section 27.8).

27.2 Indexed database tables

Exogenous columns in primary key

While the Mapping attribute allows you to map data columns in a database table onto a slice of a higher-dimensional AIMMS identifier, a different type of slicing is required when the primary key of a database table contains *exogenous* columns that are of no interest to your application. Consider, for instance, the following situations.

- You are linking to a database table that contains data for a huge set of cities, but your model only deals with a single city that is not explicitly part of the model formulation. For your application the city column is exogenous.
- A table in a database contains several versions of a particular data set, where the version number is represented by an additional version column in the table. For your application the version column is exogenous.

Indexed DatabaseTables

In your AIMMS application you can deal with these situations by partitioning a single table inside the database into a set of *virtual* lesser-dimensional tables indexed by the exogenous column(s). You can do this by declaring the database table to have an IndexDomain corresponding to the sets that map onto the exogenous columns. In subsequent READ and WRITE statements you can then refer to a particular instance of a virtual table through a reference to the database table with an explicit set element or an element parameter.

Example

The following example assumes that the table "Route Definition" contains several versions of the data, each identified by the value of an additional column version. In the AIMMS model, this column is associated with a set TableVersions given by the following declaration.

```
Set TableVersions {
    Index      : v;
    Parameter  : LatestVersion;
}
```

The following declaration will provide a number of virtual tables indexed by v.

```
DatabaseTable RouteData {
    IndexDomain  :  v;
    DataSource   :  "Topological Data";
    TableName    :  "Route Definition";
    Mapping      : {
        "version"   --> v,
        "from"      --> i,
        "to"        --> j,
        "dist"      --> Distance(i,j),
        "cost"      --> TransportCost(i,j)
    }
}
```

Note that the index v in the index domain is mapped onto the column version in the table.

Data transfer with indexed tables

In order to obtain the set of TableVersions you can follow one of two strategies:

- you can obtain the set of the available versions from the table "Route Definition" itself by declaring another DatabaseTable in AIMMS

  ```
  DatabaseTable VersionTable {
      DataSource   :  "Topological Data";
      TableName    :  "Route Definition";
      Mapping      : {
          "version"   --> TableVersions
      }
  }
  ```

- or, you can obtain the versions from a separate table in a relational database declared similarly as above.

A typical sequence of actions for data transfer with indexed tables could then be the following.

- Read the set of all possible versions from VersionTable:

```
read TableVersions from table VersionTable ;
```

- Obtain the value of LatestVersion from within the language or the graphical user interface.
- Read the data accordingly:

```
read Distance, TransportCost from RouteData(LatestVersion) ;
```

27.3 Database table restrictions

Data transfer to single tables

The AIMMS READ and WRITE statements are intended to directly transfer data to and from a *single* text or case file, or a *single* table in a database. This is the simplest form of communication with a database. If you need more advanced control over the connection with a particular database, you can access stored procedures within the database using AIMMS. Such procedures can be implemented by the database designer to accomplish advanced tasks that go beyond the ordinary. The use of stored procedures is discussed in Section 27.5.

Automatic connection

When you are connecting to a table in a database through a READ or WRITE statement, you do not have to make a connection to the server explicitly. The database table declaration and the ODBC configuration files on your system provide sufficient information to allow AIMMS to make the connection automatically whenever needed. If you need to log on to the database, you will be prompted with a log on screen. On some systems it is possible to store log on information in the ODBC data source file.

Different data representation

There is a fundamental difference in the storage of data in AIMMS and the storage of data in a database table. Whereas AIMMS stores its data separately per identifier, a database table stores the data of several indexed identifiers in records all indexed by the same single index tuple. This difference implies that AIMMS has to impose some additional restrictions on data transfer with database tables that are not needed when reading from or writing to either AIMMS case files or text files.

Assumptions about database tables

In order to be able to define the semantics of the READ and WRITE statements to database tables in an unambiguous way, AIMMS makes a number of (reasonable) assumptions about the database tables in an external database. It is, however, not always possible for AIMMS to verify these assumptions, and unexpected effects may occur when they do not hold. The following assumptions about database tables are made.

- Every database table is in second normal form, i.e. every non-primary column in the table is functionally dependent on the primary key.

- Every primary column in a database table is mapped onto an index in an AIMMS domain set.
- Every non-primary column in a database table is mapped onto a (slice of an) AIMMS identifier, such that the specific index domain of this identifier precisely matches the primary key of the database table according to the existing index mapping.

Assumptions about identifier selections

AIMMS will not allow all identifier selections to be read from or written to database tables. An identifier selection is allowed when the following conditions hold for its components.

- All parameter and variable references must have the same domain after slicing. The resulting domain must correspond to the primary key of the database table.
- During a WRITE statement in REPLACE mode you can only write a simple set or relation mapped onto the primary key of a database table as long as there are no non-primary columns, or when the selection comprises all the columns of the table.
- AIMMS allows each domain set associated with a primary column in a table of any dimension to be read from that table.

Simply stated

The above rules can be summarized by stating that the database table can be transformed into an AIMMS composite table for the indexed identifiers in it.

Selections are sparse

Identifier selections in READ and WRITE statements form a one-to-one correspondence with a sparse set of records in the database table. During a READ statement the sparsity pattern is determined by all the records in the database table. During a WRITE statement the sparsity pattern is determined by all indexed identifiers in the selection. Records will be written for only those tuples for which at least one of the indexed identifiers or tuple references has a non-default value. Thus, the transferred data resulting from a WRITE statement is equivalent to the single composite table in AIMMS for all indexed identifiers in the selection.

Creation of records

Writing data to a database in either merge or replace mode may lead to the creation of new records in a database table. New records will be created when AIMMS writes a tuple for a key for which no record is available. If the table has non-primary attributes for which no data is written, AIMMS will leave these attributes empty when it creates new rows.

Removal of records: REPLACE COLUMNS/ROWS mode

AIMMS supports two replace modes for writing: REPLACE COLUMNS mode (default) and REPLACE ROWS mode.

- In REPLACE or REPLACE COLUMNS mode, AIMMS will only remove data from columns mapped onto identifiers in your model. Rows will only be re-

moved from the database table if *all* columns in the table are mapped onto identifiers in your model.
- In REPLACE ROWS mode, AIMMS will remove all rows whose primary key does not correspond to an index tuple being written during the WRITE statement. Columns that are not mapped onto identifiers in your model, either are assigned their default value specified in the database, or a NULL value otherwise. As a consequence, you should make sure that all non-nullable columns in the table are mapped onto identifiers in your model (or have a default value in the database) during REPLACE ROWS mode.

AIMMS will only remove records if the selection you are writing comprises all the columns in the database table, including the set mapped onto the primary key. In this way, AIMMS ensures that no data is lost in the table inadvertently.

Filtering on records

Using the DatabaseTable interface it is only possible to filter records using simple domain conditions formulated in a FILTERING clause. For huge database tables it may be desirable to use more advanced filtering techniques designed to restrict the number of records to be transferred. This can be done inside the database application itself in the following two ways.

- Create a *view* in the database that does the filtering for you, and then use the standard READ statement. This is the most straightforward approach, and is sufficient if the filter does not depend on AIMMS data.
- Create a *stored procedure* in the database that can be activated through a DatabaseProcedure in AIMMS. This allows you to filter records dependent on the value of some AIMMS identifiers that are used as arguments of the stored procedure (see Section 27.5).

27.4 Data removal

Data removal

The AIMMS database interface offers limited capabilities to manage the tables in a database. Such management is typically done through the use of stored procedures within a database. AIMMS, however, offers you the possibility to remove data from a database table by means of the EMPTY or TRUNCATE statement.

Empty database columns

The EMPTY statement can remove data from a database table in two manners.

- When you use a database table identifier in the identifier selection in an EMPTY statement, AIMMS will remove all data from that table.
- When you use a database table identifier behind the IN clause in an EMPTY statement, AIMMS will empty all columns in the corresponding database table which are mapped onto the AIMMS identifiers in the identifier selection of that EMPTY statement.

For more details on the EMPTY statement, refer to Sections 25.3.

Examples

The examples in this paragraph illustrate various uses of the EMPTY statement applied to database tables.

- The following statement removes all data from the table CityTable.

```
empty CityTable ;
```

- The following statement removes the data from the table CityTable that maps onto the AIMMS identifier Demand.

```
empty Demand in CityTable ;
```

- The following statement removes the data associated with the version OldVersion from the indexed table RouteTable(v). The data associated with other versions will remain intact.

```
empty RouteTable(OldVersion) ;
```

- The following statement removes the data from the table RouteTable(v) for all versions v in the set Versions.

```
empty RouteTable;
```

- The following statement removes the data from the table RouteTable(v) for all versions sv in the subset SelectedVersions of the set Versions.

```
empty RouteTable(sv);
```

- The following statement removes the data in the column mapped onto the AIMMS identifier Transport and associated with the version Latest-Version from the indexed table RouteTable(v).

```
empty Transport in RouteTable(LatestVersion) ;
```

Truncate database tables

Depending on the type of the underlying data source you are connecting to, you can use the TRUNCATE statement to empty the database table in a very fast way. The drawbacks of the TRUNCATE statement are that not all type of data sources support the
tt TRUNCATE statement and rollback support for the TRUNCATE operation is also depending on the type of data source you are connecting to. In case the underlying data source does not support truncating, depending on the setting of the option Warning_truncate_table a warning is issued and AIMMS will use the slower EMPTY statement to empty the table.

Example

The following statement removes all data from the table CityTable at once.

```
truncate table CityTable ;
```

27.5 Executing stored procedures and SQL queries

Sophisticated control

When transferring data from or to a database table, you may need more sophisticated control over the data link than offered by the standard DatabaseTable interface. AIMMS offers you this additional control by letting you have access to stored procedures as well as letting you execute SQL statements directly. The following two paragraphs provide some examples where such control may be useful.

Useful for data processing

Your application may require its data in a somewhat different form than is directly available in the database. In this case you may have to perform some pre-processing of the data in the database. Similarly, you may want to perform post-processing in the database after writing data to it. In such circumstances you may call a stored procedure to perform these tasks for you.

Useful for dynamic access

In some cases, the required data for your application may need to be the result of a parameterized query of the database, i.e. a database table whose contents is dependent on one or more parameters which are only known during runtime. Such dynamic tables are usually obtained as the *result set* of a stored procedure or of a parameterized query. In this case AIMMS will allow you to use a stored procedure call or a dynamically composed SQL query inside the READ statement as if it were a database table. Please note that it's currently not possible to read a result set from an Oracle stored procedure, since Oracle uses a non-standard mechanism for that (involving so-called *ref cursors*).

The DatabaseProcedure declaration

Every stored procedure or SQL query that you want to call from within AIMMS must be declared as a DatabaseProcedure within your application. The attributes of a DatabaseProcedure are listed in Table 27.2.

Attribute	Value type	See also page
DataSource	*string*	447
Arguments	*argument-list*	143
StoredProcedure	*string-expression*	
SQLQuery	*string-expression*	
Owner	*string-expression*	447
Property	UseResultSet	
Mapping	*mapping-list*	448
Comment	*comment string*	19
Convention	*convention*	449, 534

Table 27.2: DatabaseProcedure attributes

SQL query or stored procedure

A DatabaseProcedure in AIMMS can represent either a (dynamically created) SQL query or a call to a stored procedure. AIMMS makes the distinction on the basis of the StoredProcedure and SQLQuery attributes. If the StoredProcedure attribute is nonempty, AIMMS assumes that the DatabaseProcedure represents a stored procedure and expects the SQLQuery attribute to be empty, and vice versa.

The StoredProcedure attribute

With the StoredProcedure attribute you can specify the name of the stored procedure within the ODBC data source that you want to be called. The StoredProcedure wizard will let you select any stored procedure name available within the specified ODBC data source. If the stored procedure that you want to call is not owned by yourself, or if there are name conflicts, you should specify the owner with the Owner attribute.

The SQLQuery attribute

You can use the SQLQuery attribute to specify the SQL query that you want to be executed when the DatabaseProcedure is called. The value of this attribute can be any string expression, allowing you to generate a dynamic SQL query using the arguments of the DatabaseProcedure.

The Arguments attribute

With the Arguments attribute you can indicate the list of *scalar* arguments of the database procedure. The specified arguments must have a matching declaration in a declaration section local to the DatabaseProcedure. If the DatabaseProcedure represents a stored procedure, the argument list is interpreted as the argument list of the stored procedure. When you use the StoredProcedure wizard, AIMMS will automatically enter the argument list, including their AIMMS prototype, for you. For a DatabaseProcedure representing an SQL query, you can use the arguments in composing the SQL query string.

Input-output type

For SQL queries all arguments must be Input arguments, as the query cannot modify them. For stored procedures, the StoredProcedure wizard will by default set the input-output type of each argument equal to its SQL input-output type. However, if you want to discard the result of any output argument, you can change its type to Input.

The Property attribute

With the Property attribute of a DatabaseProcedure you can indicate the intended use of the procedure.

- When you do not specify the property UseResultSet, AIMMS lets you call the DatabaseProcedure as if it were an AIMMS procedure.
- When you do specify the property UseResultSet, AIMMS lets you use the DatabaseProcedure as a parameterized table in the READ statement. In that case, you can also provide a Mapping attribute to specify the mapping from column names in the result set onto the corresponding AIMMS identifiers.

Stored procedure examples

The following declarations will make two stored procedures contained in the data source "Topological Data" available in your AIMMS application. The local declarations of all arguments are omitted for the sake of brevity. They are all assumed to be Input arguments.

```
DatabaseProcedure StoreSingleTransport {
    DataSource       : "Topological Data";
    StoredProcedure  : "SP_STORE_SINGLE_TRANSPORT";
    Arguments        : (from, to, transport);
}
DatabaseProcedure SelectTransportNetwork {
    DataSource       : "Topological Data";
    StoredProcedure  : "SP_DISTANCE";
    Arguments        : MaxDistance;
    Property         : UseResultSet;
    Mapping          : {
        "from"        --> i,
        "to"          --> j,
        "dist"        --> Distance(i,j),
        ("from","to") --> Routes
    }
}
```

The procedure StoreSingleTransport can be used like any other AIMMS procedure, as in the following statement.

```
StoreSingleTransport( 'Amsterdam', 'Rotterdam',
                      Transport('Amsterdam', 'Rotterdam') );
```

The second procedure SelectTransportNetwork can be used in a READ statement as if it were a database table, as illustrated below.

```
read from table SelectTransportNetwork( UserSelectedDistance );
```

SQL query example

The following example illustrates the declaration of a DatabaseProcedure representing a direct SQL query. Its aim is to delete those records in the specified table for which the column VersionCol equals the specified version. Both arguments must be declared as local Input string parameters.

```
DatabaseProcedure DeleteTableVersion {
    DataSource  : "Topological Data";
    Arguments   : (DeleteTable, DeleteVersion);
    SQLQuery    : {
        FormatString( "DELETE FROM %s WHERE VersionCol = '%s'",
                      DeleteTable, DeleteVersion )
    }
}
```

Executing SQL statements directly

In addition to executing SQL queries through DatabaseProcedure, AIMMS also allows you to execute SQL statements directly within a data source. The interface for this mechanism is simple, and forms a convenient alternative for a DatabaseProcedure when you want to execute a single SQL statement only once.

The procedure DirectSQL

You can send SQL statements to a data source by calling the procedure DirectSQL with the following prototype:

- DirectSQL(*data-source*, *SQL-string*)

Both arguments of the procedure should be string expressions. Note that in case the SQL statement also produces a result set, then this set is ignored by AIMMS.

Example

The following call to DirectSQL drops a table called "Temporary_Table" from the data source "Topological Data".

```
DirectSQL( "Topological Data",
           "DROP TABLE Temporary_Table" );
```

Use FormatString

The procedure DirectSQL does not offer direct capabilities for parameterizing the SQL string with AIMMS data. Instead, you can use the function FormatString to construct symbolic SQL statements with terms based on AIMMS identifiers.

27.6 Database transactions

Transactions

By default, AIMMS places a transaction around any *single* WRITE statement to a database table. In this way, AIMMS makes sure that the complete WRITE statement can be rolled back in the event of a database error during the execution of that WRITE statement. You can increase the amount of transactional control over READ, WRITE statements and SQL queries through the procedures

- StartTransaction([*isolation-level*])
- CommitTransaction
- RollbackTransaction

The procedure StartTransaction

With the procedure StartTransaction you can manually initiate a database transaction. As a consequence, you can commit or roll back the changes in the database caused by all WRITE statements and SQL queries executed within the context of the transaction simultaneously. You can specify the exact semantics of the transaction through its only (optional) argument *isolation-level*, which must be an element from the predefined set AllIsolationLevels. You cannot call StartTransaction recursively, i.e. you must call CommitTransaction or CallbackTransaction prior to the next call to StartTransaction. The procedure returns a value of 1 if the transaction was started successfully, or 0 otherwise.

The set AllIsolationLevels

Besides the ability to commit roll back *all* the changes made to the database during the transaction, AIMMS supports the following isolation levels for transactions:

- ReadUncommitted: a transaction operating at this level can see uncommitted changes made by other transactions,
- ReadCommitted (default): a transaction operating at this level cannot see changes made by other transactions until those transactions are committed,
- RepeatableRead: a transaction operating at this level is guaranteed not to see any changes made by other transactions in values it has already read during the transaction, and
- Serializable: a transaction operating at this level guarantees that all concurrent transactions interact only in ways that produce the same effect as if each transaction were entirely executed one after the other.

Note that not all database servers may support all of these isolation levels, and may cause the call to StartTransaction to fail.

The procedure CommitTransaction

Through the procedure CommitTransaction you can commit all the changes that you have made to the database since the previous call to StartTransaction. The function returns a value of 1 of the changes are committed successfully, or 0 otherwise.

The procedure RollbackTransaction

With the procedure RollbackTransaction you can roll back (i.e. undo) all the changes that you have made to the database since the previous call to StartTransaction. The function returns a value of 1 of the changes are rolled back successfully, or 0 otherwise.

27.7 Testing the presence of data sources and tables

Fail safe access

When you want to run an AIMMS-based application on the computer of an end-user, you may want to make sure that the data sources and database tables required to run the application successfully are present, prior to actually initiating any data transfer. Normally, trying to execute a READ and WRITE statements on a nonexisting data source or database table causes AIMMS to generate run-time errors, which might be confusing to your end-users. By first verifying the presence of the required data sources and database tables, you are able to generate error messages which are more meaningful to your end-users.

Syntax

You can test the presence of data sources and database tables on a host computer through the functions

- TestDataSource(*data-source*)
- TestDatabaseTable(*data-source*, *table-name*)

Both *data-source* and *table-name* are string arguments.

The procedure TestDataSource

With the procedure TestDataSource you can check whether the ODBC data source named *data-source* is present on the host computer on which your AIMMS application is being run. The procedure returns 1 if the data source is present, or 0 otherwise.

The procedure TestDatabaseTable

The function TestDatabaseTable lets you check whether a given table named *table-name* exists in the data source named *data-source*. The procedure returns 1 if the database table is present in the given data source, or 0 otherwise. However, the procedure TestDatabaseTable will not let you check whether the table contains the columns which you expect it to contain. If you try to access columns in the database table which are not present during either a READ or WRITE statement, AIMMS will still generate a run-time error to this effect.

27.8 Dealing with date-time values

Mapping date-time values

Special care is required when you want to read data from or write data to a database which represents a date, a time, or a time stamp in the database table. The ODBC technology uses a fixed string format for each of these data types. Most likely, this format will not coincide with the format that you use to store dates and times in your modeling application.

Mapped onto calendars

When a column in a database table containing date-time values maps onto a Calendar in your AIMMS model, AIMMS will automatically convert the date-time values to the associated time slot format of the calendar, and store the corresponding values for the appropriate time slots.

Time zone translation

By default, AIMMS assumes that the date-time values mapped onto a particular CALENDAR are stored in the database according to the same time zone (ignoring daylight saving time) as specified in the TimeslotFormat attribute of that calendar (see also Section 33.7.4). In the absence of such a time zone specification, AIMMS will assume the local time zone (without daylight saving time). You can override the time zone through the TimeslotFormat attribute of a Convention. The use of Conventions with respect to Calendar is discussed in full detail in Section 33.10.

The ODBCDateTimeFormat parameter

If a date-time column in a database table does not map onto a Calendar in your model, you can still convert the ODBC date-time format into the date- or time representation of your preference, using the predefined string parameter ODBCDateTimeFormat defined over the set of AllIdentifiers. With it, you can specify, on a per identifier basis, the particular format that AIMMS should use to store dates and/or times using the formats discussed in Section 33.7. AIMMS

will never perform a time zone conversion for non-calendar data, and will ignore ODBCDateTimeFormat when it contains a date-time format specification for a CALENDAR.

Unmapped columns

If you do not specify a date-time format for a particular identifier, and the column does not map onto a Calendar, AIMMS will assume the fixed ODBC format. These formats are:

- *YYYY-MM-DD hh:mm:ss.tttttt* for date-time columns,
- *YYYY-MM-DD* for date columns, and
- *hh:mm:ss* for time columns.

When you are unsure about the specific type of a date/time/date-time column in the database table during a WRITE action, you can always store the AIMMS data in date-time format, as AIMMS can convert these to both the date and time format. During a READ action, AIMMS will always translate into the type for the column type.

Example

A stock ordering model contains the following identifiers:

- the set Products with index p, containing all products kept in stock,
- an ordinary set OrderDates with index d, containing all ordering dates, and
- a string parameter ArrivalTime(p,d) containing the arrival time of the goods in the warehouse.

The order dates should be of the format '140319', whilst the arrival times should be formatted as '12:30 PM' or '9:01 AM'. Using the time specifiers of Section 33.7, you can accomplish this through the following assignments to the predefined parameter ODBCDateTimeFormat:

```
ODBCDateTimeFormat( 'OrderDates' )  := "%y%m%d";
ODBCDateTimeFormat( 'ArrivalTime' ) := "%h:%M %p";
```

Chapter 28

Format of Text Data Files

This chapter

Data provided in text data files can be provided in scalar, list or tabular format. While the scalar and list formats can also be used in ordinary expressions, the tabular formats are only allowed for data initialization. This chapter discusses the general format of text data files with special emphasis on the two possible tabular formats. Data provided in text files can only be read through the use of the READ statement which is discussed in Chapter 26.2.

28.1 Text data files

Allowed text formats

Text data files must contain one or a sequence of identifier assignments with a *constant* right-hand side. All assignments must be terminated by a semi-colon. The following constant formats can be assigned:

- assignment of *scalar constants*,
- assignment of *constant enumerated set expressions*,
- assignment of *constant enumerated list expressions*,
- assignment of *constant tabular expressions*, and
- assignment via *composite tables*.

The first three formats can also be used in ordinary expressions, and have been discussed in Chapters 5 and 6. The tabular and composite table formats are mostly placed in external data files, and will be discussed in this chapter.

AIMMS *generated output*

When you use the WRITE statement to write the contents of some or all identifiers in your model to a text file, AIMMS will select the appropriate format and write the resulting output accordingly. If you want actual control over the way identifiers are printed, you should use the PUT or DISPLAY statements (see also Sections 31.2 and 31.3).

Easily generated

The text formats allowed in AIMMS are straightforward, and it is not difficult to generate these formats either manually or through an external program. As a result, text files form an ideal input medium when you quickly need to create a small data set to test your AIMMS application, or when data is obtained from a program to which a direct link cannot be made.

Example

The following initialization statements illustrate an arrangement of assignments of scalar constants, constant enumerated sets and lists which can be used in an text data file.

```
Cities      := DATA { Amsterdam, Rotterdam, Antwerp, Berlin, Paris } ;

Supply(i)   := DATA { Amsterdam :  50,
                      Rotterdam : 100,
                      Antwerp   :  75  } ;
PricePerMile := 50 ;

LargestCity := 'Paris' ;
```

Dimensions must match

There is an important rule that applies to any data initialization statement in an text data file: the dimensions of left-hand side identifier and the right-hand side expressions must be equal. For instance, the assignment

```
Supply(i) := 100 ;
```

cannot be made inside an text data file for data initialization. Of course, the above statement is a valid assignment when used inside a procedure in AIMMS.

Reducing the dimension

Sometimes it is more convenient to initialize multidimensional parameters and variables using several tables of lesser dimension than by providing a huge table covering the full index space at once. This is especially convenient when data in your model is supplied in natural portions (for instance, all city-dependent data separate for each city). AIMMS helps you in these situations by allowing you to initialize a *slice* of a parameter or a variable.

Sliced initialization

You can specify a slice of a non-scalar identifier by replacing one or more of its indices by explicit elements. The result of a slice can be either a scalar quantity which you can initialize by assigning a scalar, or a non-scalar quantity which you can initialize using either a enumerated list, a table, or a composite table.

Example

The following data assignments illustrate valid examples of sliced initialization.

```
Supply('Amsterdam')     := 75;

Distance('Amsterdam',j) := DATA { Rotterdam :  85,
                                  Antwerp   : 170,
                                  Berlin    : 660,
                                  Paris     : 530  } ;
```

28.2 Tabular expressions

Tables for initialization

For multidimensional quantities the table format often provides the most natural structure for data entry because elements are repeated less often. Tables can be used in text data files and in the InitialData attribute inside the declaration of an identifier.

Two-dimensional views

A table is a two-dimensional view of a multidimensional quantity. The index tuple of the quantity is split into two parts: row identifiers and column identifiers. Indices may not be permuted.

Example

The following example illustrates a simple example of the table format.

```
Distance(i,j) := DATA TABLE
                 Rotterdam    Antwerp    Berlin    Paris
!                ---------    -------    ------    -----
    Amsterdam        85           170       660      510
    Rotterdam                     100       700      440
    Antwerp                                 725      340
    Berlin                                           1050
;
```

The first line of a table (after the keyword DATA TABLE) contains the column identifiers. Each subsequent line contains a row identifier followed by the table entries.

Multi-dimensional entries

Row and column identifiers may be set elements, tuples of elements, or tuples containing element ranges. As a result, multidimensional identifiers can still be captured within the two-dimensional framework of a table.

Proper spacing

Column identifiers must be separated by at least one space. AIMMS keeps track of the column width by maintaining the first and last position used by each column identifier. Any entry must intersect only one column and is understood to be part of that column. AIMMS will reject any entry that intersects two columns, or falls between them.

Continuation of tables with +

Even though the table format is a convenient way to enter data, the number of columns is always restricted by the width of a line. However, by placing a + on a new line you can continue a table by repeating the table format. Row identifiers and column identifiers can be repeated in each block separated by the + sign, but must be unique within a block.

The following table illustrates a valid example of table continuation, equivalent with the previous example. *Example*

```
Distance(i,j) := DATA TABLE
                 Rotterdam    Antwerp
!                ---------    -------
    Amsterdam        85           170
    Rotterdam                     100
+
                 Berlin    Paris
!                ------    -----
    Amsterdam      660      510
    Rotterdam      700      440
    Antwerp        725      340
    Berlin                 1050
;
```

Tables can be used for the initialization of both parameters and sets. When used for parameter initialization, table entries are either blank or contain explicit numbers, quoted or unquoted set elements and quoted strings. Entries in tables used for set initialization are either blank or contain a "*" denoting membership. *Data and membership tables*

The detailed syntax of a table constant is given by the following diagram, where the symbol "\n" stands for the newline character. *Syntax*

table :

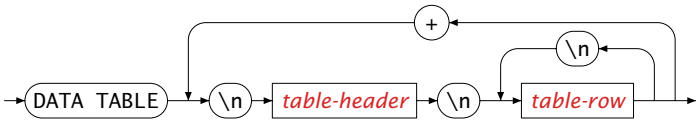
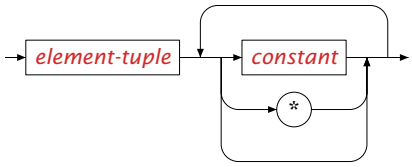


table-header :

table-row :



28.3 Composite tables

Multiple identifiers

A composite table is a bulk form of initialization, and is similar in structure to a table in a database. Using a composite table you can initialize simple sets, relations, parameters, and variables in a single statement. Composite tables always form a single block, and can only be used in text data files.

Format

The first line of a composite table contains column identifiers that define the index columns and the quantity columns. The subsequent lines contain data entries. Like in a tabular expression, entries in a composite table may be either blank or contain explicit numbers, quoted or unquoted set elements and quoted strings, depending on the type of the identifier associated with a column. Blank entries in the quantity columns are treated as "no assignment.", while blank entries in the index columns are not allowed. All data entries must lie directly below their corresponding column identifier as in regular tables.

Indices must come first

The full index space is declared in the first group of column identifiers, and is comparable to the *primary key* in a database table. The remaining column identifiers declare various quantities that must share the identical index space. Note that, unlike in tabular expressions, index columns in a data entry row of a composite table cannot refer to tuples or ranges of elements, but only to single set elements.

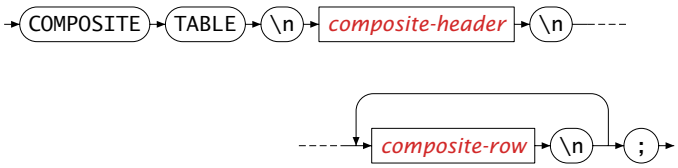
Example

The following statement illustrates a valid example of a composite table. It initializes the relation `Routes`, as well as the parameters `Distance` and `TransportCost`, all of which are defined over the index space (`i`, `j`).

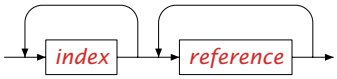
```
COMPOSITE TABLE
    i           j           Routes    Distance    TransportCost
!   ---------   ---------   ------    --------    -------------
    Amsterdam   Rotterdam     *           85        1.00
    Amsterdam   Antwerp       *          170        2.50
    Amsterdam   Berlin                   660       10.00
    Amsterdam   Paris         *          510        8.25
    Rotterdam   Antwerp       *          100        1.20
    Rotterdam   Berlin        *          700       10.00
    Rotterdam   Paris                    440        7.50
    Antwerp     Berlin        *          725       11.00
    Antwerp     Paris         *          340        5.00
    Berlin      Paris                   1050       17.50
;
```

The detailed syntax of the composite table is given by the following diagram, where the symbol "\n" stands for the newline character. *Syntax*

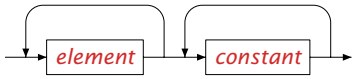
composite-table :



composite-header :



composite-row :



Chapter 29

Reading and Writing Spreadsheet Data

Treating Excel as a database

While it is technically possible to exchange data with Excel through the READ and WRITE statements using the ODBC database connectivity interfaces (see Chapter 27), the Excel ODBC drivers have many limitations. Essential internal SQL statements like UPDATE and DELETE are not supported by these interfaces, effectively making the READ statement the only AIMMS statement which might be used in this setup. For this reason, we strongly recommend you not to use this setup, but to consider to use the spreadsheet functions, described in this chapter, instead. Please note that there are no ODBC drivers available for OpenOffice Calc, so the remarks above apply to the Excel case only.

The Excel Add-In

On the other hand, *from within Excel*, it is possible to exchange data with an AIMMS model in a variety of list and tabular formats using the Excel add-in provided with AIMMS. The Excel add-in is described in full detail in the *Excel Add-In User's Guide*.

The OpenOffice Calc function library

From AIMMS version 3.12 FR1 on, it is also possible to communicate with OpenOffice Calc workbooks from within the AIMMS model (there is no equivalent of the Excel add-in for OpenOffice Calc). The function library is the same as used for the communication with Excel from within the AIMMS model. To use the functions with OpenOffice Calc workbooks instead of Excel workbooks, simply use the extension .ods in the WorkbookName argument of the functions.

This chapter

This chapter provides a brief description of an AIMMS function library that allows you programmatic access, *from within your model*, to the extensive data exchange capabilities provided by the Excel add-in.

29.1 An example

The Excel transport example...

To illustrate the functionality of the Excel add-in, the AIMMS distribution contains an example, which provides a simple Excel workbook that illustrates the use of the Excel add-in. In this example workbook, all the input and output data of a transport model in AIMMS is retained in the workbook and exchanged with AIMMS using the data exchange functionality provided by the Excel add-in. You can find the example in the *Examples* directory of your AIMMS installation.

... started from within AIMMS

In this section, you will learn how the same data exchange could be accomplished from within your model using the spreadsheet function library of AIMMS. The source code illustrated in this section is contained in the AIMMS model accompanying the Excel Link example workbook. Thus, if you run this model in a stand-alone way from within AIMMS, the Excel Link example also serves as an example of the spreadsheet function library.

Retrieving the input data

The input data of the transport model consists of:

- a set Depots with index d,
- a set Customers with index c,
- a parameter Supply(d),
- a parameter Demand(c), and
- a parameter UnitTransportCost(d,c).

Using the spreadsheet function library, the following function calls retrieve all input data from the Excel workbook whose name is stored in the string parameter WorkbookName.

```
Spreadsheet::SetActiveSheet( WorkbookName, "Transport Model" );
Spreadsheet::RetrieveSet( WorkbookName, Depots, "DepotsRange" );
Spreadsheet::RetrieveSet( WorkbookName, Customers, "CustomersRange" );
Spreadsheet::RetrieveParameter( WorkbookName, Supply, "SupplyRange" );
Spreadsheet::RetrieveParameter( WorkbookName, Demand, "DemandRange" );
Spreadsheet::RetrieveTable( WorkbookName, UnitTransportCost,
                    "UnitTransportRange", "DepotsRange", "CustomersRange" );
```

This sequence of function calls, with the exception of the first call, is the direct counterpart of the sequence of actions in the Excel workbook example used to pass the model data to the AIMMS model.

Explained

By calling the function Spreadsheet::SetActiveSheet, you indicate to AIMMS that all following calls operate on a single sheet, allowing you to omit the sheet name as an optional argument in subsequent calls. Through the functions

- Spreadsheet::RetrieveSet,
- Spreadsheet::RetrieveParameter, and
- Spreadsheet::RetrieveTable,

you indicate to AIMMS that the corresponding set and parameter data must be obtained from the specified named Excel ranges. The functionality of these functions is exactly the same as the functionality of the corresponding actions in the Excel add-in. Note that ranges can also be described using the standard A1 and R1C1 styles of Excel.

Writing back the solution

The input data of the transport model consists of:

- a variable Transport(d,c), and
- a variable TotalCost containing the objective value of the optimization model.

These values can be stored in the given workbook using the following function calls.

```
Spreadsheet::SetActiveSheet( WorkbookName, "Transport Model" );
Spreadsheet::AssignParameter( WorkbookName, Transport, "TransportRange", sparse: 1 );
Spreadsheet::AssignValue( WorkbookName, TotalCost, "TotalCostRange" );
```

Explained

Again, these functions provide exactly the same functionality as the corresponding actions in the Excel add-in, and the sequence of function calls corresponds in a one-to-one fashion to the sequence of actions in the Excel workbook example to retrieve the solution back from AIMMS. Through the optional sparse argument of Spreadsheet::AssignParameter you can indicate whether zero values should be passed as 0.0 or as a blank.

Running a macro

The following function call illustrates how a macro contained in a workbook can be run from within your AIMMS model.

```
Spreadsheet::RunMacro( WorkbookName, "AssignRandomTransportCost" );
```

In the Excel Link example this macro is used to randomize the values of the range holding the values of UnitTransportCost. After re-retrieving the input data again and solving the model, this may result in a different optimal solution to the transport model.

29.2 Function overview

Function overview

In this section you will find an overview of all the functions provided by the spreadsheet function library. The function library contains both

- control functions, and
- data exchange functions.

All functions are described in full detail in the Function Reference.

Function naming

From AIMMS 3.12 Feature Release 1 on, the first part of the function names has changed from Excel... to the more general Spreadsheet::..., to reflect the fact that the functions are not exclusively used to communicate with Excel anymore. When you want to work with an OpenOffice Calc workbook, the WorkbookName argument of the functions should end in .ods (which is the extension of Calc workbooks). Any other ending of this argument will result in AIMMS operating on an Excel workbook.

Control functions

The control functions listed in Table 29.1 allow you to perform actions such as opening and closing workbooks and worksheets, copying and printing ranges, and running macros contained in the workbook.

Not in Excel add-in

The control functions listed in Table 29.1 do not have a direct counterpart in the AIMMS Excel add-in. They represent a subset of common spreadsheet commands, which may be convenient when reading and writing data to an Excel or OpenOffice Calc workbook.

Procedure	Description
Spreadsheet::CreateWorkbook	Creates a workbook
Spreadsheet::SaveWorkbook	Saves an opened workbook
Spreadsheet::CloseWorkbook	Closes an opened workbook
Spreadsheet::AddNewSheet	Adds a new sheet to a workbook
Spreadsheet::DeleteSheet	Delete a sheet from a workbook
Spreadsheet::SetActiveSheet	Sets the currently active sheet
Spreadsheet::Print	Prints a range from a workbook
Spreadsheet::ClearRange	Clears the specified range
Spreadsheet::CopyRange	Copies a source into a destination range
Spreadsheet::SetVisibility	Changes the visibility of a workbook
Spreadsheet::SetUpdateLinksBehavior	Sets the behavior w.r.t. linked workbooks
Spreadsheet::ColumnName	Returns the name of a numbered column
Spreadsheet::ColumnNumber	Returns the number of a named column
Spreadsheet::RunMacro	Runs the specified macro

Figure 29.1: Spreadsheet control functions

Data exchange functions

The functions listed in table 29.2 can be used to exchange set data, scalar values, one- and two-dimensional identifiers, and general multi-dimensional identifiers with tabular ranges in an Excel or Calc sheet. Each of these functions corresponds to an associated action in the Excel add-in.

Function	Description
`Spreadsheet::AssignSet`	Assigns set elements to specified range
`Spreadsheet::RetrieveSet`	Fills set with elements from specified range
`Spreadsheet::AssignValue`	Assigns scalar value to specified range
`Spreadsheet::RetrieveValue`	Fills scalar parameter from specified range
`Spreadsheet::AssignParameter`	Assigns 1- or 2-dimensional parameter to raange
`Spreadsheet::RetrieveParameter`	Fills 1- or 2-dimensional parameter from range
`Spreadsheet::AssignTable`	Assigns multi-dimensional parameter to range
`Spreadsheet::RetrieveTable`	Fills multi-dimensional parameter from range

Figure 29.2: Spreadsheet data exchange functions

Chapter 30

Reading and Writing XML Data

This chapter

The Extensible Markup Language (XML) is a universal format for exchanging structured documents and data on the web. For those unfamiliar with XML, Section 30.1 provides a short introduction. It is taken literally from http://www.w3.org/XML/1999/XML-in-10-points, and is copyrighted © 1999–2000 by the W3C organization. Sections 30.2 onwards explain in detail how AIMMS lets you employ the XML data format from within your AIMMS applications.

Further information about XML

If you are unfamiliar with XML, the explanation given here will probably not be sufficient. The best starting point to obtain further information about XML, as well as references to specific XML specifications, formats and tools is the W3C XML site http://www.w3.org/XML/.

30.1 XML in 10 points

XML is for structuring data

Structured data includes things like spreadsheets, address books, configuration parameters, financial transactions, and technical drawings. XML is a set of rules (you may also think of them as guidelines or conventions) for designing text formats that let you structure your data. XML is not a programming language, and you don't have to be a programmer to use it or learn it. XML makes it easy for a computer to generate data, read data, and ensure that the data structure is unambiguous. XML avoids common pitfalls in language design: it is extensible, platform-independent, and it supports internationalization and localization. XML is fully Unicode-compliant.

XML looks a bit like HTML

Like HTML, XML makes use of tags (words bracketed by '<' and '>') and attributes (of the form `name="value"`). While HTML specifies what each tag and attribute means, and often how the text between them will look in a browser, XML uses the tags only to delimit pieces of data, and leaves the interpretation of the data completely to the application that reads it. In other words, if you see "<p>" in an XML file, do not assume it is a paragraph. Depending on the context, it may be a price, a parameter, a person, a p...(and who says it has to be a word with a "p"?).

XML is text, but isn't meant to be read

Programs that produce spreadsheets, address books, and other structured data often store that data on disk, using either a binary or text format. One advantage of a text format is that it allows people, if necessary, to look at the data without the program that produced it; in a pinch, you can read a text format with your favorite text editor. Text formats also allow developers to more easily debug applications. Like HTML, XML files are text files that people shouldn't have to read, but may when the need arises. Less like HTML, the rules for XML files are strict. A forgotten tag, or an attribute without quotes makes an XML file unusable, while in HTML such practice is tolerated and is often explicitly allowed. The official XML specification forbids applications from trying to second-guess the creator of a broken XML file; if the file is broken, an application has to stop right there and report an error.

XML is verbose by design

Since XML is a text format and it uses tags to delimit the data, XML files are nearly always larger than comparable binary formats. That was a conscious decision by the designers of XML. The advantages of a text format are evident (see point 3), and the disadvantages can usually be compensated at a different level. Disk space is less expensive than it used to be, and compression programs like zip and gzip can compress files very well and very fast. In addition, communication protocols such as modem protocols and HTTP/1.1, the core protocol of the Web, can compress data on the fly, saving bandwidth as effectively as a binary format.

XML is a family of technologies

XML 1.0 is the specification that defines what "tags" and "attributes" are. Beyond XML 1.0, "the XML family" is a growing set of modules that offer useful services to accomplish important and frequently demanded tasks. Xlink describes a standard way to add hyperlinks to an XML file. XPointer and XFragments are syntaxes in development for pointing to parts of an XML document. An XPointer is a bit like a URL, but instead of pointing to documents on the Web, it points to pieces of data inside an XML file. CSS, the style sheet language, is applicable to XML as it is to HTML. XSL is the advanced language for expressing style sheets. It is based on XSLT, a transformation language used for rearranging, adding and deleting tags and attributes. The DOM is a standard set of function calls for manipulating XML (and HTML) files from a programming language. XML Schemas 1 and 2 help developers to precisely define the structures of their own XML-based formats. There are several more modules and tools available or under development. Keep an eye on W3C's technical reports page.

XML is new, but not that new

Development of XML started in 1996 and has been a W3C Recommendation since February 1998, which may make you suspect that this is rather immature technology. In fact, the technology isn't very new. Before XML there was SGML, developed in the early '80s, an ISO standard since 1986, and widely used for large documentation projects. The development of HTML started in 1990. The designers of XML simply took the best parts of SGML, guided by the experience

with HTML, and produced something that is no less powerful than SGML, and vastly more regular and simple to use. Some evolutions, however, are hard to distinguish from revolutions... And it must be said that while SGML is mostly used for technical documentation and much less for other kinds of data, with XML it is exactly the opposite.

XML leads HTML to XHTML

There is an important XML application that is a document format: W3C's XHTML, the successor to HTML. XHTML has many of the same elements as HTML. The syntax has been changed slightly to conform to the rules of XML. A document that is "XML-based" inherits the syntax from XML and restricts it in certain ways (e.g, XHTML allows "<p>", but not "<r>"); it also adds meaning to that syntax (XHTML says that "<p>" stands for "paragraph", and not for "price", "person", or anything else).

XML is modular

XML allows you to define a new document format by combining and reusing other formats. Since two formats developed independently may have elements or attributes with the same name, care must be taken when combining those formats (does "<p>" mean "paragraph" from this format or "person" from that one?). To eliminate name confusion when combining formats, XML provides a namespace mechanism. XSL and RDF are good examples of XML-based formats that use namespaces. XML Schema is designed to mirror this support for modularity at the level of defining XML document structures, by making it easy to combine two schemas to produce a third which covers a merged document structure.

XML is the basis for RDF and the Semantic Web

W3C's Resource Description Framework (RDF) is an XML text format that supports resource description and metadata applications, such as music playlists, photo collections, and bibliographies. For example, RDF might let you identify people in a Web photo album using information from a personal contact list; then your mail client could automatically start a message to those people stating that their photos are on the Web. Just as HTML integrated documents, menu systems, and forms applications to launch the original Web, RDF integrates applications and agents into one Semantic Web. Just like people need to have agreement on the meanings of the words they employ in their communication, computers need mechanisms for agreeing on the meanings of terms in order to communicate effectively. Formal descriptions of terms in a certain area (shopping or manufacturing, for example) are called ontologies and are a necessary part of the Semantic Web. RDF, ontologies, and the representation of meaning so that computers can help people do work are all topics of the Semantic Web Activity.

XML is license-free, platform-independent and well-supported

By choosing XML as the basis for a project, you gain access to a large and growing community of tools (one of which may already do what you need!) and engineers experienced in the technology. Opting for XML is a bit like choosing SQL for databases: you still have to build your own database and your own programs and procedures that manipulate it, and there are many tools available and many people who can help you. And since XML is license-free, you can build your own software around it without paying anybody anything. The large and growing support means that you are also not tied to a single vendor. XML isn't always the best solution, but it is always worth considering.

30.2 Introduction to XML support in AIMMS

XML support in AIMMS

In order to help you understand the XML support available within AIMMS to its full extent, this section provides an explanation of the basic concepts used in XML. If you are already familiar with XML and XML schemas, the material in this section may help you to understand how the XML concepts you are already familiar with are used by the XML facilities in AIMMS.

The XML data format

The XML data format is a text format built around just two syntactical components, *elements* and *attributes*. Because the semantics of these components are not fixed and can be user-defined, the XML data format can be used to represent virtually any meaningful concept.

XML elements

XML elements are denoted by a start tag (a word identifying the type of element enclosed by the "<" and ">" characters) and an end tag (the same element type enclosed by the "</" and ">" characters). Elements delimit the piece of XML data between its start and end tag. Elements may hold only text or numeric data, or they can provide depth to XML data, since the enclosed XML data may contain other XML elements. An element in a stream of XML data is, in fact, a node in the tree associated with the entire stream of XML data. The root node of this tree corresponds to the *obligatory and unique* root element of the XML data stream.

XML elements without content

If an element does not contain any enclosed XML content, it is possible to omit the end tag alltogether, and enclose the start tag between "<" and "/>" characters. This format is commonly used if the element contains only attributes.

XML attributes

XML attributes provide additional information about a particular element in an XML data stream. Attributes are specified by the form `name="value"` between the "<" and ">" (or "/>") characters of the start tag of the element in question.

The Data Reconciliation example

All examples in this chapter make use of a single AIMMS project that comes with the AIMMS system, the *Data Reconciliation* project. In this example, flows between units in a chemical plant, as well the chemical composition of these flows, are measured. Unfortunately, such measurements might not be internally consistent despite, for instance, a physical requirement that the sum of all flows into any unit be equal to the sum of all flows out of that unit (i.e. no material is created or lost within a unit). Due to the inaccuracy of the measurement devices (or, even worse, broken measurement devices), the measured values do not necessarily satisfy such balances. The objective of the model is to find a set of flow values and compositions which *are* internally consistent, and lie as close as possible to the corresponding measured values. Such consistent values are called *reconciled* values. Within the model the measured and reconciled values are stored in the identifiers

- `MeasuredFlow(f)`,
- `MeasuredComposition(f,c)`,
- `Flow(f)`, and
- `Composition(f,c)`,

where `f` is an index into a set of `Flows`, and `c` is an index into a set of `Components`. Table 30.1 contains the measured and reconciled flow values and compositions used throughout this chapter.

Flow name	Measured values				
	Flow value [ton/h]	Flow composition [%]			
		N_2	H_2	NH_3	Ar
Inflow	111.98	26.96	72.71		0.33
Mix		24.56			4.91
NH3-Mix		19.99			
NH3-Flow	105.59				
Residue			69.68		
Ar-Flow					
Feedback	358.00				

Flow name	Reconciled values				
	Flow value [ton/h]	Flow composition [%]			
		N_2	H_2	NH_3	Ar
Inflow	117.03	26.96	72.71	0.00	0.33
Mix	475.03	23.95	71.08	0.05	4.91
NH3-Mix	475.03	19.99	59.08	15.27	5.66
NH3-Flow	105.59	0.00	0.00	100.00	0.00
Residue	369.44	23.57	69.68	0.07	6.67
Ar-Flow	11.44	89.19	0.00	0.00	10.81
Feedback	358.00	22.82	70.48	0.07	6.62

Table 30.1: Measured and reconciled values

XML data example

The XML fragment below illustrates a possible XML format to store the measured and reconciled flows and compositions. The root element FlowMeasurementData has a date attribute to indicate the date of the measurements. The root element contains one or more Flow elements which contain the measured (if any) and reconciled flow values for all the flows in the network. Each Flow element contains a single Composition element, which, in turn, contains one or more Component elements with the measured (if any) and reconciled composition values for each component of the flow in question.

```
<FlowMeasurementData xmlns="http://www.aimms.com/Reconciliation" date="2001-10-01">
    <Flow name="Inflow" measured="111.98" reconciled="117.03">
        <Composition>
            <Component name="N2" measured="26.96" reconciled="26.96"/>
            <Component name="H2" measured="72.71" reconciled="72.71"/>
            <Component name="Ar" measured="0.33" reconciled="0.33"/>
        </Composition>
    </Flow>
    <Flow name="Mix" reconciled="475.03">
        <Composition>
            <Component name="N2" measured="24.56" reconciled="23.95"/>
            <Component name="H2" reconciled="71.08"/>
            <Component name="NH3" reconciled="0.05"/>
            <Component name="Ar" measured="4.91" reconciled="4.91"/>
        </Composition>
    </Flow>
    <Flow name="NH3-Mix" reconciled="475.03">
        <Composition>
            <Component name="N2" measured="19.99" reconciled="19.99"/>
            <Component name="H2" reconciled="59.08"/>
            <Component name="NH3" reconciled="15.27"/>
            <Component name="Ar" reconciled="5.66"/>
        </Composition>
    </Flow>
    <Flow name="NH3-Flow" measured="105.59" reconciled="105.59">
        <Composition>
            <Component name="NH3" reconciled="100.00"/>
        </Composition>
    </Flow>
    <Flow name="Residue" reconciled="369.44">
        <Composition>
            <Component name="N2" reconciled="23.57"/>
            <Component name="H2" measured="69.68" reconciled="69.68"/>
            <Component name="NH3" reconciled="0.07"/>
            <Component name="Ar" reconciled="6.67"/>
        </Composition>
    </Flow>
    <Flow name="Ar-Flow" reconciled="11.44">
        <Composition>
            <Component name="N2" reconciled="89.19"/>
            <Component name="Ar" reconciled="10.81"/>
        </Composition>
    </Flow>
    <Flow name="Feedback" measured="358.00" reconciled="358.00">
        <Composition>
            <Component name="N2" reconciled="22.82"/>
            <Component name="H2" reconciled="70.48"/>
            <Component name="NH3" reconciled="0.07"/>
            <Component name="Ar" reconciled="6.62"/>
        </Composition>
```

```
    </Flow>
</FlowMeasurementData>
```

Not unique

The XML data format illustrated above is not unique. For instance, the measured and reconciled values could have been represented by child elements of the Flow and Component elements instead of by attributes. Thus, a different, but equally valid, XML representation of the same data is illustrated in the XML data snippet below.

```
    <Flow name="Inflow">
        <MeasuredValue>111.984</MeasuredValue>
        <ReconciledValue>117.034</ReconciledValue>
        <Composition>
            <Component name="N2">
                <MeasuredValue>26.960</MeasuredValue>
                <ReconciledValue>26.960</ReconciledValue>
            </Component>
            ...
        </Composition>
    </Flow>
```

The particular XML data format chosen may be a matter of taste, or the result of a formal agreement between several parties who wish to use the corresponding XML data.

XML schema

To support you in defining a particular XML data format in a formal manner, XML provides an XML-based standard to specify such definitions. This standard is called *XML Schema*. It allows you, among other things, to specify

- the allowed (tree) structure of a particular XML data format in terms of all possible elements and their child elements,
- the minimum and maximum number of times a particular element can occur,
- which attributes are supported by particular elements,
- whether attributes are optional or required, and
- the intended data types of elements and attributes in your XML data format.

To create an XML schema file that matches an intended XML data format, it is best to use one of the tools available for this purpose. For more detailed information about XML schema, as well as the tools available for creating an XML schema file, refer to http://www.w3.org/XML/Schema

XML schema example

The following XML schema definition, formally defines the XML data format as used in the XML data example above.

```
<xs:schema targetNamespace="http://www.aimms.com/Reconciliation"
           xmlns="http://www.aimms.com/Reconciliation"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="FlowMeasurementData">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Flow" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Composition" minOccurs="0">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Component" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:attribute name="name" type="xs:string" use="required"/>
                        <xs:attribute name="measured" type="xs:double" use="optional"/>
                        <xs:attribute name="reconciled" type="xs:double" use="optional"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="measured" type="xs:double" use="optional"/>
            <xs:attribute name="reconciled" type="xs:double" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="date" type="xs:date" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Schema namespaces

An XML schema definition can specify a namespace by which the schema is to be known. In the example above, the `targetNamespace` attribute of the `xs:schema` element specifies that the schema that follows defines the namespace `http://www.aimms.com/XMLSchema/ReconciliationExample`. In the XML data example listed earlier in this section, the `xmlns` attribute of the root element specifies that all element and attribute names underneath the root element are to be interpreted in the context of that namespace.

Two modes of XML support

AIMMS allows you to read and write XML data from within your model in two modes:

- it lets AIMMS generate and read XML data based on identifier definitions in your model, or
- it lets AIMMS generate and read XML data according to a given XML schema specification.

AIMMS-*generated XML*

In the first mode, AIMMS will generate and read XML for the subset of identifiers that you specify. The format of the generated XML closely resembles the declaration of the identifiers in your model, generates XML data for one identifier at a time, and adds a tree level for each dimension. Letting AIMMS generate XML data for your model is the fastest way of getting XML data that corresponds to your model, but

- gives you little control over the final result, and
- programs that use the generated XML data have to adhere to the generated format.

User-defined XML

In the second mode, AIMMS assumes that you already have a XML schema file that specifies the precise XML data format that you want to generate from within AIMMS, or want to read from an external XML data file. AIMMS provides a tool to let you map the elements and attributes in the XML schema onto sets and multidimensional identifiers in your model. Based on this mapping, and the data in your model, you can let AIMMS generate XML data according to the specified schema, or let AIMMS fill the corresponding identifiers according to an XML data file in the specified format.

30.3 Reading and writing AIMMS-generated XML data

Obtaining generated XML output

Through the functions

- `GenerateXML`(*XMLFile*,*IdentifierSet*[,*merge*][,*SchemaFile*])
- `ReadGeneratedXML`(*XMLFile*[,*merge*])

AIMMS will generate XML output associated with one or more identifiers in your model, or read AIMMS-generated XML from a file.

The function `GenerateXML`

Using the function `GenerateXML`, you can let AIMMS write XML data to the file *XMLFile*. AIMMS will generate XML data for all the identifiers in the *IdentifierSet* argument, which must be a subset of the predefined set `AllIdentifiers`. With the optional *merge* argument (default 0) you can indicate whether you want to merge the generated XML data in another XML document, in which case AIMMS will omit the XML header from the generated XML file. This allows you to merge the contents of the generated file into another XML file. Note that setting the *merge* argument to 1 does not mean that the generated XML data will be appended to the specified file, its contents are always completely overwritten. If you specify a *SchemaFile* name, AIMMS will also generate an XML schema file with the specified file name, matching the generated XML data. All data in the XML file is represented in terms of the currently active unit convention (see also Section 32.8). The function will return 1 if successful, or 0 if not.

The function `ReadGeneratedXML`

With the function `ReadGeneratedXML` you can read back AIMMS-generated XML data from the specified *XMLFile*. With the optional *merge* argument (default 0), you can choose whether you want to merge the data included in the XML file with the existing data, or overwrite any existing data (default). All data in the XML file will be interpreted in accordance with the currently active unit convention (see also Section 32.8). The function will return 1 if successful, or 0 if not.

Fixed format

The XML data format generated by the function GenerateXML solely depends on the declaration of the identifiers for which you want the XML data to be generated. Thus, you can use the generated XML data simply to store some model data in an XML file, ready to be read back into AIMMS through the function ReadGeneratedXML, or by any other program that adheres to the XML data format as generated by AIMMS.

Example

A call to the function GenerateXML, for the identifiers listed in Section 30.2, will result in the following XML data being generated.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<AimmsData>
    <Flows>
        <Flows elem="Inflow"/>
        <Flows elem="Mix"/>
        <Flows elem="NH3-Mix"/>
        <Flows elem="NH3-Flow"/>
        <Flows elem="Residue"/>
        <Flows elem="Ar-Flow"/>
        <Flows elem="Feedback"/>
    </Flows>
    <MeasuredFlow>
        <f elem="Inflow" value="111.98"/>
        <f elem="NH3-Flow" value="105.59"/>
        <f elem="Feedback" value="358.00"/>
    </MeasuredFlow>
    <Flow>
        <f elem="Inflow" value="117.03"/>
        <f elem="Mix" value="475.03"/>
        <f elem="NH3-Mix" value="475.03"/>
        <f elem="NH3-Flow" value="105.59"/>
        <f elem="Residue" value="369.44"/>
        <f elem="Ar-Flow" value="11.44"/>
        <f elem="Feedback" value="358.00"/>
    </Flow>
    <MeasuredComposition>
        <nmf elem="Inflow">
            <c elem="N2" value="26.96"/>
            <c elem="H2" value="72.71"/>
            <c elem="Ar" value="0.33"/>
        </nmf>
        <nmf elem="Mix">
            <c elem="N2" value="24.56"/>
            <c elem="Ar" value="4.91"/>
        </nmf>
        <nmf elem="NH3-Mix">
            <c elem="N2" value="19.99"/>
        </nmf>
        <nmf elem="Residue">
            <c elem="H2" value="69.68"/>
        </nmf>
    </MeasuredComposition>
    <Composition>
        <nmf elem="Inflow">
            <c elem="N2" value="26.96"/>
            <c elem="H2" value="72.71"/>
            <c elem="Ar" value="0.33"/>
        </nmf>
```

```
        <nmf elem="Mix">
            <c elem="N2" value="23.95"/>
            <c elem="H2" value="71.08"/>
            <c elem="NH3" value="0.05"/>
            <c elem="Ar" value="4.91"/>
        </nmf>
        <nmf elem="NH3-Mix">
            <c elem="N2" value="19.99"/>
            <c elem="H2" value="59.08"/>
            <c elem="NH3" value="15.27"/>
            <c elem="Ar" value="5.66"/>
        </nmf>
        <nmf elem="NH3-Flow">
            <c elem="NH3" value="100.00"/>
        </nmf>
        <nmf elem="Residue">
            <c elem="N2" value="23.57"/>
            <c elem="H2" value="69.68"/>
            <c elem="NH3" value="0.07"/>
            <c elem="Ar" value="6.67"/>
        </nmf>
        <nmf elem="Ar-Flow">
            <c elem="N2" value="89.19"/>
            <c elem="Ar" value="10.81"/>
        </nmf>
        <nmf elem="Feedback">
            <c elem="N2" value="22.82"/>
            <c elem="H2" value="70.48"/>
            <c elem="NH3" value="0.07"/>
            <c elem="Ar" value="6.62"/>
        </nmf>
    </Composition>
</AimmsData>
```

Numeric width and precision

Using the AIMMS options XML_number_width and XML_number_precision you can specify the print width and precision of any numerical data generated through the function GenerateXML. The rules are as follows.

- If the option XML_number_width is set to -1, AIMMS will always use scientific format with precision XML_number_precision.
- If the option XML_number_width is 0, AIMMS will print a fixed point floating point number with precision XML_number_precision, provided the number can be represented exactly with the given precision, otherwise scientific format will be used.
- If the option XML_number_width is greater than 0, AIMMS will print a fixed point floating point number with precision XML_number_precision, provided the number to be printed fits within the specified width, otherwise scientific format will be used.

30.4 Reading and writing user-defined XML data

Writing user-defined XML

If you already have a given XML data format to which you want your AIMMS application to adhere, the simple XML generation functions discussed in the previous section will not work. This section discusses the tools and functions provided by AIMMS to help you read and write XML data in a given format, on the explicit assumption that your XML data format is formally described through an XML schema file. If you do not have a schema file for your XML data format, you are advised to use one of the XML schema editors available on the market to construct an XML schema file corresponding to your XML data format.

Mapping XML schema to AIMMS identifiers

Once you do have an XML schema file corresponding to your XML data format, you must create a mapping between the tree structure described by the XML schema, and the identifiers in your AIMMS model that will hold the corresponding data. This mapping is described using another XML format (as illustrated later in this section), which, in principle, can be edited manually. However, to create such a mapping, you can also use the **Tools-XML Schema Mapping** menu. This will ask you to select an XML schema file (with an `.xsd` extension), and will open the **XML Schema Mapping** dialog box for the corresponding schema, as illustrated in Figure 30.1. If there is already an AIMMS

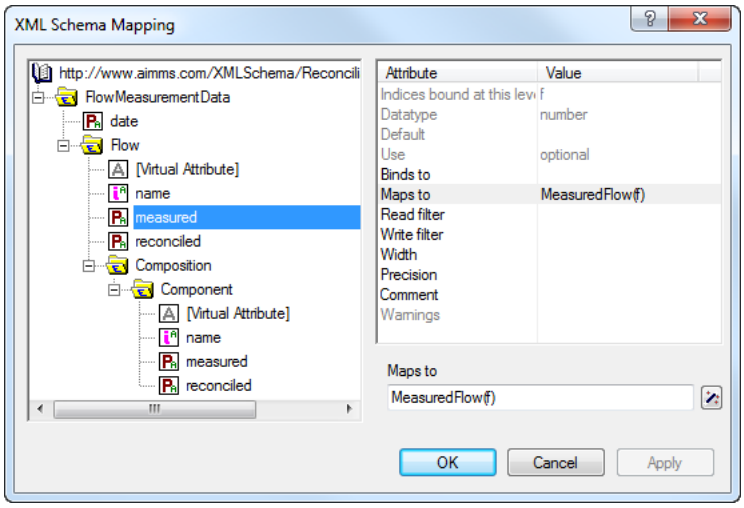


Figure 30.1: The **XML Schema Mapping** dialog box

XML Mapping file (with an `.axm` extension) corresponding to the XML schema file, AIMMS will also read the information in that file, and adapt the attributes of the nodes in the XML mapping tree accordingly.

Binding indices

The mapping between an XML schema and AIMMS is based on the principle that any element that occurs multiple times in an XML data stream can be associated with an index in your AIMMS model. If an index is bound for a particular element, it is also considered to be bound for all attributes and child elements of the element at hand. These can then be mapped onto multidimensional AIMMS identifiers defined over all indices bound at a particular level in the XML tree.

Index value

The element values of such an index associated with an element occurring multiple times in the XML tree can come from several sources:

- a *required* (i.e. non-optional) attribute of the element,
- the data of a direct data-only child element (i.e. without any child elements of its own) which occurs *exactly* once, or
- if there is no such attribute or child element, an element name generated by AIMMS as if there were an attribute containing the generated name.

Examples

In the example of Section 30.2, the index `f` is bound to the element `Flow` through the value of its attribute `name`.

```
<Flow name="Inflow" measured="111.98" reconciled="117.03">
    ...
</Flow>
```

Equally, the index `f`, associated with the `Flow` element, could have obtained its value from the data-only child element `FlowName`, as illustrated below.

```
<Flow>
   <FlowName>Inflow</FlowName>
   <MeasuredValue>111.98</MeasuredValue>
   <ReconciledValue>117.03</ReconciledValue>
    ...
</Flow>
```

Consider the following XML logging format.

```
<LogEntries>
   <Log date="2008-03-31 12:07:31" severity="error">No value for 'Inflow'</Log>
   <Log date="2008-03-31 12:07:35" severity="warning">Error for 'Inflow'</Log>
</LogEntries>
```

None of the attributes nor the element value can uniquely identify a `Log` element. Rather the `LogEntries` element contains a sequence of `Log` entries. The log date, severity and message can perfectly be stored in parameters `LogDate(l)`, `LogSeverity(l)` and `LogMessage(l)`, where the values of index `l` into the set `LogEntries` are numbered elements generated by AIMMS when reading the XML file, and ignored when writing.

Mapped data

In addition to binding indices, the values of attributes and data-only elements can also be mapped to multidimensional identifiers in your model. Such multidimensional identifiers can be defined over a subset of, or all, indices bound at the level of the attribute or element to be mapped. Attributes mapped to multidimensional identifiers may be optional, corresponding to the mapped identifier holding a default value in the AIMMS model.

Example

In the example above, the attributes `measured` and `reconciled` are mapped to the multidimensional identifiers `MeasuredFlow(f)` and `Flow(f)`, respectively. This is a valid mapping since the index `f` is bound at the level of the element `Flow` and hence also to all its child attributes and elements. Similarly, the identifiers `MeasuredFlow(f)` and `Flow(f)` could have been mapped to the data-only elements `MeasuredValue` and `ReconciledValue` in the second part of the example above.

Mapping tree nodes

The **XML Schema Mapping** dialog box displays an XML mapping tree based on the information available in the schema file. The XML mapping tree consists of the following components.

- A single root node `AimmsXMLSchemaMapping`, which contains the single `ElementMapping` node for the root element defined in the XML schema to be mapped. In the **XML Schema Mapping** tree, the `AimmsXMLSchemaMapping` node is displayed by the icon.
- `ElementMapping` nodes, each of which will have zero or more `AttributeMapping`, `VirtualAttributeMapping` and `ElementMapping` child nodes. In the **XML Schema Mapping** tree, a data-only `ElementMapping` node is displayed by either the icon, or the icon when a data-only element is bound to an index, or the icon when a data-only element is mapped to multidimensional data. `ElementMapping` nodes with children are displayed by the icon.
- `AttributeMapping` nodes, which do not have child nodes. In the tree, an `AttributeMapping` node is displayed by either the icon, or the icon when the attribute is bound to an index, or the icon when the attribute is mapped to multidimensional data.
- `VirtualAttributeMapping` nodes, which basically behave like `AttributeMapping` nodes, but have no counterpart in the XML schema. In the tree, a `VirtualAttributeMapping` node is displayed by either the icon, or the icon when the virtual attribute is bound to an index. `VirtualAttributeMapping` nodes are automatically inserted by AIMMS underneath elements that can occur multiple times, and can only be bound to an index. They can be used to let AIMMS generate element names for an index that cannot be bound to a real attribute or child element.

Mapping attributes

To each node type in the mapping tree, a number of possibly node-specific attributes are associated. Some of these attributes are based on the information in the schema file and cannot be edited, while others define the actual mapping between the XML schema and identifiers in your model, and can naturally be edited. When creating a mapping tree, AIMMS will look for an .axm file corresponding to the schema file you selected, and read the actual mapping attributes from the mapping file.

AimmsXMLSchemaMapping attributes

The AimmsXMLSchemaMapping node supports the attributes listed in Table 30.2.

Attribute	Use	Editable	Stored	Value-type
MappedNameSpace	info	no	yes	*namespace-URI*
AimmsModel	optional	yes	yes	*string*
default-width	optional	yes	yes	*integer*
default-precision	optional	yes	yes	*integer*
comment	optional	yes	yes	*string*

Table 30.2: AimmsXMLSchemaMapping attributes

The MappedNameSpace attribute

The MappedNameSpace attribute contains the namespace URI (Universal Resource Identifier) by which the XML schema is identified. AIMMS will retrieve it from the XML schema, whenever the schema contains a namespace URI, or will generate an artificial namespace URI "http://tempuri.org/AIMMS/auto-generated-namespace" if the XML schema does not contain this information.

The AimmsModel attribute

Using the AimmsModel attribute you can indicate the AIMMS model for which the mapping is intended. The information you enter here is solely for your own use, and is ignored by AIMMS when reading or writing XML data according to this mapping.

The default-width and -precision attributes

Using the default-width and default-precision attributes you can specify the width and precision with which you want AIMMS to write numerical data, when writing an XML file subject to this mapping. These attributes override the AIMMS options XML_number_width and XML_number_precision discussed in Section 30.3, and use the same semantics.

ElementMapping attributes

An ElementMapping node supports the attributes listed in Table 30.3. Some attributes used in an element mapping do not apply to all ElementMapping nodes. Binding the contents of an element to an index, or mapping the contents to a multidimensional identifier in your model, is only useful if the element is a data-only element, and not when the element contains child elements. When reading an XML schema file, AIMMS distinguishes between these two types of elements, and omits the attributes for mapping data-only elements whenever

Attribute	Use	Data-only	Editable	Stored	Value-type
name	required	no	no	yes	*string*
occurrence	info	yes	no	no	*string*
datatype	info	yes	no	no	*string*
default	info	yes	no	yes	*string*
binds-to	optional	yes	yes	yes	*index-reference*
maps-to	optional	yes	yes	yes	*reference*
width	optional	yes	yes	yes	*integer*
precision	optional	yes	yes	yes	*integer*
read-filter	optional	no	yes	yes	*expression*
write-filter	optional	no	yes	yes	*expression*
comment	optional	no	yes	yes	*string*

Table 30.3: ElementMapping attributes

appropriate. If the schema file indicates that an element can have a mixed content (i.e. both character data and child elements), AIMMS will ignore the character data.

XML schema-based attributes

The occurrence, datatype and default attributes of an ElementMapping node contain information about the element that is obtained from the XML schema. The values of these attributes cannot be edited, and are displayed in the **XML Schema Mapping** dialog box solely for your information.

The occurrence attribute

The occurrence attribute of an element can hold the values optional/once, optional/many, never, once, or many. If you try to bind an index to an optional data-only element, AIMMS will issue a warning, since this can potentially cause problems when reading an XML file.

The datatype attribute

In the datatype attribute, AIMMS displays the datatypes as either unspecified, number, integer, string, or any, whichever is nearest to the datatype of the element specified in the XML schema. You can use this information to determine to which AIMMS identifiers a particular element can be mapped.

The default attribute

In the default attribute, AIMMS displays the default value of a data-only element as specified in the XML schema file (if any). If there is a default value, this information is also stored in the mapping file, as this information is used by AIMMS to interpret the value of non-existent elements when reading an XML file.

The binds-to attribute

With the binds-to attribute you can indicate that AIMMS must bind the contents of a data-only element to a particular index in your model. The value of the binds-to attribute must be a reference to an index in your AIMMS model.

As explained at the start of this section, the binding propagates to the direct parent of the element, and recursively to any of the child attributes and elements of the parent. Those indices that are bound at a particular level of the tree, are displayed in the **XML Schema Mapping** dialog box in the attribute `Indices bound at this level`, which is automatically updated by AIMMS if you change the value of a `binds-to` attribute.

The `maps-to` attribute

With the `maps-to` attribute you can indicate that the contents of a data-only element must be mapped to a multidimensional identifier in your model (including subsets). The value of this attribute must be a reference to an AIMMS identifier in your model, and can refer to the indices that are bound at the level of the `ElementMapping` in question (or a subset thereof). Note that you might obtain unexpected results when reading XML data if the `maps-to` attribute does not refer to all indices bound at this level. If there are multiple instances of the element (corresponding to indices not used in the identifier), only the value of the most recent instance will be registered. The expression that you specify for this attribute can be a slice of a higher-dimensional identifier, and the indices may also be permuted.

`maps-to` in the presence of `binds-to`

Even if you have specified a `binds-to` attribute for a node in the tree, you are also allowed to specify the `maps-to` attribute as well, which will then be used when reading and writing an XML file in the given XML data format. If, in that situation, the `maps-to` attribute contains a reference to a multidimensional identifier, AIMMS will assign a value of 1.0 to that identifier, or if the `maps-to` attribute contains a reference to a, possibly multidimensional, subset, AIMMS will add the corresponding tuple to the subset. When writing an XML file, AIMMS will always write out the element if the identifier contained in the `maps-to` attribute contains non-default data, even if there is no other data to be written that is defined over the index associated with the `binds-to` attribute.

The `read-filter` attribute

Using the `read-filter` attribute you can specify an AIMMS expression to use as a filter when reading an XML data file. The value of the `read-filter` attribute must be a reference to a multidimensional identifier in your model, similar to the `maps-to` attribute, or can be 0 or 1 (the default). If the value is 0, the element and all its child attributes and elements are ignored when reading an XML file. If the value is a reference to an AIMMS identifier, the element, along with its child attributes and elements, is skipped if the identifier at hand does not contain a nonzero value for the index tuple bound at that particular position in the XML file. If the `read-filter` attribute refers to an identifier that is also read from the XML file, AIMMS will use the value for that identifier as contained in the XML file, provided that this value is read before the corresponding reference to the `read-filter` is evaluated.

The write-filter attribute

With the write-filter attribute you can specify an AIMMS expression to use as a filter when writing an XML data file. The value of the write-filter attribute must be a reference to a multidimensional identifier in your model, similar to the maps-to attribute, or can be 0 or 1. If the value is 0, the element and all its child attributes and elements are ignored when writing an XML file. If the value is 1, the element is always written, regardless of whether there are any nondefault data within your model for that particular element. If there is no nondefault data, AIMMS will write the corresponding default value. If the value is a reference to an AIMMS identifier, the element, along with its child attributes and elements, is skipped if the identifier at hand does not contain a nonzero value for the index tuple bound at that particular position in the XML file.

The width and precision attributes

Using the width and precision attributes of a data-only element you can override the values of the default-width and default-precision attributes of the AimmsXMLSchemaMapping node (or, eventually, of the AIMMS options XML_number_width and XML_number_precision) for the element in question. The attributes will only be used if a maps-to attribute has also been specified. With these options you can determine, for each individual element type, how numerical data will be formatted when writing an XML file.

AttributeMapping attributes

AttributeMapping nodes support the attributes listed in Table 30.4.

Attribute	Use	Editable	Stored	Value-type
name	required	no	yes	*string*
datatype	info	no	no	*string*
default	info	no	yes	*string*
use	info	no	no	*namespace-URI*
binds-to	optional	yes	yes	*index-reference*
maps-to	optional	yes	yes	*reference*
width	optional	yes	yes	*integer*
precision	optional	yes	yes	*integer*
read-filter	optional	yes	yes	*expression*
write-filter	optional	yes	yes	*expression*
comment	optional	yes	yes	*string*

Table 30.4: AttributeMapping attributes

The use attribute

The use attribute contains the value of the attribute of the same name obtained from the XML schema, and indicates whether an XML attribute is optional, required or prohibited. If you try to bind an optional attribute to a index in your AIMMS model, AIMMS will issue a warning, since such bindings may

cause problems when reading an XML file in which the optional attribute is not present.

Other attributes similar to element attributes

The remaining attributes of an AttributeMapping node have identical interpretations to those of an ElementMapping node. For information about these attributes refer to the documentation for the corresponding attributes of ElementMapping nodes above.

Example

The following XML data fragment shows the mapping between the XML data file, illustrated in Section 30.2, and the identifiers

- MeasuredFlow(f),
- Flow(f),
- MappedMeasuredComposition(f,c), and
- MappedComposition(f,c)

which contain the corresponding data in the *Data Reconciliation* project.

```
<AimmsXMLSchemaMapping xmlns="http://www.aimms.com/XMLSchema/AimmsXMLMappingSchema"
            MappedNameSpace="http://www.aimms.com/Reconciliation"
            default-width=16 default-precision=2>
  <ElementMapping name="FlowMeasurementData">
    <AttributeMapping name="date" maps-to="ReconciliationDate"/>
    <ElementMapping name="Flow">
      <AttributeMapping name="measured" maps-to="MeasuredFlow(f)"/>
      <AttributeMapping name="name" binds-to="f"/>
      <AttributeMapping name="reconciled" maps-to="Flow(f)"/>
      <ElementMapping name="Composition">
        <ElementMapping name="Component">
          <AttributeMapping name="measured" maps-to="MappedMeasuredComposition(f,c)"/>
          <AttributeMapping name="name" binds-to="c"/>
          <AttributeMapping name="reconciled" maps-to="MappedComposition(f,c)"/>
        </ElementMapping>
      </ElementMapping>
    </ElementMapping>
  </ElementMapping>
</AimmsXMLSchemaMapping>
```

VirtualAttributeMapping attributes

VirtualAttributeMapping nodes support the attributes listed in Table 30.5. The VirtualAttributeMapping allows you to associate an index with an element that occurs multiple times in your XML file, but which has no unique attribute or child element in the XML schema to which you can bind this index. A VirtualAttributeMapping allows you to still associate such elements with an index, as if there were a virtual, hidden attribute to which you bind. When reading an XML file, the element names associated with that index are then generated by AIMMS either numbered on the basis of a given prefix, or by retrieving the names from the element contents itself. When writing an XML file, the element names associated with an index bound to a VirtualAttributeMapping attribute are ignored.

Attribute	Use	Editable	Stored	Value-type
binds-to	required	yes	yes	*index-reference*
maps-to	optional	yes	yes	*reference*
read-filter	optional	yes	yes	*expression*
write-filter	optional	yes	yes	*expression*
assume-element-value	required	yes	yes	Yes / No
element-prefix	optional	yes	yes	*string*
comment	optional	yes	yes	*string*

Table 30.5: VirtualAttributeMapping attributes

VirtualAttributeMapping attributes

The binds-to, maps-to, read-filter and write-filter have the exact same interpretation as for a normal AttributeMapping. Through the assume-element-value attribute you can indicate whether AIMMS should generate element values when reading, or, when the parent element is a data-only element, whether the element content should be taken as the element value for the index. The default value of the assume-element-value attribute is No. Element names generated by AIMMS are numbered starting from 1, with the prefix specified in the element-prefix attribute.

binds-to is mandatory

Note that the binds-to attribute is required for a VirtualAttributeMapping attribute. The VirtualAttributeMapping node and all changes you made to any of its other attributes in the **XML Schema Mapping** dialog box will be ignored when saving the mapping, unless the binds-to attribute has a value.

Example

Consider the XML logging format discussed above

```
<LogEntries>
   <Log date="2008-03-31 12:07:31" severity="error">No value for 'Inflow'</Log>
   <Log date="2008-03-31 12:07:35" severity="warning">Error for 'Inflow'</Log>
</LogEntries>
```

The following schema mapping maps the contents of this XML file to identifiers LogDate(l), LogSeverity(l) and LogMessage(l), where l is an index into a set LogEntries.

```
<AimmsXMLSchemaMapping xmlns="http://www.aimms.com/XMLSchema/AimmsXMLMappingSchema"
            MappedNameSpace="http://www.aimms.com/LoggingData"
            default-width=16 default-precision=2>
  <ElementMapping name="LogEntries">
    <ElementMapping name="Log" maps-to="LogMessage(l)">
      <VirtualAttributeMapping binds-to="l" assume-element-value="No"
                               element-prefix="logentry-"/>
      <AttributeMapping name="date" maps-to="LogDate(l)"/>
      <AttributeMapping name="severity" maps-to="LogSeverity(l)"/>
    </ElementMapping>
  </ElementMapping>
</AimmsXMLSchemaMapping>
```

When reading the XML file, AIMMS will create two elements 'logentry-1' and 'logentry-2' into the set LogEntries. When writing the XML file, AIMMS will write Log elements whenever there is non-default data for LogDate(l), LogSeverity(l) or LogMessage(l), regardless of the specific format of the elements in the set LogEntries.

A second example

Consider the following XML file

```
<Flows>
   <Flow>Inflow</Flow>
   <Flow>Mix</Flow>
   <Flow>NH3-Mix</Flow>
   <Flow>NH3-Flow</Flow>
   <Flow>Residue</Flow>
   <Flow>Ar-Flow</Flow>
   <Flow>Feedback</Flow>
</Flows>
```

This XML format can be used to represent an AIMMS set Flows with an index f. The following schema mapping accomplishes this.

```
<AimmsXMLSchemaMapping xmlns="http://www.aimms.com/XMLSchema/AimmsXMLMappingSchema"
            MappedNameSpace="http://www.aimms.com/FlowsExample"
            default-width=16 default-precision=2>
  <ElementMapping name="Flows">
    <ElementMapping name="Flow">
      <VirtualAttributeMapping binds-to="f" maps-to="Flows"
                               assume-element-value="Yes"/>
    </ElementMapping>
  </ElementMapping>
</AimmsXMLSchemaMapping>
```

In this mapping, the element values of the Flow elements are taken as the value of a virtual attribute bound to the index f. The maps-to attribute is added to ensure that on reading the set Flows is filled with the encountered flow names, and on writing a Flow element is written out for every element in the set Flows.

Checking and saving the mapping file

On pressing the **OK** button in the **XML Schema Mapping** dialog box, AIMMS checks the validity of your mapping, and reports any errors it encounters. If there are no errors, AIMMS will save (or update) the mapping file associated with the XML schema file (.xsd extension) that you selected when opening the dialog box. The mapping file will be saved as an .axm file, with the same base name as the .xsd file.

Obtaining user-defined XML

Once you have created a mapping file between a given XML schema and the appropriate identifiers in your model, you can use the functions

- WriteXML(*XMLFile*,*MappingFile*[,*merge*])
- ReadXML(*XMLFile*,*MappingFile*[,*merge*][,*SchemaFile*])

to read data from, and write data to, an XML data file in the specified format.

The function WriteXML

The function WriteXML lets AIMMS generate XML data and write it into the file *XMLFile* based on the mapping file *MappingFile*. The optional *merge* argument (default 0) indicates whether you want to merge the generated XML data into another XML document, in which case AIMMS will omit the XML header from the generated XML file. This allows you to merge the contents of the generated file into another XML file. Note that setting the *merge* argument to 1 does not result in the generated XML data being appended to the specified file, its contents are completely overwritten. All data in the XML file are represented with respect to the currently active unit convention (see also Section 32.8). The function will return 1 if successful, or 0 if not.

Adding a namespace

If your XML schema file defines a namespace, reflected in the MappedNameSpace attribute of the root node in the corresponding .axm file, AIMMS will add this namespace to the XML file written by WriteXML through the xmlns attribute the root node of that file. If your XML schema file does not define a namespace, the MappedNameSpace attribute in the .axm file contains an artificial namespace URI "http://tempuri.org/AIMMS/auto-generated-namespace", which will not be added as the xmlns attribute to the root node of the file being written

The function ReadXML

Using the function ReadXML you can let AIMMS read the XML data contained in the file *XMLFile* into the AIMMS identifiers specified in the mapping file *MappingFile*. If the mapping file contains a valid (i.e. not generated by AIMMS) namespace URI of the corresponding XML schema, AIMMS requires the root element of the XML data file to be also associated with the namespace through the xmlns attribute. With the optional *merge* argument (default 0), you may indicate whether you want to merge the data included in the XML file with the existing data, or overwrite any existing data (default). All data in the XML file will be interpreted in accordance with the currently active unit convention (see also Section 32.8). The function will return 1 if successful, or 0 if not.

Schema validation

If you specify an optional *SchemaFile*, the XML parser used by AIMMS will validate the contents of the XML data contained in your XML file against this schema. This will only work, however, if the specified schema file defines a namespace matching the xmlns attribute of the root node of your XML file.

Chapter 31

Text Reports and Output Listing

Reporting facilities

The AIMMS system has several reporting features to present model results to you or an end-user.

- The *graphical (end-)user interface* lets you not only view your model results, but also change input values and run the model interactively. In general, the graphical user interface is the most convenient and direct way to verify model results and view the effect of input changes.
- A *print page* allows you to obtain a hard-copy of your graphical model results. It is created in the graphical user interface of AIMMS and can contain the same objects as pages in the end-user interface. Single print pages or reports composed of multiple print pages can be printed either from within the end-user interface or from within the model. Printing pages and the available functions that you can use in your model to initiate printing is discussed in the AIMMS User's Guide.
- An *text report* lets you save your model results in files. It is created as part of your model using PUT and DISPLAY statements. The result can be written to either a file or to a text window in the graphical user interface. Text reports are convenient, for instance, when you need to generate a special format input file for an external program.
- The *listing file* lets you view the contents of all constraints and variables of a particular mathematical program in your model just before or after solving it. The listing file is a convenient medium for debugging the precise contents of the constraints in a mathematical program generated on the basis of your model and data.

This chapter

This chapter concentrates on the last two reporting media. It explains how to create and print text reports. More specifically, it discusses the File declaration, as well as the PUT and DISPLAY statements. It also explains how you can optionally create a text report consisting of pages each built up of a header, footer and data area. The remaining part of the chapter will explain the format of the constraint and solution listings generated by AIMMS.

31.1 The File declaration

File declaration

External file names that you want to use for reporting must be linked to AIMMS identifiers in your model. In this way, external file names become data. Whenever you want to send output to a particular external file, you must refer to its associated identifier. This linking is achieved using a File declaration, the attributes of which are given in Table 31.1.

Attribute	Value-type	See also page
Name	*string-expression*	
Device	disk, window, void.	
Mode	replace, merge	
Encoding	an element in AllCharacterEncodings	18
Text	*string*	19
Comment	*comment string*	19
Convention	*convention*	449, 534

Table 31.1: File attributes

The Name attribute

With the Name attribute you can specify the actual name of the disk file or window that you want to refer to. If the file identifier refers to a disk file, the Name will be the file name on disk. If it refers to a window the Name attribute will serve as the title of the window. If you do not specify a name, AIMMS will construct a default name, using the internal identifier name as the root and ".put" as the extension.

The Device attribute

The Device attribute can have three values

- disk (default),
- window, and
- void.

You can use it to indicate whether the output should be directed to an external file on disk, a window in the graphical user interface, or whether no output should be generated at all. This latter void device is very convenient, for instance, to hide output statements in your code that are useful during the development of your model but should not be displayed in an end-user version.

The Mode attribute

You can use the Mode attribute to specify whether the file or window should be overwritten (replace mode, default), or appended to (merge mode). The graphical window in the user interface differs from a file in that it can be closed manually by the user. In this case, its contents are lost and AIMMS starts writing to a new instance regardless of the mode.

Example

The following File declarations illustrate the declaration of a link to the external file "result.dat" in the Output subdirectory of the project directory, and a text window that will appear with the title "Model results". The contents of ResultFile will be overwritten whenever it is opened, while the window ResultWindow will be appended to whenever possible.

```
File ResultFile {
    Name       : "Output\\result.dat";
    Device     : disk;
    Mode       : replace;
}
File ResultWindow {
    Name       : "Model results";
    Device     : window;
    Mode       : merge;
}
```

The Encoding attribute

In the Encoding attribute of a file, a specific character encoding can be specified for that file, either as a specific element of the set AllCharacterEncodings or as an element parameter with the set AllCharacterEncodings as its range. Encodings are explained in Paragraph *Text files* on Page 18. In the example below, the attribute Encoding states that code page WINDOWS-1252 should be used for the file WindmillLocations.txt. This code page is not uncommon in the Netherlands.

```
File WindMillLocs {
    Name       :  "WindmillLocations.txt";
    Encoding   :  'WINDOWS-1252';
}
```

The statement Write to file WindMillLocs ; will subsequently write the file "WindmillLocations.txt" using the character encoding WINDOWS-1252. When the Encoding attribute is not specified, the statements Read from file and Write to file will use the encodings specified by the options default_input_character_encoding and default_output_character_encoding respectively. The default of these options is the preferred encoding UTF8. The Encoding attribute is ignored when reading from files which start with a Unicode BOM (Byte Order Mask).

The Convention attribute

With the Convention attribute you can indicate that AIMMS must assume that the data in the file is to be stored according to the units provided in the specified convention. If the unit specified in the convention differs from the unit in which AIMMS stores its data internally, the data is scaled just prior to data transfer. For the declaration of Conventions you are referred to Section 32.8.

31.2 The PUT statement

Customized text reports

AIMMS provides two statements to create a customized text output report in either a file or in a text window in the user interface. They are the PUT and the DISPLAY statements. The result of these statements must always be directed to either a single file or a window.

Basic steps

The following steps are required to create a customized text report:

- direct the output to the appropriate File identifier, and
- print one or more strings, set elements, numerical items, or tabular arrangements of data to it.

These basic operations are the subject of the subsequent subsections. At the end of the section, an extended example will illustrate most of the discussed features.

Stream versus page mode

AIMMS can produce text reports in two modes. They are:

- *stream mode*, in which all lines are printed consecutively, and
- *page mode*, where the report is divided into pages of equal length, each consisting of a header, a footer and a data area.

Most aspects, such as opening files, output direction, and formatting, are the same for both reporting modes. Only the structuring of pages is an extra aspect of the page mode, and is discussed in Section 31.4.

31.2.1 Opening files and output redirection

Opening files

Disk files and windows are opened automatically as soon as output is written to them. You can send output to a particular file by providing the associated File identifier as the first argument of a PUT statement, which designates the file as the *current file*.

PUT without file identifier

If you use the DISPLAY statement or any of the PUT operators listed in Table 31.2 without a file identifier, AIMMS will direct the output to the current file, i.e., the file last opened through the PUT statement.

Undirected output

When you have not yet selected a current file yet, AIMMS will send the output of any PUT or DISPLAY statement to the standard listing file associated with your model.

Example

The following statements illustrates how to send output to a particular file.

```
PUT ResultFile ;
PUT "The model results are:" ;
DISPLAY Transport ;
PUTCLOSE;
```

The first PUT statement sets the current file equal to ResultFile, causing the output of the subsequent PUT and DISPLAY statements to be directed to it. The final PUTCLOSE statement will close ResultFile.

File identifiers only

Unlike other statements like READ and WRITE which allow you to represent files by strings or string parameters as well, the PUT statement requires that you use a File identifier to represent the output file. The way in which output to a file is generated by the PUT statement is completely controlled by the suffices associated with the corresponding File identifier (see also Section 31.4).

Accessing the current file

AIMMS has two pre-defined identifiers that provide access to the current file. They are

- the element parameter CurrentFile (into the set AllIdentifiers) containing the current File identifier, and
- the string parameter CurrentFileName, containing the file name or window title associated with the current file identifier.

The parameter CurrentFileName is output only.

Changing the current file

To select another current file, you can use either of two methods:

- use the PUT statement to (re-)direct output to a different file, or
- set the identifier CurrentFile to the File identifier of your choice.

Closing external files

Closing an external file can be done in two ways:

- automatically, by quitting AIMMS at the end of a session, or
- manually by calling "PUTCLOSE *file-identifier*" during execution.

Files left open

If you leave a file open during the execution of a procedure, AIMMS will temporarily close it at the end of the current execution, and re-open it in *append mode* at the beginning of a subsequent execution. This enables you to inspect the PUT files in between runs.

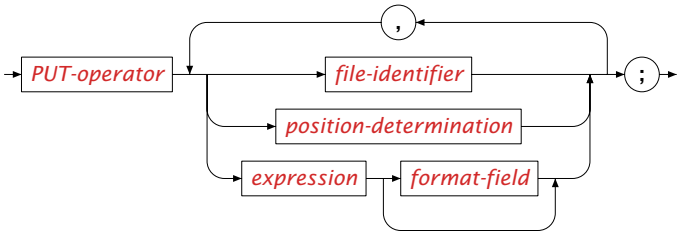
31.2.2 Formatting and positioning PUT items

The PUT statement

Besides selecting the current file, the PUT statement can be used to output one or more individual strings, numbers or set elements to an external text file or window. Each item can be printed in either a default or in a customized manner. The syntax of the PUT statement follows.

Syntax

put-statement :



PUT operators

All possible variants of the PUT operator are listed in Table 31.2. The PUT and PUTCLOSE operators can be used in both stream mode and page mode. The operators PUTHD, PUTFT and PUTPAGE only make sense in page mode, and are discussed in Section 31.4.

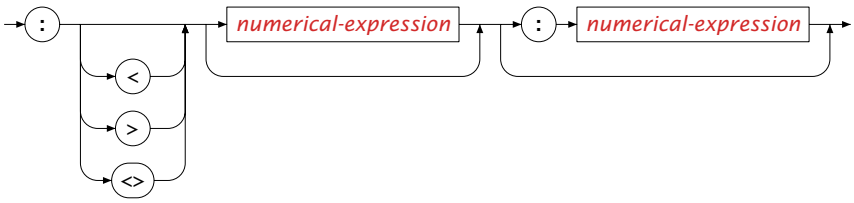
Statement	Description	Stream mode	Page mode
PUT	Direct output or write output	•	•
PUTCLOSE	PUT and close current file	•	•
PUTHD	Write in header area		•
PUTFT	Write in footer area		•
PUTPAGE	PUT and output current page		•

Table 31.2: PUT keywords

Put items are always scalar

All PUT operators only accept scalar expressions. For each scalar item to be printed you can optionally specify a format field, with syntax:

format-field : *Syntax*



With the format field you specify *Format fields*

- whether the item is to be centered, left aligned or right aligned,
- the field width associated with an identifier, and
- the precision.

Customized default values for the justification, field width and precision can be specified through PUT-related options, which can be set via the Options menu. Table 31.3 shows a number of examples of format fields, where m and n are expressions evaluating to integers.

PUT argument	Justification	Field width (characters)	Precision
item	default	default	default
item:m	default	m	default
item:m:n	default	m	n
item:<m:n	left	m	n
item:>m:n	right	m	n
item:<>m:n	centered	m	n

Table 31.3: Format specification of PUT arguments

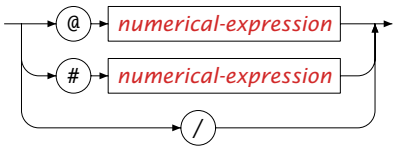
For numerical expressions the precision is the number of decimals to be displayed. For strings and set elements the precision is the maximum number of characters to be displayed. The numbers or characters are placed into a field with the indicated width, and are positioned as specified. *Interpretation of precision*

The PUT syntax for formatting and displaying multiple items on a single line is somewhat similar to the reporting syntax in programming languages like FORTRAN or PASCAL. If you are a C programmer, you may prefer to construct and format a single line of text using the FormatString function (see also Section 5.3.2). In this case you only need the PUT statement to send the resulting string to a text report or window. *Using the FormatString function*

Position determination

For advanced reporting the PUT statement allows you to directly position the cursor at a given row or column. The syntax is shown in the following syntax diagram.

position-determination :



How to position

There are three special arguments for the PUT statement that can be used to position printable items in a file:

- the "@" operator—for horizontal positioning on a line,
- the "#" operator—for vertical positioning, and
- the newline operator "/".

These three operators are explained in Table 31.4, where the symbols k and l are expressions evaluating to integers.

Operator	Meaning
@k	Start printing the next item at column k of the current line.
#l	Goto line l of current page (page mode only).
/	Goto new line.

Table 31.4: Position determination

Page mode only

Using the vertical positioning operator # only makes sense when you are printing in page mode. When printing in stream mode all lines are numbered consecutively from the beginning of the report, and added to the output file or window as soon as AIMMS encounters the newline character /. In page mode, AIMMS prints pages in their entirety, and lines are numbered per page. As a result, you can write to any line within the current page.

31.2.3 Extended example

Example

This example builds upon the transport model used throughout the manual. The following group of statements will produce a text report containing the contents of the identifiers Supply(i), Demand(j) and Transport(i,j), in a combined tabular format separated into right aligned columns of length 12.

The statements

```
! Direct output to ResultFile
put ResultFile ;

! Construct a header for the table
put @13, "Supply":>12, @30, "Transport":>12, /, @30 ;

for ( j ) do   put j:>12 ;   endfor ;
put // ;

! Output the values for Demand
put "Demand", @30 ;
for ( j ) do   put Demand(j):>12:2 ;   endfor ;
put // ;

! Output Supply and Transport
for ( i ) do
    put i:<12, Supply(i):>12:2, @30 ;
    for ( j ) do   put Transport(i,j):>12:2 ;   endfor;
    put / ;
endfor ;

! Close ResultFile
putclose ResultFile ;
```

The produced report

For a particular small data set containing only three Dutch cities, the above statements could result in the following report being generated.

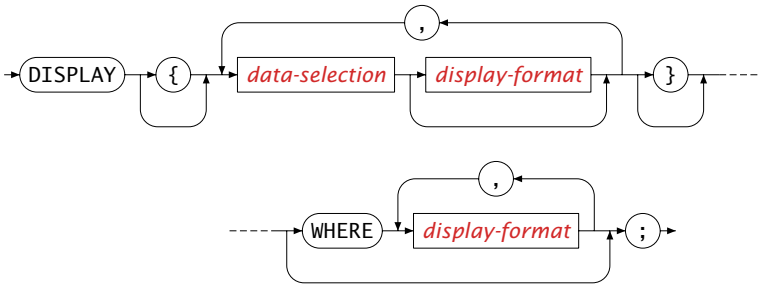
```
                 Supply        Transport
                               Amsterdam   Rotterdam    Den Haag

Demand                              5.00       10.00       15.00

Amsterdam           10.00           2.50        2.50        5.00
Rotterdam           12.50           2.50        5.00        5.00
Den Haag             7.50           0.00        2.50        5.00
```

31.3 The DISPLAY statement

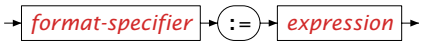
Output in AIMMS *format*

You can use the DISPLAY statement to print the data associated with sets, parameters and variables to a file or window in AIMMS format. As this format is also very easy to read, the DISPLAY statement is an excellent alternative for printing indexed identifiers.

display-statement : *Syntax*



display-format :



Display format

All data selections of a DISPLAY statement are printed by AIMMS in the form of a data assignment.

- Sets are printed in the form of a set assignment with an *enumerated set* on the right-hand side.
- (Slices of) parameters and variables are printed in the form of data assignments, which can be either a table format, a list format, or a composite table.

For indexed parameters and variables AIMMS uses a default display format which is dependent on the dimension.

Overriding the display format

You can override the default AIMMS format by specifying a *display format*, consisting of one or more format specifications, in the WHERE clause. AIMMS supports the following format specifiers:

- DECIMALS: the number of decimals to be printed for each entry,
- ROWDIM: the dimension of the row space,
- COLDIM: the dimension of the column space, and
- COLSPERLINE: the desired numbers of columns per line.

When a format specifier is not specified, AIMMS will use the system default.

Number of decimals

All format specifications in a WHERE clause are applied to the entire collection of data selections printed in the DISPLAY statement. By specifying a DECIMAL format specifier for a particular data selection in the DISPLAY statement, you can also override the number of decimals printed for each data selection individually. You cannot specify other format specifiers for individual data selections.

Obtaining lists and tables

If you have set the dimension of either the row or column space to zero, AIMMS will print the identifier in list format. If both the dimension of the row and column space are greater than zero, AIMMS will print the identifier as a table. AIMMS will honor your request to print the desired number of columns per line if the resulting width does not exceed the default page width. In the latter case, AIMMS will reduce the number of columns until they fit within the requested page width. The default page width can be set as an option within your project.

Outer indices for slicing

If the sum of the dimensions of the row and column space is less than the dimension of the parameter or variable to be displayed, AIMMS will display the identifiers as slices of the requested format, where the slices are taken by fixing the first indices in the domain.

Composite tables

When all arguments of the DISPLAY statement have the same domain and you enclose them by braces, AIMMS will print their values as a single composite table. In this case, you can only specify the precision with which each column must be printed. AIMMS will ignore any of the other display options in combination with the composite table format.

Example

The following statements illustrate the use of the DISPLAY statement and its various display options.

- The following statement will display the data of the variable Transport with 2 decimals and in the default format.

  ```
  display Transport where decimals := 2;
  ```

  The execution of this statement results in the following output being generated.

  ```
  Transport :=
  data table
           Amsterdam   Rotterdam  'Den Haag'
        !  ----------  ----------  ----------
  Amsterdam        2.50        2.50        5.00
  Rotterdam        2.50        5.00        5.00
  'Den Haag'                   2.50        5.00
  ;
  ```

- The following statement displays the subselection of the slice of the variable Transport consisting of all transports departing from the set LargeSupplyCities.

  ```
  display Transport(i in LargeSupplyCities,j) where decimals := 2;
  ```

  This statement will result in the following table, assuming that LargeSupplyCities contains only Amsterdam and Rotterdam.

  ```
  Transport :=
  data table
  ```

```
              Amsterdam   Rotterdam  'Den Haag'
          !  ----------  ----------  ----------
 Amsterdam        2.50        2.50        5.00
 Rotterdam        2.50        5.00        5.00
    ;
```

- The following DISPLAY statement displays Transport with no rows, two columns (i.e. in list format), and two entries per line.

  ```
  display Transport where decimals:=2, rowdim:=0, coldim:=2, colsperline:=2;
  ```

  The resulting output looks as follows.

  ```
  Transport := data
  { ( Amsterdam , Amsterdam  ) : 2.50,  ( Amsterdam , Rotterdam  ) : 2.50,
    ( Amsterdam , 'Den Haag' ) : 5.00,  ( Rotterdam , Amsterdam  ) : 2.50,
    ( Rotterdam , Rotterdam  ) : 5.00,  ( Rotterdam , 'Den Haag' ) : 5.00,
    ( 'Den Haag', Rotterdam  ) : 2.50,  ( 'Den Haag', 'Den Haag' ) : 5.00 } ;
  ```

- In the following DISPLAY statement the row and column display dimensions do not add up to the dimension of Transport.

  ```
  display Transport where decimals:=2, rowdim:=0, coldim:=1, colsperline:=3;
  ```

  As a result AIMMS considers the indices corresponding to the dimension deficit as outer, and displays Transport by means of three one-dimensional displays, each of the requested dimension.

  ```
  Transport('Amsterdam', j) := data
  { Amsterdam  : 2.50,  Rotterdam  : 2.50,  'Den Haag' : 5.00 } ;

  Transport('Rotterdam', j) := data
  { Amsterdam  : 2.50,  Rotterdam  : 5.00,  'Den Haag' : 5.00 } ;

  Transport('Den Haag', j) := data
  { Rotterdam  : 2.50,  'Den Haag' : 5.00 } ;
  ```

- The following DISPLAY statement illustrates how a composite table can be obtained for identifiers defined over the same domain, with a different number of decimals for each identifier.

  ```
  display { Supply decimals := 2, Demand decimals := 3 };
  ```

  Execution of this statement results in the creation of the following one-dimensional composite table.

```
Composite table:
    i             Supply   Demand
!   ----------    ------  -------
    Amsterdam      10.00    5.000
    Rotterdam      12.50   10.000
    'Den Haag'      7.50   15.000
    ;
```

31.4 Structuring a page in page mode

Page-based files

In addition to the continuous stream mode of operation of the PUT statement discussed in the previous section, AIMMS also provides a page-based file format. AIMMS divides a page-based file into pages of a specified length, each consisting of a header, a body, and a footer. Figure 31.1 gives an overview of a page in a page-based report.

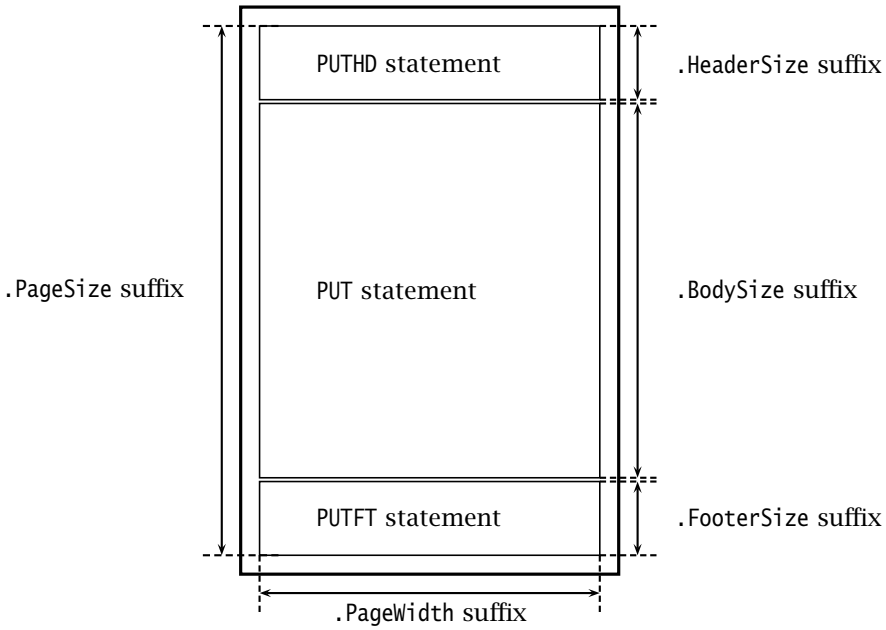


Figure 31.1: Overview of a page in a page based report

Switching to page mode

You can switch between page and stream by setting the .PageMode suffix of a file identifier to 'on' or 'off' (the elements of the predefined set OnOff), respectively, as in the statement ResultFile.PageMode := 'on'. The value of the .PageMode suffix is 'off' by default. When switching to another mode AIMMS will begin with a new page or close the last page.

Page size and width

The default page size is 60 lines. You can overwrite this default by setting the .PageSize suffix of the file identifier to another positive integer value. For instance, ResultFile.PageSize := 10 will give short pages with only ten lines per page. The default page width is 132 columns. You can change this default by setting the .PageWidth suffix of the file identifier.

Headers and footers

The header and footer of a document can be specified by using the PUTHD and PUTFT statements. They are equivalent to the PUT statement but write in the header and footer area instead of in the page body. The size of the header and footer is not preset, but is determined by the contents of the PUTHD and PUTFT statements. The header and footer keep their contents from page to page.

Margins

There are no specific attributes for either the top, bottom, left or right margins of a page. You essentially control these margins by either resizing the header or footer of a page, or by positioning the PUT items in a starting column of your choice using the @ operator of the PUT statement.

Page structure

Table 31.5 summarizes the file attributes for structuring pages. With the exception of the page body size (read only) you can modify their defaults by using assignment statements.

Suffix	Description	Default
.PageMode	Mode	'off'
.PageSize	Page size	60
.PageWidth	Page width	132
.PageNumber	Current page number	1
.BodyCurrentColumn	Body current column	–
.BodyCurrentRow	Body current row	–
.BodySize	Body size	–
.HeaderCurrentColumn	Header current column	–
.HeaderCurrentRow	Header current row	–
.HeaderSize	Header size	–
.FooterCurrentColumn	Footer current column	–
.FooterCurrentRow	Footer current row	–
.FooterSize	Footer size	–

Table 31.5: Page structure attributes

Positioning in page mode

The positioning operators @, #, and / explained in Section 31.2 are also applicable in page mode. However, AIMMS offers you additional file attributes for positioning items in a page-based file.

Current row and column

Whenever you PUT an item into a header, footer, or page body, there is a current row and a current column. AIMMS keeps track of which row and column are current through the suffices .BodyCurrentRow and .BodyCurrentColumn of the File identifier. You can either read or overwrite these values using assignment statements. Similar suffices also exist for the header and the footer area.

Modifying size of page sections

After having specified the header, footer, or page body, you may want to change their size at some stage during the process of writing pages. By specifying the .BodySize, .HeaderSize and .FooterSize suffices you can modify the size (or empty) the page body, the header, or the footer. The value of the .BodySize suffix can be at most the value of the .PageSize suffix minus the value of the .HeaderSize and .FooterSize suffices.

Printing the page number

Whenever you write the contents of the .PageNumber suffix of a File identifier in its header or the footer area, AIMMS will replace it with the current page number whenever it prints a page of a page based report. By default, the first page will be numbered 1, but you can override this by assigning another value to the .PageNumber suffix.

31.5 The standard output listing

The .lis extension

AIMMS produces a standard output listing file for each run of a procedure and each solution of a mathematical program. The name of this listing is the base name of the model file with the extension ".lis". The listing is optionally generated during the first execution in a session and, depending on the option settings, can also be generated during subsequent execution—after updates to parameters and variables.

Contents of a listing file

A standard output listing file can contain one or more of the following items:

- a *source listing*—the source code as compiled,
- a *constraint listing*—a printout of the generated individual constraints of a mathematical program,
- a *solution listing*—the solution values for its variables and constraints,
- a *solver status file*—a progress report on the solution process, and
- any *undirected text output* produced from PUT or DISPLAY statements.

Output activated via options

By default, the standard output listing will be empty unless you set options that activate AIMMS to print one or more of the items in the list above. By not setting options, you avoid the creation of lengthy output files every time you run a model. In addition, you speed up the solution process by avoiding unnecessary overhead.

Examining the solution process

Whenever you want to inspect the model at the individual constraint level, or want to examine the performance of the solver in some detail, then a listing file is your ultimate source of information. The required options for the production of this file can be set from within the model text or from within the graphical interface of AIMMS. They are retained with your project. For more specific information on each of the available options, please consult the AIMMS help file.

Example

After setting the option constraint_listing to 1, AIMMS produces the following standard listing for the transport model used throughout this manual. The model uses a small example data set containing just a few Dutch cities. A detailed explanation of the listing format is given at the end.

The constraint listing

```
This is the first constraint listing of TransportModel.

----  MeetDemand

MeetDemand('Amsterdam') .. [ 1 | 1 | after ]

    + 1 * Transport('Amsterdam' ,'Amsterdam' ) + 1 * Transport('Rotterdam' ,'Amsterdam' )
    >= 5.88 ; (lhs=5.88)

MeetDemand('Rotterdam') .. [ 1 | 2 | after ]

    + 1 * Transport('Amsterdam' ,'Rotterdam' ) + 1 * Transport('Rotterdam' ,'Rotterdam' )
    >= 12.4 ; (lhs=12.4)

MeetDemand('Den Haag') .. [ 1 | 3 | after ]

    + 1 * Transport('Amsterdam' ,'Den Haag'  ) + 1 * Transport('Rotterdam' ,'Den Haag'  )
    >= 12.8 ; (lhs=12.8)

----  MeetSupply

MeetSupply('Amsterdam') .. [ 1 | 4 | after ]

    + 1 * Transport('Amsterdam' ,'Amsterdam' ) + 1 * Transport('Amsterdam' ,'Rotterdam' )
    + 1 * Transport('Amsterdam' ,'Den Haag'  )
    <= 16 ; (lhs=15.1)

MeetSupply('Rotterdam') .. [ 1 | 5 | after ]

    + 1 * Transport('Rotterdam' ,'Amsterdam' ) + 1 * Transport('Rotterdam' ,'Rotterdam' )
    + 1 * Transport('Rotterdam' ,'Den Haag'  )
    <= 16 ; (lhs=16)

----  TotalCost_definition

TotalCost_definition .. [ 1 | 6 | after ]

    + 1 * TotalCost
    - 3.34 * Transport('Amsterdam','Amsterdam') - 11.7 * Transport('Amsterdam','Rotterdam')
    -   13 * Transport('Amsterdam','Den Haag' ) -    9 * Transport('Rotterdam','Amsterdam')
    -    2 * Transport('Rotterdam','Rotterdam') -    3 * Transport('Rotterdam','Den Haag' )
    = 0 ; (lhs=0)
```

Explanation

The above listing contains all the individual constraints generated by AIMMS on the basis of the model formulation and the particular data set loaded at the time of the SOLVE statement. Each individual constraint name is followed by three entries within square brackets.

- The first entry represents the number of times that a SOLVE statement has been executed.

- The second entry is a consecutive number assigned to each individual constraint being printed.
- The third entry indicates when the constraint listing is generated (either "before" or "after" a SOLVE statement has been executed).

Bracketed at the end of each constraint is the value of the left-hand side. You can compare this with the right-hand side to evaluate the status of the constraint. By setting the option constraint_variable_values to 1 you get a more extensive listing that also includes the values and bounds of the variables that are included in each constraint.

Solution listing

The following solution listing results from setting the option solution_listing to 1. Note that the listing includes values for each of the suffices attached to variables and constraints. The status column for variables indicates whether or not the variable is basic, frozen, at bound, or bound exceeded. Similarly, the status column for constraints indicates the same basis and bound information as for variables.

Example solution listing

```
This is the first solution report of TransportModel after a solve.

The 1 scalar variable:
Name       Lower    level  Upper  ReducedCost  Status
---------  -----  -------  -----  -----------  ------
TotalCost   -inf  172.079    inf            0  Basic

The variable "Transport(i,j)" contains the following 6 columns:
         i           j  Lower   level  Upper  ReducedCost  Status
----------  ----------  -----  ------  -----  -----------  ------
Amsterdam   Amsterdam       0   5.880    inf        0.000  Basic
Amsterdam   Rotterdam       0   9.200    inf        0.000  Basic
Amsterdam   'Den Haag'      0   0.000    inf        0.300  At bound
Rotterdam   Amsterdam       0   0.000    inf       15.360  At bound
Rotterdam   Rotterdam       0   3.200    inf        0.000  Basic
Rotterdam   'Den Haag'      0  12.800    inf        0.000  Basic

The 1 scalar constraint:
Name                   ShadowPrice  Status
--------------------  -----------  ------
TotalCost_definition            0

The constraint "MeetDemand(j)" contains the following 3 rows:
         j   Lower   level  Upper  ShadowPrice  Status
----------  ------  ------  -----  -----------  ------
Amsterdam    5.880   5.880    inf        3.340  At bound
Rotterdam   12.400  12.400    inf       11.700  At bound
'Den Haag'  12.800  12.800    inf       12.700  At bound

The constraint "MeetSupply(i)" contains the following 2 rows:
         i  Lower   level  Upper  ShadowPrice  Status
----------  -----  ------  -----  -----------  ------
Amsterdam    -inf  15.080     16        0.000  Basic
Rotterdam    -inf  16.000     16       -9.700  At bound
```

Part VII

Advanced Language Components

Chapter 32

Units of Measurement

This chapter

This chapter describes how to incorporate dimensional analysis into an AIMMS application. As will be explained, you can define quantities and their corresponding units, and associate these units with identifiers in your model. AIMMS automatically checks for unit consistency in all the constraints and assignment statements. In addition, AIMMS allows you to specify unit conventions. With this facility it is possible for end-users around the world to select their preferred convention, and view the model data in the units associated with that convention.

32.1 Introduction

Units are common

Measurement plays a central role in observations of the real world. Most observed quantities are measured in some unit (e.g. dollar, hour, meter, etc.), and the magnitude of the unit influences the mental picture that you may have of an object (e.g. ounce, kilogram, ton, etc.). When you combine such objects in a numerical relationship, the corresponding units must be *commensurable*. Without such consistency, the mathematical relationships become meaningless.

Why units in models

There are several good reasons to track units throughout a model. The explicit mentioning of units can enhance the readability of a model, which is especially helpful when others read and/or maintain your model. Units provide the AIMMS compiler with additional checking power to find errors in model formulations. Finally, through the use of units you can let AIMMS perform the job of unit conversion and scaling.

Standard units

The model editor in AIMMS will give you access to a large number of quantities and units, and in particular to those of the International System of Units (referred to as SI from the French “Systeme Internationale”). The SI system is an improved metric system adopted by the Eleventh General Conference of Weights and Measures in 1960. The entire SI system of measurement is constructed from the atomic base units associated with the following nine basic quantities.

Quantity	Atomic Base Unit	Text
length	m	meter
mass	kg	kilogram
time	s	second
temperature	K	kelvin
amount of mass	mol	mole
electric current	A	ampere
luminous intensity	cd	candela
angle	rad	radian
solid angle	sr	steradian

Table 32.1: Basic SI quantities and their base units

Derived quantities and units

All quantities which are not one of the nine basic SI quantities are called *derived* quantities. Each such quantity has a derived base unit which can be expressed in terms of the atomic base units of the basic SI quantities. Optionally, a compound unit symbol can be associated with such a derived base unit, like the symbol `N` for the unit `kg*m/s^2`. The following table illustrates some of the more well-known derived quantities and their corresponding derived base units. Note that five of them have an associated compound unit symbol. Many other derived quantities are available in AIMMS.

Quantity	Derived Base Unit	Text
area	`m^2`	square meter
volume	`m^3`	cubic meter
force	`N = kg*m/s^2`	newton
pressure	`Pa = kg/m*s^2`	pascal
energy	`J = kg*m^2/s^2`	joule
power	`W = kg*m^2/s^3`	watt
charge	`C = A*s`	coulomb
density	`kg/m^3`	kilogram per cubic meter
velocity	`m/s`	meter per second
angular velocity	`rad/s`	radian per second

Table 32.2: Selected derived SI quantities and their base units

Related units

Aside from the base unit that must be associated with every quantity, it is also possible to specify a number of *related* units. Related units are those units that can be expressed in terms of their base unit by means of a linear relationship. A typical example is the unit `km` which is related to the base unit `m` by means of the linear relationship x `km = 1000*`x `m`. Similarly, the unit `degC` (degree Celsius) is related to the base unit `K` through the formula x `degC = (`x `+ 273.15) K`.

Standard unit prefix notation

Frequently, related units are a multiple of their own base unit, which is reflected through a prefix notation that indicates the level of scaling. Table 32.3 shows the standard SI prefix symbols and their corresponding scaling factor. Familiar examples are `kton`, `MHz`, `kJ`, etc. Note that any prefix can be applied to any base unit except the kilogram. The kilogram takes prefixes as if the base unit were the gram.

Factor	Name	Symbol	Factor	Name	Symbol
10^{1}	deca	da	10^{-1}	deci	d
10^{2}	hecto	h	10^{-2}	centi	c
10^{3}	kilo	k	10^{-3}	milli	m
10^{6}	mega	M	10^{-6}	micro	mu
10^{9}	giga	G	10^{-9}	nano	n
10^{12}	tera	T	10^{-12}	pico	p
10^{15}	peta	P	10^{-15}	femto	f
10^{18}	exa	E	10^{-18}	atto	a
10^{21}	zetta	Z	10^{-21}	zepto	z
10^{24}	yotta	Y	10^{-24}	yocto	y

Table 32.3: Prefixes of the International System

Flexible specification

To give you maximum freedom to choose your own quantities, units and naming conventions, AIMMS is not exclusively committed to any particular standard. However, you are encouraged to use the standard SI units and prefix symbols to make your model as readable and maintainable as possible.

Summary of terminology

Thus far you have encountered *basic* quantities (Table 32.1) and *derived* quantities (Table 32.2). Each quantity has a *base* unit. The base unit of a basic quantity is defined through a unit symbol, referred to as an *atomic* unit. All other base units are *derived* base units. Such units are defined through an expression in terms of other base units, which can eventually be translated into an expression of atomic base units. You have the option to associate a unit symbol with any derived base unit, which is referred to as a *compound* unit symbol. Whenever you have associated a unit symbol with the base unit of either a basic or derived quantity, you are also allowed to specify one or more *related* unit symbols by specifying the corresponding linear relationship.

32.2 The `Quantity` declaration

Declaration

In AIMMS, all units are uniquely coupled to declared quantities. For each declared `Quantity` you must specify an identifier together with one or more of its attributes listed in Table 32.4.

Attribute	Value-type	Mandatory
BaseUnit	[*unit-symbol*] [=] [*unit-expression*]	yes
Text	*string*	
Conversion	*unit-conversion-list*	
Comment	*comment string*	

Table 32.4: Quantity attributes

The BaseUnit attribute

You must always specify a base unit for each quantity that you declare. Its value is either

- an *atomic* unit symbol,
- a *unit expression*, or
- a *compound* unit symbol with unit expression.

A unit symbol can be any sequence of the characters a–z, the digits 0–9, and the symbols _, @, &, %, |, as well as a currency symbol not starting with a digit, or one of the special unit symbols 1 and -. The latter two special unit symbols allow you, for instance, to declare model identifiers without unit, or to express unitless numerical data in terms of percentages.

Currency symbols

AIMMS supports the currency symbols as defined by the Unicode committee, see http://unicode.org/charts/PDF/U20A0.pdf. These currency symbols include $, €, ç, ¢, £, and ¤.

Separate namespace

AIMMS stores unit symbols in namespaces separate but parallel to the identifier namespaces. Hence, you are free to choose unit symbols equal to the names of global identifiers within your model. Namespaces in AIMMS are discussed in full detail in Section 35.4.

Backward compatibility

AIMMS 3.8 and older use only a singleton unit namespace which was a potential cause of nameclashes when units with the same name are declared from quantities or unit parameters declared in different namespaces. In order to obtain the old behaviour one can make sure that all units are declared within the global namespace or set the option singleton_unit_namespace to on. This option can be found in the backward compatibility category.

Example

The following example illustrates the three types of base units.

```
Quantity Length {
    BaseUnit   : {
        m                    ! atomic unit
    }
}
```

```
Quantity Time {
    BaseUnit   : {
        s                   ! atomic unit
    }
}
Quantity Velocity {
    BaseUnit   : {
        m/s                 ! unit expression
    }
}
Quantity Frequency {
    BaseUnit   : {
        Hz = 1/s            ! compound unit symbol with unit expression
    }
}
```

The atomic unit symbols `m` and `s` are the base units for the quantities `Length` and `Time`. The unit expression `m/s` is the base unit for the quantity `Velocity`. The compound unit symbol `Hz`, defined by the unit expression `1/s`, is the base unit of the quantity `Frequency`.

Derived can be used as basic

The previous example strictly adheres to the SI standards, and, for example, defines the base unit of the derived quantity `Frequency` in terms of the base unit of `Time`. In general, this is not necessary. If `Time` is not used anywhere else in your model, you can just provide the base unit `Hz` for `Frequency` without providing its translation in SI base units. `Frequency` then becomes a basic quantity, and `Hz` becomes an atomic base unit.

Unit expressions

The unit expressions that you can enter in the `BaseUnit` attribute can only consist of

- unit symbols (base and/or related units),
- constant factors,
- the two operators "`*`" and "`/`",
- parentheses, and
- the power operator "`^`" with integer exponent.

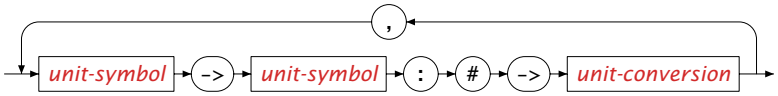
The common precedence order of the operators "`*`", "`/`" and "`^`" is as described in Section 6.3. Unit expressions are discussed in full detail in Section 32.6.

The `Conversion` attribute

With the `Conversion` attribute you can declare and define one or more related unit symbols by specifying the (linear) transformation to the associated base unit. The conversion syntax is as follows.

Syntax

unit-conversion-list :



Unit conversion explained

A unit conversion must be defined using a linear expression of the form $(\# \cdot a + b)$ where # is a special token, and the operator $\cdot$ stands for either multiplication or division. The coefficients a and b can be either numerical constants or references to scalar parameters. An example in which the use of scalar parameters is particularly convenient is the conversion between currencies parameterized by a varying exchange rate.

Example

```
Quantity Length {
    BaseUnit    : m;
    Conversions : {
        km   -> m    :  # -> # * 1000,
        mile -> m    :  # -> # * 1609
    }
}
Quantity Temperature {
    BaseUnit    : degC;
    Conversions : degC -> degF :  # -> # * 1.8 + 32;
}
Quantity Energy {
    BaseUnit    : J = kg * m^2 / s^2;
    Conversions : {
        kJ  -> J     :  # -> # * 1000 ,
        MJ  -> J     :  # -> # * 1.0e6,
        kWh -> J     :  # -> # * 3.6e6
    }
}
Quantity Currency {
    BaseUnit    : US$;
    Conversion  : {
        DM  -> US$   : # -> # * ExchangeRate('DM') ,
        DFl -> US$   : # -> # * ExchangeRate('DFl')
    }
}
Quantity Unitless {
    BaseUnit    : 1;
    Conversions : % -> 1      : # -> # / 100;
}
```

32.3 Associating units with model identifiers

The `Unit` attribute

To associate units with scalar or multi-dimensional identifiers in your model, you can specify a unit definition for such identifiers through the `Unit` attribute. The `Unit` attribute is only supported for the following identifier types:

- parameters,
- variables,
- constraints,
- arcs,
- nodes,
- function and procedure arguments, and
- internal and external functions.

Visibility of the Unit attribute

Within the AIMMS **Model Explorer**, the Unit attribute is only visible in the attribute forms of the identifier types listed above if your model already contains the declarations of one or more quantities. If you only want to use the Unit attribute to specify a scale factor for an identifier (see below), you can make the Unit attribute visible in all attribute forms by adding a *unitless* quantity to your model (i.e. a quantity with base unit 1).

Unit attribute value

In its simplest form, the unit definition of a parameter, variable or constraint is just a reference to a base or compound unit symbol. In general, it can be a unit expression based on the same syntax as described previously for specifying a derived unit expression in a Quantity declaration. The complete syntax of unit expressions is discussed in Section 32.6.

Example

The declaration

```
Variable VelocityOfItem
    IndexDomain  : i;
    Unit         : km/h;
}
```

introduces a variable VelocityOfItem(i) with a corresponding unit km/h. This declaration could also have been written as

```
Variable VelocityOfItem {
    IndexDomain  : i;
    Unit         : 1000*m/h;
}
```

which contains an explicit scale factor of 1000, instead of using the derived unit symbol km.

Units also for scaling

When you do not use unit symbols, you can still use the Unit attribute to indicate the appropriate scale factor to be used for an identifier. These scale factors, whether or not in the presence of unit symbols, will be used by AIMMS to scale the corresponding data during various computations, as explained in Section 32.5.

Use of units

By specifying units for some or all the identifiers in your model, AIMMS will perform the following unit-related tasks for you:

- automatic checking of the statements in your model for unit consistency (see Section 32.4),
- automatic scaling of identifiers in assignments, DISPLAY and READ/WRITE statements (see Section 32.5), and
- automatic conversion of arguments (and result value) of external procedures and functions (see Section 32.5), and
- automatic scaling of the variables and constraints in a mathematical program (see Section 32.5.1).

The .Unit suffix

For all identifier types for which you can specify a Unit attribute, there is also an associated .Unit suffix. The value of the .Unit suffix is a unit expression that equals the unit specified within the Unit attribute of the identifier at hand.

Use of the .Unit suffix

The .Unit suffix is most commonly used in the following situations:

- when generating reports by means of the PUT and DISPLAY statements (see Sections 31.2 and 31.3, respectively),
- when displaying units in strings generated by the %u conversion specifier of the FormatString function (see Section 5.3.2), and
- when performing sensitivity analysis of mathematical programs in the presence of variables and constraints which have a non-empty Unit attribute (see Section 32.5.1).

Indices not always required

If you want to reference the .Unit suffix of a multidimensional identifier, it is not always necessary to use the corresponding indices of the identifier in its .Unit suffix reference. The use of indices is only necessary if the Unit attribute actively depends on the indices, for instance, because it

- contains a multidimensional scale factor, or
- refers to a multidimensional unit parameter (see also Section 32.9).

In all other cases, a reference to just the identifier name is sufficient.

Example

Consider the declaration of the variable VelocityOfItem(i) above. Its UNIT attribute is the constant unit km/h, whence it can be obtain through the (scalar) reference

```
VelocityOfItem.Unit
```

Unit-valued parameters are permitted

When the Unit attribute of an identifier contains references to *unit-valued* parameters (see Section 32.9), such references will be evaluated, within the context of the .Unit suffix, to their corresponding unit expressions. Thus, the .Unit suffix will always result in a unit expression containing only unit symbols declared in one or more Quantity declarations.

32.4 Unit analysis

Unit consistency

By associating a unit with every relevant identifier in your model, you enable AIMMS to automatically verify whether all terms in the assignments and constraints of your model are unit consistent. When AIMMS detects unit inconsistencies, this may help you to solve conceptual problems in your model, which could otherwise have remained undetected for a long time.

... always in atomic units

With every derived unit or compound unit symbol, it is possible to associate a unique unit expression consisting of a constant scale factor and atomic units only. *All assignments and definitions in* AIMMS *are interpreted as formulas expressed in terms of these atomic unit expressions, and unit consistency checking is based on this interpretation.* While ignoring the constant scale factors, AIMMS will verify that the atomic unit expression for every term in either an assignment statement or a constraint is identical. If the resulting unit check identifies an inconsistency, an error or warning will be generated.

Example

Consider the identifiers a, b, and c having units [m], [km], and [10*m] respectively, all with [m] as their corresponding associated atomic unit expression, and scale factors 1, 1000 and 10, respectively. Then the assignment

```
c := a + b ;
```

is *unit consistent*, because all terms share the same atomic unit expression [m].

Constant expressions

If an expression on the right-hand side of an assignment consists of a constant scalar term or a data expression (preceded by the keyword DATA), AIMMS will assume by default that such expressions have the same unit as the identifier on the left-hand side. If the intended unit of the right-hand side is different than the declared unit of the identifier on the left, you should explicitly specify the appropriate unit for this term, by locally overriding the unit as explained in Section 32.7.

Constant terms in expressions

On the other hand, if a non-constant expression contains a constant term, then AIMMS will make no assumption about the intended unit of the constant term. In fact, it is considered unitless. If a unit inconsistency occurs for that reason, you should explicitly add a unit to the constant term to resolve the inconsistency, as explained in Section 32.7.

Example

Given parameters a ([m] and b ([km]), as well as a 1-dimensional parameter d(i) with associated unit [m], the following assignments illustrate the interpretation of constant numbers by AIMMS.

```
a    := 10;                   ! OK:    constant number 10 interpreted as [m]
a    := 10 [km];              ! OK:    constant number 10 interpreted as [km]
d(i) := DATA { 1: 10, 2: 20 }; ! OK:    all data interpreted as [m]
a    := 10*b;                 ! OK:    constant number 10 considered unitless
a    := b + 10;               ! ERROR: unit inconsistency, constant term 10 unitless
a    := b + 10 [km];          ! OK:    unit inconsistency resolved
```

Automatic unit checking on or off

By default, the global AIMMS option to perform automatic unit analysis is on and inconsistencies are detected. AIMMS will produce either warning messages or error messages (the former is the default). You can find the full details on all unit-related options in the help file that comes with your AIMMS system.

Automatic scale consistency

The assignment `c := a + b` of the first example in this section is unit consistent, but it does not appear to be *scale consistent* since the units of `a`, `b` and `c` have different scales. In AIMMS, however, a unit consistent assignment is automatically scale consistent, because AIMMS translates and stores all data in terms of the underlying atomic unit expression. In the example, this implies that the use of the values of `a`, `b`, and `c` as well as the assignment are in the atomic unit `[m]`. Consequently, AIMMS can now directly execute the assignment, and the scale consistency is automatically ensured. Of course, any *display* of values of `a`, `b` and `c` will be again in terms of the units associated with these identifiers.

Advanced example...

This example illustrates a number of identifiers with compound unit definitions. It is based on the SI units for weight, velocity and energy, and uses the derived units `ton`, `km`, `h` and `MJ`.

```
Variable WeightOfItem {
    IndexDomain  : i;
    Unit         : ton;
}
Variable VelocityOfItem {
    IndexDomain  : i;
    Unit         : Velocity: km/h;
}
Variable KineticEnergyOfItem {
    IndexDomain  : i;
    Unit         : MJ;
    Definition   : 1/2 * WeightofItem(i) * VelocityOfItem(i)^2;
}
```

Any display of these variables will be in terms of `ton`, `km/h` and `MJ`, respectively, but internally AIMMS uses the units `kg`, `m/s` and `kg*m^2/s^2` for storage. The latter represent the corresponding unique atomic unit expressions associated with weight, velocity and energy.

... is unit consistent

As a consequence of specifying units, there will be an automatic consistency check on the defined variable `KineticEnergyOfItem(i)`. AIMMS interprets the definition of `KineticEnergyOfItem(i)` as a formula expressed in terms of the atomic units. The relevant unit components are:

- `[ton ] = 10^3    * [kg        ]`,
- `[km/h] = (1/3.6) * [m/s       ]`, and
- `[MJ  ] = 10^6    * [kg*m^2/s^2]`.

The definition of `KineticEnergyOfItem(i)` as expressed in terms of atomic units is `kg*(m/s)^2`, while its own unit in terms of atomic units is `kg*m^2/s^2`. These two unit expressions are consistent.

Beware of non-absolute units

If the unit conversion between a derived unit and its corresponding atomic unit not only consists of a scale factor, but also contains a constant term, such a derived unit is referred to as a *non-absolute* unit. If an arithmetic expression in your model refers to identifiers or constants expressed in a non-absolute unit, you should pay special attention to make sure that the result of the computation is what you intended. The following example makes the point.

Example

Consider the following quantity declaration.

```
Quantity Temperature {
    BaseUnit     : K;
    Conversions  : degC -> K :  # -> # + 273.15;
}
```

Given this declaration, what is the result of the assignment

```
x := 1 [degC] + 2 [degC];
```

where x is a scalar parameter with unit degC? Following the rules explained above—AIMMS stores all data and performs all computations in terms of atomic units— AIMMS performs the following computation internally

```
x := 274.15 [K] + 275.15 [K];
```

resulting in an assignment to x of 549.3 [K] = 276.15 [degC], which is probably not the intended answer. The key observation is that in an addition only one of the operands should be expressed in a non-absolute unit. Similarly, in a multiplication or division probably none of the operands should be expressed in a non-absolute unit. The mistake in the above assignment is that the second argument in fact should be a temperature difference (e.g. between 3 [degC] and 1 [degC]), which precisely yields an expression in terms of the corresponding absolute unit K:

```
x := 1 [degC] + (3 [degC] - 1 [degC]);     ! equals 274.15 [K] + 2 [K] = 3 [degC]
```

Using temperature differences is more common in assignments to identifiers like LengthIncreasePerDegC (expressed in [m/degC]), which probably takes the form of a *difference quotient*, as illustrated below.

```
LengthIncreasePerDegC := (Length1 - Length0) / (Temperature1 - Temperature0);
```

Units and intrinsic functions

When you use an intrinsic AIMMS function (see Section 6.1.4) inside an expression in your model, the unit associated with the corresponding function call will in general depend on its arguments. The unit relationship between the arguments and the result of the function falls into one of the following function categories.

- *Unitless* functions, for which both the arguments and the result are dimensionless. Examples are: exp, log, log10, errorf, atan, cos, sin, tan, degrees, radians, atanh, cosh, sinh, tanh, and the exponential operator with a non-constant exponent.

- *Transparent* functions that do not alter units. Examples are: `abs`, `max`, `min`, `mod`, `ceil`, `floor`, `precision`, `round`, and `trunc`.
- *Conversion* functions that convert units in a predictable way. Examples are: `sqr`, `sqrt`, and the exponential operator with a constant integer exponent.

Explicit units in expressions

In some exceptional cases, one or more terms in an expression may not be unit consistent with the other terms in the expression. To restore unit consistency, AIMMS allows you to explicitly specify a unit for the inconsistent term(s) as an emergency measure. The syntax for such unit overrides is explained in Section 32.7. You should make sure, however, that these explicit unit overrides do not affect the scale consistency of the expression (see Section 32.7).

32.4.1 Unit analysis of procedures and functions

Unit analysis of procedures and functions

Once you have associated units of measurement with the global identifiers in your model, you will also need to associate units of measurement with the arguments, local identifiers and result values of procedures and functions. When you do so, you enable AIMMS to perform the common unit analysis on the statements in the bodies of all internal procedures and functions. For external procedures and functions, AIMMS cannot perform a unit analysis on the function and procedure bodies, but will use the assigned units for scaling purposes as explained in Section 32.5.

Two procedure types

In general, one can distinguish two types of procedures and functions, namely

- procedures and functions of a very specific nature, whose arguments and result values have associated units of measurement that are constant and known a priori, and
- procedures and functions of a very general nature, whose arguments and result values can have any associated unit of measurement.

An example of the latter type is a function with a single one-dimensional argument to compute the average of all values contained in its argument. For such a function, the specific units associated with the argument and the result values are not known a priori, but it is known that they must be equal.

Express units in local unit parameters

To let you declare procedure and functions of the second type, AIMMS allows you to express the units of measurement of its arguments and the result values in terms of unit parameters (see also Section 32.9) declared locally within the procedure or function. At runtime, AIMMS will dynamically determine the value of the unit parameter, based on the actual arguments passed to the procedure or function. In addition, AIMMS will verify that the unit of a function value is commensurate with the remainder of the statement or expression from which it was called.

Example

The function MyAverage in this example computes the average of a general one-dimensional identifier. It combines AIMMS' ability to define arguments over local sets (see Section 10.1), with a unit expressed in term of a local unit parameter. Its declaration is given by

```
Function MyAverage {
    Arguments : (Ident);
    Unit      : LocalUnit;
    Body      : {
        MyAverage := sum(i, Ident(i)) / Card(LocalSet)
    }
}
```

The single argument Ident(i) of the function MyAverage is defined by

```
Parameter Ident {
    IndexDomain  : i;
    Unit         : LocalUnit;
}
Set LocalSet {
    Index        : i;
}
UnitParameter LocalUnit;
```

Note that Ident(i) is defined over a local set LocalSet and that its unit is expressed in terms of a local unit parameter LocalUnit, both of which are determined at runtime. Because the unit of the function MyAverage itself is also equal to LocalUnit, the assignment in the body of MyAverage is unit consistent.

32.5 Unit-based scaling

Scaled versus unscaled values

With each identifier for which you have specified a Unit attribute, AIMMS associates two values:

- the *scaled* value (i.e. expressed in terms of the unit specified), and
- the *unscaled* value (i.e. expressed in terms of the associated atomic unit expression).

The transformation between scaled and nonscaled values is completely determined by the product of explicit and implicit scale factors associated with the various quantity and unit definitions.

When used

As mentioned in Section 32.4, AIMMS uses internally unscaled values for all storage and arithmetic computations. This guarantees automatic scale consistency. However, for external use, scaled values are more natural when exchanging data with components outside the AIMMS execution system. Specifically, AIMMS uses scaled values when

- displaying the data of an identifier in the (end-)user interface,
- exchanging data for a particular identifier with files and databases using the READ and WRITE statements,

- passing arguments to external procedures and functions,
- storing the result value(s) of an external function, and
- communicating the variables and constraints of a mathematical program to a solver.

Units in displays

When displaying data in either the graphical user interface or in PUT and DISPLAY statements, AIMMS will transfer data using the scaled unit specified in the definition of the identifier. For example, if you have specified kton as the unit attribute of an identifier while the underlying atomic unit is kg, AIMMS will still display the identifier values in kton.

. . . and data entry

Similarly, when reading data from or writing data to scalar numerical constants, lists, tables, composite tables (either graphical or in data files), or tables (in databases) using the READ and WRITE statements, AIMMS assumes that this data is provided in the (scaled) units that you have specified in the identifier declarations in your model, and will transform all data to the corresponding unscaled values for internal storage.

Override default scaling

You can override the default scaling based on the content of the Unit attribute either locally within the graphical end-user interface or model source, or globally using Conventions. Local and global overrides are discussed in complete detail in Sections 32.7 and 32.8.

32.5.1 Unit-based scaling of mathematical programs

Automatic scaling for solvers

During communications with a solver, AIMMS will scale all variables and constraints (including variable definitions) in accordance with the scale factor associated with the Unit attribute in their declaration. This choice is based on the assumption that the specified units reflect the expected order of magnitude of the numbers associated with the variables, parameters and constraints, and that these numbers will neither be very large nor very small. As a result, the values of all rows and columns in the generated mathematical program are expected to be of the same, reasonable, order of magnitude. Especially nonlinear solvers may greatly benefit from this choice.

Main example revisited

In the main example of Section 32.4, the scale factors are 10^3 for the identifier WeightOfItem(i), $1/3.6$ for VelocityOfItem(i), and 10^6 for KineticEnergyOfItem(i). The entire constraint associated with the defined variable is then scaled according to the scale factor of the unit of the definition variable KineticEnergyOfItem(i), MJ. This corresponds with dividing the left- and right-hand side of the constraint by 10^6. Thus, the resulting expression communicated to the solver by AIMMS will be:

```
KineticEnergyOfItemColumn(i) =
    1/2 * (1/10^3) * WeightofItemColumn(i) * ((1/3.6) * VelocityOfItemColumn(i))^2 ;
```

Notice that each variable shown in this expression has a suffix "Column" to indicate that it corresponds to a column in the matrix underlying the mathematical program.

Units of reduced cost and shadow price

Some care is needed when you have requested sensitivity information associated with a mathematical program, such as the reduced costs of variables and shadow prices of constraints. The basic rules with respect to retrieving sensitivity information are as follows:

- All sensitivity suffices in AIMMS, such as the .ReducedCost and .ShadowPrice suffix, are unitless.
- All sensitivity suffices hold the exact numerical value as computed by the solver, i.e. expressed with respect to the scaled values that are communicated to the solver by AIMMS.

Motivating the choice of unitless

The reason for not associating units with the sensitivity suffices is that a single variable or constraint may be used in multiple mathematical programs, each with its own objective. As each objective may have a different associated unit, and the reduced costs and shadow prices express properties of a variable or constraint with respect to the objective, it is inherently impossible to associate a single unit with the .ReducedCost and .ShadowPrice suffices.

Unit- and scale consistent sensitivity data

You may encounter scaling problems when you want to perform direct computations with the sensitivity suffices of variables and constraints. Using the .Unit suffix and AIMMS' capabilities to override units of subexpressions (see Sections 32.6 and 32.7), however, it is easy to formulate expressions that

- result in the correct unscaled numerical values that can be used directly in AIMMS computations, and
- have an associated unit that is consistent with their interpretation.

Example with unit overrides

Assuming that ExampleVariable and ExampleConstraint are part of a mathematical program, with ObjectiveVariable as its objective function, one can obtain the correct values by locally overriding the units of the .ReducedCost and .ShadowPrice suffices through the expressions:

```
(ExampleVariable.ReducedCost  ) [ObjectiveVariable.Unit / ExampleVariable.Unit  ]
(ExampleConstraint.ShadowPrice) [ObjectiveVariable.Unit / ExampleConstraint.Unit]
```

Example with unit functions

Alternatively, you can use the function EvaluateUnit (see Section 32.6.2) to obtain the same result

```
ExampleVariable.ReducedCost *
        EvaluateUnit( ObjectiveVariable.Unit / ExampleVariable.Unit )
ExampleConstraint.ShadowPrice *
        EvaluateUnit( ObjectiveVariable.Unit / ExampleConstraint.Unit )
```

Introducing new parameters

If you need to perform multiple computations with these expressions, or want to display them in the graphical end-user interface, you are advised to assign these expressions to additional parameters in your model with the appropriate associated units.

Example with convention

When you have used a Convention to override the default scaling during the SOLVE statement, the expressions above should be augmented by applying the functions ConvertUnit and EvaluateUnit (see Section 32.6.1):

```
ExampleVariable.ReducedCost *
        EvaluateUnit( ConvertUnit(ObjectiveVariable.Unit, ConventionUsed) /
                      ConvertUnit(ExampleVariable.Unit, ConventionUsed)    )
ExampleConstraint.ShadowPrice *
        EvaluateUnit( ConvertUnit(ObjectiveVariable.Unit, ConventionUsed) /
                      ConvertUnit(ExampleConstraint.Unit, ConventionUsed)  )
```

This will result in a scaling factor that is consistent with the variable and constraint scaling convention passed to the solver. You cannot obtain the same result by locally overriding the units of the .ReducedCost and .ShadowPrice suffices, as unit local overrides only accept simple unit expressions (see Section 32.6).

Use of unit parameters

If your model contains multiple computations concerning the .ReducedCost and .ShadowPrice suffices, each with identical scale factors, you may consider assigning the unit expressions required for scaling these suffices to unit parameters (see Section 32.9). You can then directly use such unit parameters in a local unit override, rather than having to repeat possibly complex unit expressions time and again. For instance, if ScaledUnit is a unit parameter defined by

```
ScaledUnit := ConvertUnit(ObjectiveVariable.Unit, ConventionUsed) /
              ConvertUnit(ExampleVariable.Unit, ConventionUsed) ;
```

then the correctly scaled expression for the reduced cost of ExampleVariable can be simplified to

```
(ExampleVariable.ReducedCost) [ScaledUnit]
```

You can use a local override, because a reference to a scalar unit parameter again forms a valid simple unit expression (see Section 32.6).

32.6 Unit expressions

Unit expressions

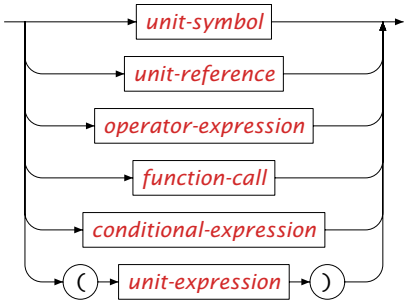
Unit expressions can be used at various places in an AIMMS model, such as:

- the BaseUnit attribute of a Quantity declaration (defined in Section 32.2),
- in a local unit override of a numerical (sub-)expression (discussed in Section 32.7)
- in a convention list of the PerUnit, PerQuantity or PerIdentifier attributes of a Convention (see also Section 32.8), or

- on the right hand side of an assignment to a unit parameter (see Section 32.9).

The syntax of a unit expression is straightforward, and given below.

unit-expression : *Syntax*



Unit symbols and references

The simplest form of unit expression is just a unit symbol, as defined in either the `BaseUnit` or the `Conversion` attribute of a `Quantity` declaration. A reference to either a (scalar or indexed) unit parameter (see Section 32.9) or to the `.Unit` suffix of any identifier with an associated unit (see Section 32.3), is a second form of unit expression.

Unit operators and functions

More complex unit expressions can be obtained by applying the binary unit operators *, / and ^, with the usual left-to-right evaluation order. The following rules apply:

- the operand on the right of the * operator must be a unit expression, while the operand on the left can either be a unit expression or a numerical expression (expressing a numeric scale factor),
- both operands of the / operator must be unit expressions, and
- the operand on the left of the ^ operator must be a unit expression, while the exponent operand must be an integer numerical expression.

In addition, AIMMS supports a number of unit functions, which can create new unit values or construct associated unit values from a given unit expression (see Section 32.6.1).

Three types of unit expressions

However, AIMMS requires that any unit expressions uniquely falls into one of the three categories

- unit constant,
- simple unit expression, or
- computed unit expression.

Unit constants are unit expressions which consist solely of unit symbols, scalar constants and the three unit operators *, / and ^. Unit constants can be used in

Unit constants

- the BaseUnit attribute of a Quantity,
- the lists associated with a Convention, and
- the unit-valued function Unit.

In addition, unit constants can be

- displayed and entered via the AIMMS graphical user interface,
- assigned to unit parameters through data statements (see Chapter 28), and
- exchanged with external data sources via the READ and WRITE statements (see Chapter 26).

Simple unit expressions are an extension of unit constants. They are unit expressions which consist solely of unit symbols, unit references without indexing, scalar constants and the three unit operators *, / and ^. Simple unit expressions can be used in

Simple unit expressions

- local unit overrides, and
- assignments to unit parameters.

Computed unit expression can use the full range of unit expressions, with the exception of unit constants. If you want to refer to unit constants within the context of a computed unit expression, you must embed it within a call to the function Unit, discussed in the next section. Computed unit expressions can be used

Computed unit expressions

- in assignments to unit parameters, and
- as an argument of the functions ConvertUnit, AtomicUnit and EvaluateUnit (see Sections 32.6.1 and 32.6.2).

32.6.1 Unit-valued functions

AIMMS supports the following unit-valued functions:

Unit-valued functions

- Unit(*unit-constant*)
- StringToUnit(*unit-string*)
- AtomicUnit(*unit-expr*)
- ConvertUnit(*unit-expr*, *convention*)

The function Unit

The function Unit simply returns its argument, which must be a unit constant. The function Unit is available to allow the usage of unit constants within computed unit expressions (as discussed in the previous section).

The function StringToUnit

The function StringToUnit converts a string, which represents a unit expression, to the corresponding unit value. You can use this function, for instance, after reading external string data that needs to be converted to real unit values for further use in your model.

The function AtomicUnit

With the function AtomicUnit you can retrieve the atomic unit expression corresponding to the unit expression passed as the argument to the function. Thus, the unit expression

```
AnIdentifier.Unit / AtomicUnit(AnIdentifier.Unit)
```

will result in a (unitless) unit value that exactly represents the scale factor between the unit of an identifier and its associated atomic unit expression. You can obtain the corresponding numerical value, to be used in numerical expressions, by applying the function EvaluateUnit discussed in the next section.

The function ConvertUnit

The function ConvertUnit returns the unit value corresponding to the unit expression of the first argument, but taking into consideration the convention specified in the second argument. If the first argument contains a reference to a .Unit suffix, AIMMS will apply the full range of conversions including those specified in the PerIdentifier attribute of the convention.

Examples

The expression

```
ConvertUnit(AnIdentifier.Unit, ConventionUsed)
```

returns the associated unit of the identifier AnIdentifier as if the convention ConventionUsed were active. A further example of the use of the function ConvertUnit is given in Section 32.5.1.

32.6.2 Converting unit expressions to numerical expressions

Numeric value of a unit expression

Although numerical values and unit values are two very distinct data types in AIMMS, the distinction between the two in real life applications is not always as strict. For instance, in the previous section the computation of the ratio between a unit and its associated atomic unit expression returned a unit value, which represents nothing more than a (unitless) scale factor. In practice, however, it is the numeric scale factor value that is of interest, and can be used in numerical computations.

The function EvaluateUnit

Using the function EvaluateUnit you can compute the numerical value associated with a computed unit expression. Its syntax is:

- EvaluateUnit(*computed-unit-expression*)

The numeric function value precisely corresponds to one unit of the specified computed unit expression, measured in the evaluated unit of its argument.

Example

The following assignment to the scalar parameter ScaleFactor computes the (unitless) scale factor between the unit of an identifier and its associated atomic unit expression.

```
ScaleFactor := EvaluateUnit( AnIdentifier.Unit / AtomicUnit(AnIdentifier.Unit) );
```

Extension of local overrides

As you will see in the next section, the function EvaluateUnit offers extension the local unit override capability. The argument of EvaluateUnit can be a computed unit expression (see Section 32.6), whereas local unit overrides can only accept simple unit expressions.

32.7 Locally overriding units

Locally overriding units

In some rare occasions the unit specified in the declaration of a particular identifier does not necessarily have to match with the unit of the data for that identifier. In that case, AIMMS allows you just to override the unit of a particular expression locally. Such a local unit override of an expression always takes the simple form

(*expression*) [*simple-unit-expression*]

where *expression* is some AIMMS expression, and *simple-unit-expression* is a simple unit expression as explained in Section 32.6. If *expression* solely consists of a numeric constant, AIMMS allows you to omit the parentheses around it.

Where to use

You can use local unit overrides in a variety of data I/O related situations.

- In a WRITE, DISPLAY or PUT statement, you can use a local unit override to specify the particular unit in which data must be written to a file, database table or window.
- In the FormatString function, you can use a local unit override to specify the unit in which a numeric argument corresponding to a %n format specifier must be formatted.
- On the left side of a data assignment, in the header of a composite table, or in a READ statement, you can use a local unit override to specify the unit in which the supplied data to be provided.

Commensurate requirement

In all these data I/O statements and expressions, AIMMS requires that the unit provided in the override is commensurate with the original unit that can be associated with the expression.

Example

Given the declarations of the examples in the Section 32.4, the following data I/O statements locally override the default unit [km/h] of the identifier VelocityOfItem with the commensurate unit [mph].

- Override per identifier:

```
(VelocityOfItem) [mph] := DATA { car: 55, truck: 45 };
read (VelocityOfItem) [mph] from table VelocityTable;
display (VelocityOfItem) [mph];
```

- Override per individual entry:

```
put (VelocityOfItem('car')) [mph];
StringVal := FormatString("Speed in [mph]: %n", (VelocityOfItem('car')) [mph]);
```

Recall that parentheses are always required when you want to override the default unit in expressions and statements, unless the overridden expression is a simple numeric constant.

Override for consistency

In addition to overriding units during a data exchange, you can also override the unit of a (sub)expression in an assignment with the purpose of enforcing unit consistency of all terms in the assignment. This is especially useful when there are numeric constants inside your expressions. AIMMS will add the appropriate scale factor if the specified unit override does not match with the corresponding atomic unit expression.

Examples

The following examples illustrate unit overrides with the purpose of enforcing unit consistency.

- Consider the assignment

```
SoundIntensity := (10 * log10( SoundLevel / ReferenceLevel )) [dB];
```

  If SoundIntensity has an associated unit of [dB], the right hand side of the assignment, which by itself is unitless, must be locally overridden to make the entire assignment unit consistent.
- Consider the assignment

```
a := b + 10 [km];
```

  where both a and b are measured in terms of length. As discussed in Section 32.4, AIMMS will make no assumption about the unit associated with the numerical constant 10 in the expression on the right-hand side of the assignment. In order to make the assignment unit consistent, an explicit unit override of the constant term is required. If the associated base unit is [m], AIMMS will automatically add a scale factor of 1000, whence the assignment will numerically evaluate to a := b + 10*1000.

Caution is needed

If you explicitly associate a unit with an expression which already contains one or more identifiers *with* associated units, the numerical result can be unexpected. This is due to fact that AIMMS, during expression evaluation, uses the unscaled numerical values with respect to the associated atomic units of each identifier. To illustrate, reconsider the assignment

```
a := (b * c) [km];
```

but now assume that the identifiers a, b, and c have units [km], [km], and [10*m]. If the values of b and c are 1 [km](=1000 [m]) and 50 [10*m](=500 [m]), respectively, the numerical result of a after the assignment will amount to (500 * 1000)*1000 [m]= 500000 [km], which may not be the result that you intended.

32.8 Globally overriding units through Conventions

Unit conventions

In addition to locally overriding the unit definition of an identifier in a particular statement, you can also *globally* override the default format for data exchange using READ and WRITE, DISPLAY and SOLVE statements by selecting an appropriate *unit convention*. A convention offers a global medium to specify alternative (scaled) units for multiple quantities, units, and identifiers. In addition, one can specify alternative representations for a calendar in a convention.

Effect of conventions

Once you have selected a convention, AIMMS will interpret all data transfer with an external component according to the units that are specified in the convention. When no convention has been selected for a particular external component, AIMMS will use the default convention, i.e. apply the unit as specified in the declaration of an identifier. For a compound quantity not present in a convention, AIMMS will apply the convention to all composing atomic units used in the compound quantity.

Convention attributes

Conventions must be declared before their use. The list of attributes of a Convention declaration are described in Table 32.5.

Attribute	Value-type	See also page
Text	*string*	
Comment	*comment string*	
PerIdentifier	*convention-list/reference*	
PerQuantity	*convention-list*	
PerUnit	*convention-list*	
TimeslotFormat	*timeslot-format-list*	570

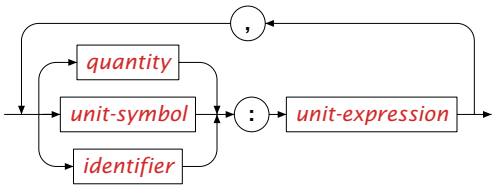
Table 32.5: Simple Convention attributes

Convention list

A convention list is a simple list associating single quantities, units and identifiers with a particular (scaled) unit expression. The specified unit expressions must be consistent with the base unit of the quantity, the specified unit, or the identifier unit, respectively.

Syntax

convention-list :



Customizable conventions

In addition to a fixed convention list, the `PerIdentifier` attribute also accepts a reference to a unit-valued parameter defined over the set `AllIdentifiers` or a subset thereof. In that case, the convention will dynamically construct a convention list based on the contents of the unit-valued parameter.

Example

The following declaration illustrates the use of a `Convention` to define the more common units in the Anglo-American unit system at the quantity level, the unit level and the identifier level.

```
Convention AngloAmericanUnits {
    PerIdentifier  : {
        GasolinePurchase : gallon,
        PersonalHeight   : feet
    }
    PerQuantity    : {
        Velocity         : mph,
        Temperature      : degF,
        Length           : mile
    }
    PerUnit        : {
        cm               : inch,
        m                : yard,
        km               : mile
    }
}
```

Customizable example

Assuming that `IdentifierUnits` is a unit-valued parameter defined over `AllIdentifiers`, the following `Convention` declaration illustrates a convention that can be customized at runtime by modifying the contents of the unit parameter `IdentifierUnits`.

```
Convention CustomizableConvention {
    PerIdentifier  : IdentifierUnits;
}
```

Application order

For a particular identifier, AIMMS will select a unit from a convention in the following order.

- If a unit has been specified for the identifier, AIMMS will use it.
- If the identifier can be associated with a specific quantity in the convention, AIMMS will use the unit specified for that quantity.
- In all other cases AIMMS will apply the convention to an atomic unit directly, or to all composing atomic units used in a compound unit.

Timeslot format list

In addition to globally overriding units, `Conventions` can also be used, through the `TimeslotFormat` attribute, to override the time slot format of calendars. You may need to specify alternative time slot formats, for instance, when you are reading data from an external database or file, in which all dates are not specified in the same time zone as the one your model assumes. The `TimeslotFormat` attribute of a `Convention` is discussed in full detail in Section 33.10.

The `Convention` attribute

You can declare more than one convention in your model. A `Convention` attribute can be specified for the following node types in the model tree, which all correspond to an external component:

- the main model (used for the end-user interface or as default for all other external components),
- a mathematical program,
- a file (also when used to refer to a DLL containing a library of external procedures and functions used by AIMMS), and
- a database table or procedure.

The value of the `Convention` attribute can be a specific convention declared in your model, or a string or element parameter referring to a particular unit convention.

Convention semantics

For data exchange with all aforementioned external components AIMMS will select a unit convention in the following order.

- If an external component has a nonempty `Convention` attribute, AIMMS will use that convention.
- For display in the user interface, or for data exchange with external components without a `Convention` attribute, AIMMS will use the convention specified for the main model (see also Section 35.2), if present.
- If the main model and external components have no `Convention` attribute, AIMMS will use the default convention, i.e. use the unit as specified in the declaration of each identifier.

Example

The following declaration of a File identifier shows the use of the Convention attribute. All the output to the file ResultFile will be displayed in Anglo-American units.

```
File ResultFile {
    Name       : "Output\\result.dat";
    Convention : AngloAmericanUnits;
}
```

32.9 Unit-valued parameters

Parametrized units

In some cases not all entries of an indexed identifier have the same associated unit. An example is the diet model where the nutritive value of each nutrient for a single serving of a particular food type is measured in a different unit.

Unit-valued parameters

In order to deal with such situations, AIMMS allows the declaration of (indexed) *unit-valued* parameters which you can use in the unit definition of the other parameters and variables in your model. In the model tree, unit-valued parameters are available as a special type of parameter declaration, with attributes as given in Table 32.6.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	
Quantity	*quantity*	
Default	*unit-expression*	
Property	NoSave	45
Text	*string*	19
Comment	*comment string*	19, 32
Definition	*unit-expression*	

Table 32.6: UnitParameter attributes

The Quantity attribute

You should specify the Quantity attribute if all unit values stored in the unit parameter can be associated with a single quantity declared in your model. The effect of specifying a quantity in the Quantity attribute of a unit parameter is twofold:

- during assignments to the unit parameter, AIMMS will verify whether the assigned unit values are commensurate with the base unit of specified quantity, and
- AIMMS will modify its (compile-time) unit analysis to use the specified quantity rather than an artificial quantity based on the name of the unit parameter (see below).

The Default and Definition attributes

The Default and Definition attributes of a unit parameter have the same purpose as the Default and Definition attribute of ordinary parameters, except that the resulting values must be unit expressions (see Section 32.6). If you have specified a quantity in the Quantity attribute, AIMMS will verify that these unit expressions are commensurate with the specified quantity.

Allowed unit values

All unit values read from an external data source, or assigned to a unit parameter, either via an assignment or through its Definition attribute, must evaluate to existing unit symbols only. A compile- or runtime error will occur, when a unit value refers to a unit symbol that is not defined in any of the Quantity declarations contained in your model.

Use of unit parameters

With unit parameters you can create, store and manipulate scalar or multidimensional collections of unit values. The unit values stored in a unit parameter can be used, for instance:

- to associate a parametrized (i.e. multidimensional) collection of units with a single multidimensional identifier (through its Unit attribute), or
- to specify a local unit override based on a unit (or collection of units) that is not known a priori.

Unit analysis...

When a Unit attribute of an identifier contains a reference to a unit parameter, this can, but need not, modify the way in which AIMMS conducts its usual unit analysis. There are two distinct scenarios, both described below.

... with associated quantity

If the unit parameter has an associated quantity (specified through its Quantity attribute), all units stored in the unit parameter are known to be commensurate with the base unit of the quantity, although the individual scale factors may be different if the unit parameter is multidimensional. In this case, AIMMS will base its unit analysis on the associated quantity.

... without associated quantity

If there is no associated quantity, AIMMS will introduce an artificial quantity solely on the basis of the symbolic name of the unit parameter (i.e. without consideration of its dimension), and base all further unit analysis on this artificial quantity only. If there is unit consistency at the level of these artificial quantities, this automatically ensures, for multidimensional unit parameters, unit consistency at the individual level as well, regardless of the specific individual unit values stored in it.

Example

Consider the following declarations of unit-valued parameters, where f is an index into the set Foods and n an index into the set Nutrients.

```
UnitParameter NutrientUnit {
    IndexDomain  : n;
}
UnitParameter FoodUnit {
    IndexDomain  : f;
}
```

With these unit-valued parameters you can specify meaningful indexed unit expressions for the Unit attribute of the following parameters.

```
Parameter NutritiveValue {
    IndexDomain  : (f,n);
    Unit         : NutrientUnit(n)/FoodUnit(f);
}
Parameter NutrientMinimum {
    IndexDomain  : n;
    Unit         : NutrientUnit(n);
}
Variable Serving {
    IndexDomain  : f,
    Unit         : FoodUnit(f);
}
```

With these declarations, you can now easily verify that all terms in the definition of the following constraint are unit consistent at the symbolic level.

```
Constraint NutrientRequirement {
    IndexDomain  : n;
    Unit         : NutrientUnit(n);
    Definition   : sum[ f, Servings(f)*NutritiveValue(f,n) ] >=  NutrientMinimum(n);
}
```

Indexed scaling

When the Unit attribute of an identifier is parametrized by means of indexed unit parameter, AIMMS will correctly scale all data exchange with external components (see Section 32.5). During data exchange with an external component, AIMMS considers the specified units at the individual (indexed) level, and will determine the proper scaling for every individual index position. In addition, when a unit convention is active, AIMMS will scale all individual entries according to that convention, as applied to the corresponding individual entries of the indexed unit parameter. As usual, all data of an identifier with a parametrized associated unit will be stored internally in the corresponding atomic unit of every individual index value.

Example revisited

When AIMMS generates mathematical program which contains the variable Serving(f), each column corresponding to this variable will be scaled according to the scale factor of the particular unit stored in FoodUnit(f) with respect to their corresponding atomic unit expressions. Similarly, AIMMS will scale the columns corresponding to the constraint NutrientRequirement(n) according the scale factors of the units stored in NutrientUnit(n) with respect to their corresponding atomic unit expressions.

Initializing unit-valued parameters

You can initialize a unit-valued parameter through lists, tables, and composite tables like you can initialize any other AIMMS parameter (see Chapter 28). The values of the individual entries must be valid unit constants (see Section 32.6), and must be surrounded by square brackets. For compound units constants you can optionally indicate the associated quantity in a similar way as in the unit definition of a parameter.

Example

The following list initializes the unit-valued parameter NutrientUnit for a particular set of Nutrients.

```
NutrientUnit := DATA { Energy  : [kJ]  ,
                       Protein : [mg]  ,
                       Iron    : [%RDA] };
```

Unit parameters and databases

In addition, AIMMS allows you to read the initial data of a unit parameter from a database table, and write the values of a unit parameter to a database table. The unit values in the database table must be unit constants, and must be stored without square brackets.

Simultaneous unit and data initialization

When a composite table in a data file, or a table in a database contains both the values of a multidimensional unit parameter, and a corresponding numeric parameter whose Unit attribute references that unit parameter, AIMMS allows you to read both identifiers in a single pass. When reading both identifiers, AIMMS will make sure that the numeric values are interpreted with respect to the corresponding unit value that is read simultaneously.

Constant versus parametrized units

AIMMS even allows you to make assignments from identifiers with a constant unit to identifier slices of identifiers with a parametrized unit and vice versa. If AIMMS detects this special situation during compilation of your model, it will postpone the compile unit consistency check whenever necessary, and replace it with a runtime consistency check which is performed every time the assignment is executed. Because all data is stored by AIMMS with respect to atomic units internally, unit consistency again automatically implies scale consistency.

Example

Given the declarations of the previous example, assume the existence of an additional parameter EnergyContent(f) with a constant associated unit, say Kcal. Then, AIMMS will postpone the compile unit consistency check for the following two statements, and replace it with a runtime check.

```
NutritiveValue(f,'Energy') := EnergyContent(f);
EnergyContent(f)           := NutritiveValue(f,'Energy');
```

The runtime unit consistency check will only succeed, whenever the unit value of the unit parameter NutrientUnit('Energy') is commensurate with the constant unit Kcal.

Restrictions

AIMMS will only replace a compile time with a runtime unit consistency check if a unique unit can be associated with the right-hand side of the assignment at compile time. If the assigned expression consists of subexpressions which have different associated unit expressions at compile time, a compile time error will result. This is even the case when, at runtime, these unit expressions evaluate to units that are commensurate with the unit of the left-hand side of the assignment.

Chapter 33

Time-Based Modeling

This chapter

In AIMMS there are three fundamental building blocks for time-based modeling namely *horizons*, *calendars* and *timetable-based aggregation and disaggregation*. These concepts coincide with your natural view of time, but there are associated details that need to be examined. Using these building blocks, you can develop time-dependent model-based applications with substantially less effort than would otherwise be required.

33.1 Introduction

Time and models

Time plays an important role in various real-life modeling applications. Typical examples are found in the areas of planning, scheduling, and control. The time scale in control models is typically seconds and minutes. Scheduling models typically refer to hours and days, while the associated time unit in planning models is usually expressed in terms of weeks, months, or even years. To facilitate time-based modeling, AIMMS provides a number of tools to relate model time and calendar time.

Use of time periods

Time-dependent data in a model is usually associated with time periods. Some data items associated with a period index can be interpreted as taking place *during* the period, while others take place *at a particular moment*. For instance, the stock in a tank is usually measured at, and associated with, a specific moment in a period, while the flow of material into the tank is usually associated with the entire period.

Use of time as a continuous quantity

Time-dependent data in a model can also represent continuous time values. For instance, consider a parameter containing the starting times of a number of processes. Even though this representation is not ideal for constructing most time-based optimization models, it allows time to be expressed to any desired accuracy.

Calendar periods versus model periods

A large portion of the data in time-dependent models originates from the real world where quantities are specified relative to some calendar. Optimization models usually refer to abstract model periods such as p_1, p_2, p_3, etc., allowing the optimization model to be formulated independent of real time. This common distinction makes it essential that quantities associated with real calendar time can be converted to quantities associated with model periods and vice versa.

Rolling horizon

In many planning and scheduling applications, time-dependent models are solved repeatedly as time passes. Future data becomes present data and eventually becomes past data. Such a moving time span is usually referred to as a "*rolling horizon*". By using the various features discussed in this chapter, it is fairly straightforward to implement models with a rolling horizon.

Calendars and Horizons

AIMMS offers two special data types for time-based modeling applications, namely `Calendar` and `Horizon`. Both are index sets with special features for dealing with time. `Calendars` allow you to create a set of time slots of fixed length in real time, while `Horizons` enable you to distinguish past, planning and beyond periods in your model.

Timetables

In addition, AIMMS offers support for automatically creating *timetables* (represented through indexed sets) which link model periods in a `Horizon` to time slots in a `Calendar` in a flexible manner. Based on a timetable, AIMMS provides functions to let you *aggregate* data defined over a `Calendar` to data defined over the corresponding `Horizon` and vice versa. Figure 33.1 illustrates an example of a timetable relating a horizon and a calendar.

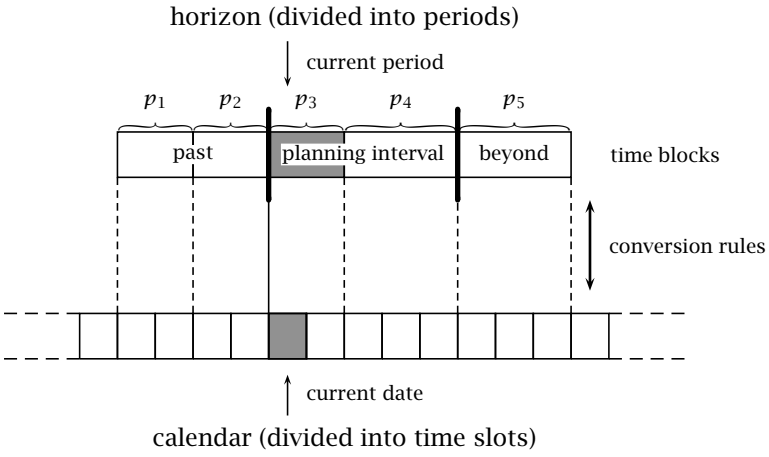


Figure 33.1: Timetable relating calendar and horizon

Explanation

The horizon consists of periods divided into three time blocks, namely a past, the planning interval, and beyond. There is a current period in the horizon which can be linked to a current date in the calendar. The calendar consists of time slots and its range is defined by a begin date and an end date. When you construct your mathematical program, it will typically be in terms of periods in the planning interval of the horizon. However, the input data of the model will typically be in terms of calendar periods. The conversion of calendar data into horizon data and vice versa is done on request by AIMMS in accordance with pre-specified conversion rules.

33.2 Calendars

Calendars

A *calendar* is defined as a set of consecutive *time slots* of unit length covering the complete time frame from the calendar's begin date to its end date. You can use a calendar to index data defined in terms of calendar time.

Calendar attributes

Calendars have several associated attributes, which are listed in Table 33.1. Some of these attributes are inherited from sets, while others are new and specific to calendars. The new ones are discussed in this section.

Attribute	Value-type	See also page	Mandatory
BeginDate	*string*		yes
EndDate	*string*		yes
Unit	*unit*		yes
TimeslotFormat	*string*		yes
Index	*identifier-list*	32	yes
Parameter	*identifier-list*	32	
Text	*string*	19	
Comment	*comment string*	19	

Table 33.1: Calendar attributes

Unit

The Unit attribute defines the length of a single time slot in the calendar. It must be specified as one of the following time units or an integer multiple thereof:

- century,
- year,
- month,
- day,
- hour,
- minute,

- second, and
- tick (i.e. sec/100).

Thus, 15*min and 3*month are valid time units, but the equivalent 0.25*hour and 0.25*year are not. Besides a constant integer number it is also allowed to use an AIMMS parameter to specify the length of the time slots in the calendar (e.g. NumberOfMinutesPerTimeslot*min).

Not predefined

Although you can only use the fixed unit names listed above to specify the Unit attribute of a calendar, AIMMS does not have a predefined Quantity for time (see also Chapter 32). This means that the units of time you want to use in your model, do not have to coincide with the time units required in the calendar declaration. Therefore, prior to specifying the Unit attribute of a calendar, you must first specify a quantity defining both your own time units and the conversion factors to the time units required by AIMMS. In the **Model Explorer**, AIMMS will automatically offer to add the relevant time Quantity to your model when the calendar unit does not yet exist in the model tree.

BeginDate and EndDate

The mandatory BeginDate and EndDate attributes of a calendar specify its range. AIMMS will generate all time slots of the specified length, whose *begin time* lies between the specified BeginDate and EndDate. As a consequence, the *end time* of the last time slot may be after the specified EndDate. An example of this behavior occurs, for instance, when the requested length of all time slots is 3 days and the EndDate does not lie on a 3-day boundary from the BeginDate. Any period references that start outside this range will be ignored by the system. This makes it easy to select all relevant time-dependent data from a database.

Reference date format

Any set element describing either the BeginDate or the EndDate must be given in the following fixed *reference date* format which contains the specific year, month, etc. up to and including the appropriate reference to the time unit associated with the calendar.

```
YYYY-MM-DD hh:mm:ss
```

All entries must be numbers *with leading zeros present*. The hours are expressed using the 24-hour clock. You do not need to specify all entries. Only those fields that refer to time units that are longer or equal to the predefined AIMMS time unit in your calendar are required. All time/date fields beyond the requested granularity are ignored. For instance, a calendar expressed in hours may have a BeginDate such as

- "1996-01-20 09:00:00", or
- "1996-01-20 09:00", or
- "1996-01-20 09",

which all refer to exactly the same time, 9:00 AM on January 20*th*, 1996.

Time zone and DST offsets

AIMMS always assumes that reference dates are specified according to the local time zone without daylight saving time. However, for calendars with granularity day AIMMS will ignore any timezone and daylight saving time offsets, and just take the day as specified. In the example above, a daily calendar with the above BeginDate will always start with period "1996-01-20", while an hourly calendar may start with a period "1996-01-19 23:00" if the difference between the local time zone, and the time zone specification in the timeslot format is 10 hours.

Format of time-related attributes

Set elements and string-valued parameters capturing time-related information must deal with a variety of formatting possibilities in order to meet end-user requirements around the globe (there are no true international standards for formatting time slots and time periods). The flexible construction of dates and date formats using the TimeslotFormat is presented in Section 33.7.

Example

The following example is a declaration of a daily calendar and a monthly calendar

```
Calendar DailyCalendar {
    Index            : d;
    Parameter        : CurrentDay;
    Text             : A work-week calendar for production planning;
    BeginDate        : "1996-01-01";
    EndDate          : "1997-06-30";
    Unit             : day;
    TimeslotFormat   : {
        "%d/%m/%y"      ! format explained later
    }
}
Calendar MonthlyCalendar {
    Index            : m;
    BeginDate        : CalendarBeginMonth;
    EndDate          : CalendarEndMonth;
    Unit             : month;
    TimeslotDormat   : {
        "%m/%y"         ! format explained later
    }
}
```

Varying number of time slots

The calendar DailyCalendar thus declared will be a set containing the elements '01/01/96',...,'06/30/97' for every day in the period from January 1, 1996 through June 30, 1997. When the BeginDate and EndDate attributes are specified as string parameters containing the respective begin and end dates (as in MonthlyCalendar), the number of generated time slots can be changed dynamically. In order to generate zero time slots, leave one of these string parameters empty.

Time zones and daylight saving time

By default, AIMMS assumes that a calendar uses the local time zone without daylight saving time, in accordance with the specification of the BeginDate and EndDate attributes. However, if this is not the case, you can modify the TimeslotFormat attribute in such a manner, that AIMMS

- will take daylight saving time into account during the construction of the calendar slots, or,
- will generate the calendar slots according to a specified time zone.

In both cases, AIMMS still requires that the BeginDate and EndDate attributes be specified as reference dates in the local time zone without daylight saving time, as already indicated. Support for time zones and daylight saving time is explained in full detail in Section 33.7.4.

33.3 Horizons

Horizons

A horizon in AIMMS is basically a set of planning periods. The elements in a horizon are divided into three groups, also referred to as time blocks. The main group of elements comprise the *planning interval*. Periods prior to the planning interval form the *past*, while periods following the planning interval form the *beyond*. When variables and constraints are indexed over a horizon, AIMMS automatically restricts the generation of these constraints and variables to periods within the planning interval.

Effect on constraints and assignments

Whenever you use a horizon to construct a time-dependent model, AIMMS has the following features:

- constraints are excluded from the past and beyond periods,
- variables are assumed to be fixed for these periods, and
- assignments and definitions to variables and parameters are, by default, only executed for the periods in the planning interval.

Horizon attributes

Horizons, like calendars, have a number of associated attributes, some of which are inherited from sets. They are summarized in Table 33.2.

Horizon attributes explained

The CurrentPeriod attribute denotes the first period of the planning interval. The periods prior to the current period belong to the past. The integer value associated with the attribute IntervalLength determines the number of periods in the planning interval (including the current period). Without an input value, all periods from the current period onwards are part of the planning interval.

Attribute	Value-type	See also page	Mandatory
SubsetOf	*subset-domain*	32	
Index	*identifier-list*	32	yes
Parameter	*identifier-list*	32	
Text	*string*	19	
Comment	*comment string*	19	
Definition	*set-expression*	34	yes
CurrentPeriod	*element*		yes
IntervalLength	*integer-reference*		

Table 33.2: Horizon attributes

Definition is mandatory

AIMMS requires you to specify the contents of a Horizon uniquely through its Definition attribute. The ordering of the periods in the horizon is fully determined by the set expression in its definition. You still have the freedom, however, to specify the Horizon as a subset of another set.

Example

Given a scalar parameter MaxPeriods, the following example illustrates the declaration of a horizon.

```
Horizon ModelPeriods {
    Index          : h;
    Parameter      : IntervalStart;
    CurrentPeriod  : IntervalStart;
    Definition     : ElementRange( 1, MaxPeriods, prefix: "p-" );
}
```

If, for instance, the scalar MaxPeriods equals 10, this will result in the creation of a period set containing the elements 'p-01',...,'p-10'. The start of the planning interval is fully determined by the value of the element parameter IntervalStart, which for instance could be equal to 'p-02'. This will result in a planning interval consisting of the periods 'p-02',...,'p-10'.

Example of use

Consider the parameter Demand(h) together with the variables Production(h) and Stock(h). Then the definition of the variable Stock can be declared as follows.

```
Variable Stock {
    IndexDomain    : h;
    Range          : NonNegative;
    Definition     : Stock(h-1) + Production(h) - Demand(h);
}
```

When the variable Stock is included in a mathematical program, AIMMS will only generate individual variables with their associated definition for those values of h that correspond to the current period and onwards. The reference Stock(h-1) refers to a fixed value of Stock from the past whenever the index h

points to the current period. The values associated with periods from the past (and from the beyond if they were there) are assumed to be fixed.

Accessing past and beyond

To provide easy access to periods in the past and the beyond, AIMMS offers three horizon-specific suffices. They are:

- the Past suffix,
- the Planning suffix, and
- the Beyond suffix.

These suffices provide access to the subsets of the horizon representing the past, the planning interval and the beyond.

Horizon binding rules

When you use a horizon index in an index binding operation (see Chapter 9), AIMMS will, by default, perform that operation only for the periods in the planning interval. You can override this default behavior by a local binding using the suffices discussed above.

Example

Consider the horizon ModelPeriods of the previous example. The following assignments illustrate the binding behavior of horizons.

```
Demand(h)                          := 10; ! only periods in planning interval (default)
Demand(h in ModelPeriods.Planning) := 10; ! only periods in planning interval

Demand(h in ModelPeriods.Past)     := 10; ! only periods in the past
Demand(h in ModelPeriods.Beyond)   := 10; ! only periods in the beyond

Demand(h in ModelPeriods)          := 10; ! all periods in the horizon
```

Use of lag and lead operators

When you use one of the lag and lead operators +, ++, - or -- (see also Section 5.2.3) in conjunction with a horizon index, AIMMS will interpret such references with respect to the *entire* horizon, and not just with respect to the planning period. If the horizon index is locally re-bound to one of the subsets of periods in the Past or Beyond, as illustrated above, the lag or lead operation will be interpreted with respect to the specified subset.

Example

Consider the horizon ModelPeriods of the previous example. The following assignments illustrate the use of lag and lead operators in conjuction with horizons.

```
Stock(h)                           := Stock(h-1) + Supply(h) - Demand(h);
Stock(h | h in ModelPeriods.Planning) := Stock(h-1) + Supply(h) - Demand(h);

Stock(h in ModelPeriods.Planning)  := Stock(h-1) + Supply(h) - Demand(h);
Stock(h in ModelPeriods.Planning)  := Stock(h--1) + Supply(h) - Demand(h);
```

The first two assignments are completely equivalent (in fact, the second assignment is precisely the way in which AIMMS interprets the default binding behavior of a horizon index). For the first element in the planning interval,

the reference h-1 refers to the last element of the past interval. In the third assignment, h-1 refers to a non-existing element for the first element in the planning interval, completely in accordance with the default semantics of lag and lead operators. In the fourth assignment, h--1 refers to the last element of the planning interval.

Data transfer on entire domain

Operations which can be applied to identifiers without references to their indices (such as the READ, WRITE or DISPLAY statements), operate on the entire horizon domain. Thus, for example, during data transfer with a database, AIMMS will retrieve or store the data for *all* periods in the horizon, and not just for the periods in the planning interval.

33.4 Creating timetables

Timetables

A *timetable* in AIMMS is an indexed set, which, for every period in a Horizon, lists the corresponding time slots in the associated Calendar. Timetables play a central role during the conversion from calendar data to horizon data and vice versa.

The procedure CreateTimeTable

Through the predefined procedure CreateTimeTable, you can request AIMMS to flexibly construct a *timetable* on the basis of

- a *time slot* in the calendar and a *period* in the horizon that should be aligned at the beginning of the planning interval,
- the desired *length* of each period in the horizon expressed as a number of time slots in the calendar,
- an indication, for every period in the horizon, whether the *length dominates* over any specified delimiter slots,
- a set of *inactive time slots*, which should be excluded from the timetable and, consequently, from the period length computation, and
- a set of *delimiter time slots*, at which new horizon periods should begin.

Syntax

The syntax of the procedure CreateTimeTable is as follows:

- CreateTimeTable(*timetable*, *current-timeslot*, *current-period*,
  period-length, *length-dominates*,
  inactive-slots, *delimiter-slots*)

The (output) *timetable* argument of the procedure CreateTimeTable must, in general, be an indexed set in a calendar and defined over the horizon to be linked to the calendar. Its contents is completely determined by AIMMS on the basis of the other arguments. The *current-timeslot* and *current-period* arguments must be elements of the appropriate calendar and horizon, respectively.

Element parameter as timetable

In the special case that you know a priori that each period in the timetable is associated with exactly one time slot in the calendar, AIMMS also allows the *timetable* argument of the `CreateTimeTable` procedure to be an element parameter (instead of an indexed set). When you specify an element parameter, however, a runtime error will result if the input arguments of the call to `CreateTimeTable` give rise to periods consisting of multiple time slots.

Several possibilities

You have several possibilities of specifying your input data which influence the way in which the timetable is created. You can:

- only specify the length of each period to be created,
- only specify delimiter slots at which a new period must begin, or
- flexibly combine both of the above two methods.

Period length

The *period-length* argument must be a positive integer-valued one-dimensional parameter defined over the horizon. It specifies the desired length of each period in the horizon in terms of the number of time slots to be contained in it. If you do not provide delimiter slots (explained below), AIMMS will create a timetable solely on the basis of the indicated period lengths.

Inactive slots

The *inactive-slots* argument must be a subset of the calendar that is specified as the range of the *timetable* argument. Through this argument you can specify a set of time slots that are always to be excluded from the timetable. You can use this argument, for instance, to indicate that weekend days or holidays are not to be part of a planning period. Inactive time slots are excluded from the timetable, and are not accounted for in the computation of the desired period length.

Delimiter slots

The *delimiter-slots* argument must be a subset of the calendar that is specified as the range of the *timetable* argument. AIMMS will begin a new period in the horizon whenever it encounters a delimiter slot in the calendar provided no (offending) period length has been specified for the period that is terminated at the delimiter slot.

Combining period length and delimiters

In addition to using either of the above methods to create a timetable, you can also combine them to create timetables in an even more flexible manner by specifying the *length-dominates* argument, which must be a one-dimensional parameter defined over the horizon. The following rules apply.

- If the *length-dominates* argument is nonzero for a particular period, meeting the specified period length prevails over any delimiter slots that are possibly contained in that period.
- If the *length-dominates* argument is zero for a particular period and the specified period length is 0, AIMMS will not restrict that period on the basis of length, but only on the basis of delimiter slots.

- If the *length-dominates* argument is zero for a particular period and the specified period length is positive, AIMMS will try to construct a period of the indicated length, but will terminate the period earlier if it encounters a delimiter slot first.

Timetable creation

In creating a timetable, AIMMS will always start by aligning the *current-timeslot* argument with the beginning of the *current-period*. Periods beyond *current-period* are determined sequentially by moving forward time slot by time slot, until a new period must be started due to hitting the period length criterion of the current period (taking into account the inactive slots), or by hitting a delimiter slot. Periods prior to *current-period* are determined sequentially by moving backwards in time starting at *current-timeslot*.

Adapting timetables

As a timetable is nothing more than an indexed set, you still have the opportunity to make manual changes to a timetable after its contents have been computed by the AIMMS procedure `CreateTimeTable`. This allows you to make any change to the timetable that you cannot, or do not want to, implement directly using the procedure `CreateTimeTable`.

Example

Consider a timetable which links the daily calendar declared in Section 33.2 and the horizon of Section 33.3, which consists of 10 periods named `p-01` ... `p-10`. The following conditions should be met:

- the planning interval starts at period `p-02`, i.e. period `p-01` is in the past,
- periods `p-01`...`p-05` have a fixed length of 1 day,
- periods `p-06`...`p-10` should have a length of at most a week, with new periods starting on every Monday.

To create the corresponding timetable using the procedure `CreateTimeTable`, the following additional identifiers need to be added to the model:

- an indexed subset `TimeTable(h)` of `DailyCalendar`,
- a subset `DelimiterDays` of `DailyCalendar` containing all Mondays in the calendar (i.e. '01-01-96', '08-01-96', etc.),
- a subset `InactiveDays` of `DailyCalendar` containing all days that you want to exclude from the timetable (e.g. all weekend days),
- a parameter `PeriodLength(h)` assuming the value 1 for the periods `p-01` ... `p-05`, and zero otherwise,
- a parameter `LengthDominates(h)` assuming the value 1 for the periods `p-01` ... `p-05`, and zero otherwise.

To compute the contents of the timetable, aligning the time slot pointed at by `CurrentDay` and period `IntervalStart`, one should call

```
CreateTimeTable( TimeTable, CurrentDay, IntervalStart,
                 PeriodLength, LengthDominates,
                 InactiveDays, DelimiterDays );
```

Period	Calendar slots	Period	Calendar slots
p-01	23/01/96 (Tue)	p-06	30/01/96 - 02/02/96 (Tue-Fri)
p-02	24/01/96 (Wed)	p-07	05/01/96 - 09/02/96 (Mon-Fri)
p-03	25/01/96 (Thu)	p-08	12/01/96 - 16/02/96 (Mon-Fri)
p-04	26/01/96 (Fri)	p-09	19/01/96 - 23/02/96 (Mon-Fri)
p-05	29/01/96 (Mon)	p-10	26/01/96 - 01/03/96 (Mon-Fri)

If all weekend days are inactive, and CurrentDay equals '24/01/96' (a Wednesday), then TimeTable describes the following mapping.

The function Timeslot-Characteristic

The process of initializing the sets used in the *delimiter-slots* and *inactive-slots* arguments can be quite cumbersome when your model covers a large time span. For that reason AIMMS offers the convenient function TimeslotCharacteristic. With it, you can obtain a numeric value which characterizes the time slot, in terms of its day of the week, its day in the year, etc. The syntax of the function is straightforward:

- TimeslotCharacteristic(*timeslot*, *characteristic*[, *timezone*[, ignoredst]])

The *characteristic* argument must be an element of the predefined set TimeslotCharacteristics. The elements of this set, as well as the associated function values are listed in Table 33.3.

Characteristic	Function value range	First
century	0, ..., 99	
year	0, ..., 99	
quarter	1, ..., 4	
month	1, ..., 12	January
weekday	1, ..., 7	Monday
yearday	1, ..., 366	
monthday	1, ..., 31	
week	1, ..., 53	
weekyear	0, ..., 99	
weekcentury	0, ..., 99	
hour	0, ..., 23	
minute	0, ..., 59	
second	0, ..., 59	
tick	0, ..., 99	
dst	0, 1	

Table 33.3: Elements of the set TimeslotCharacteristics

Day and week numbering

Internally, AIMMS takes Monday as the first day in a week, and considers week 1 as the first week that contains at least four days of the new year. This is equivalent to stating that week 1 contains the first Thursday of the new year. Through the 'week', 'weekyear' and 'weekcentury' characteristics you obtain the week number corresponding to a particular date and its corresponding year and century. For instance, Friday January 1, 1999 is day 5 of week 53 of year 1998.

Example

Consider a daily calendar DailyCalendar with index d. The following assignment to a subset WorkingDays of a DailyCalendar will select all non-weekend days in the calendar.

```
WorkingDays := { d | TimeslotCharacteristic(d,'weekday') <= 5 } ;
```

Calendar-calendar linkage

You can also use the function TimeslotCharacteristic to create a timetable linking two calendars (e.g. to create monthly overviews of daily data). As an example, consider the calendars DailyCalendar and MonthlyCalendar declared in Section 33.2, as well as an indexed set MonthDays(m) of DailyCalendar, which can serve as a timetable. MonthDays can be computed as follows.

```
MonthDays(m) := { d | TimeslotCharacteristic(d,'year') =
                          TimeslotCharacteristic(m,'year') and
                      TimeslotCharacteristic(d,'month') =
                          TimeslotCharacteristic(m,'month')    };
```

A check on the 'year' characteristic is not necessary if both calendars are contained within a single calendar year.

Time zone support

Through the optional *timezone* argument of the function TimeslotCharacteristic, you can specify with respect to which time zone you want to obtain the specified characteristic. The *timezone* argument must be an element of the predefined set AllTimeZones (see also Section 33.7.4). By default, AIMMS assumes the local time zone without daylight saving time.

Daylight saving time

When you specify a time zone with daylight saving time, you can retrieve whether daylight saving time is active through the 'dst' characteristic. With the optional argument *ignoredst* (default 0) of the function TimeSlotCharacteristic, you can specify whether you want daylight saving time to be ignored. With *ignoredst* set to 1, or in a time zone without daylight saving time, the outcome for the 'dst' characteristic will always be 0.

33.5 Data conversion of time-dependent identifiers

Time-dependent data

When you are working with time-dependent data, it is usually not sufficient to provide and work with a single fixed-time scale. The following examples serve as an illustration.

- Demand data is available in a database on a day-by-day basis, but is needed in a mathematical program for each horizon period.
- Production quantities are computed per horizon period, but are needed on a day-by-day basis.
- For all of the above data weekly or monthly overviews are also required.

The procedures `Aggregate` and `Disaggregate`

With the procedures `Aggregate` and `Disaggregate` you can instruct AIMMS to perform an aggregation or disaggregation step from one time scale to another. Both procedures perform the aggregation or disaggregation of a single identifier in one time scale to another identifier in a second time scale, given a timetable linking both time scales and a predefined aggregation type. The syntax is as follows.

- `Aggregate`(*timeslot-data*, *period-data*, *timetable*, *type*[, *locus*])
- `Disaggregate`(*period-data*, *timeslot-data*, *timetable*, *type*[, *locus*])

Time slot and period data

The identifiers (or identifier slices) passed to the `Aggregate` and `Disaggregate` procedures holding the time-dependent data must be of equal dimension. All domain sets in the index domains must coincide, except for the time domains. These must be consistent with the domain and range of the specified timetable.

Different conversions

As was mentioned in Section 33.1, time-dependent data can be interpreted as taking place *during* a period or *at a given moment* in the period. Calendar data, which takes place *during* a period, needs to be converted into a period-based representation by allocating the data values in proportion to the overlap between time slots and horizon periods. On the other hand, calendar data which takes place *at a given moment*, needs to be converted to a period-based representation by linearly interpolating the original data values.

Aggregation types

The possible values for the *type* argument of the `Aggregate` and `Disaggregate` procedures are the elements of the predefined set `AggregationTypes` given by:

- `summation`,
- `average`,
- `maximum`,
- `minimum`, and
- `interpolation`.

Reverse conversion

All of the above predefined conversion rules are characterized by the following property.

> *The disaggregation of period data into time slot data, followed by immediate aggregation, will reproduce identical values of the period data.*

Aggregation followed by disaggregation does not have this property. Fortunately, as the horizon rolls along, disaggregation followed by aggregation is the essential conversion.

The `summation` rule

The conversion rule `summation` is the most commonly used aggregation/disaggregation rule for quantities that take place *during* a period. It is appropriate for such typical quantities as production and arrivals. Data values from a number of consecutive time slots in the calendar are summed together to form a single value for a multi-unit period in the horizon. The reverse conversion takes place by dividing the single value equally between the consecutive time slots.

The `average`, `maximum`, and `minimum` rules

The conversion rules `average`, `maximum`, and `minimum` are less frequently used aggregation/disaggregation rules for quantities that take place *during* a period. These rules are appropriate for such typical quantities as temperature or capacity. Aggregation of data from a number of consecutive time slots to a single period in the horizon takes place by considering the average or the maximum or minimum value over all time slots contained in the period. The reverse conversion consists of assigning the single value to each time slot contained in the period.

Illustration of aggregation

Table 33.4 demonstrates the aggregation and disaggregation taking place for each conversion rule. The conversion operates on a single period consisting of 3 time slots in the calendar.

Conversion rule	**Calendar to horizon**			**Horizon to calendar**		
	3	1	2	3		
`summation`	6			1	1	1
`average`	2			3	3	3
`maximum`	3			3	3	3
`minimum`	1			3	3	3

Table 33.4: Conversion rules for “during” quantities

Interpolation

The `interpolation` rule should be used for all quantities that take place *at a given moment* in a period. For the `interpolation` rule you have to specify one additional argument in the `Aggregate` and `Disaggregate` procedures, the *locus*. The *locus* of the `interpolation` defines at which moment in a period—as a value between 0 and 1—the quantity at hand is to be measured. Thus, a *locus* of 0 means that the quantity is measured at the beginning of every period, a *locus* of 1 means that the quantity is measured at the end of every period, while a *locus* of 0.5 means that the quantity is measured midway through the period.

Interpolation for disaggregation

When disaggregating data from periods to time slots, AIMMS interpolates linearly between the respective loci of two subsequent periods. For the outermost periods, AIMMS assigns the last available interpolated value.

Interpolation for aggregation

AIMMS applies a simple rule for the seemingly awkward interpolation of data from unit-length time slots to variable-length horizon periods. It will simply take the value associated with the time slot in which the `locus` is contained, and assign it to the period. This simple rule works well for loci of 0 and 1, which are the most common values.

Illustration of interpolation

Table 33.5 demonstrates aggregation and disaggregation of a horizon of 3 periods, each consisting of 3 time slots, for loci of 0, 1, and 0.5. The underlined values are the values determined by the reverse conversion.

Locus	**Horizon data**								
	0			3			9		
0	<u>0</u>	1	2	<u>3</u>	5	7	<u>9</u>	9	9
1	0	0	<u>0</u>	1	2	<u>3</u>	5	7	<u>9</u>
0.5	0	<u>0</u>	1	2	<u>3</u>	5	7	<u>9</u>	9

Table 33.5: Conversion rules for interpolated data

Example

Consider the calendar `DailyCalendar`, the horizon `ModelPeriods` and the timetable `TimeTable` declared in Sections 33.2, 33.3 and 33.4, along with the identifiers

- `DailyDemand(d)`,
- `Demand(h)`,
- `DailyStock(d)`, and
- `Stock(h)`.

The aggregation of `DailyDemand` to `Demand` can then be accomplished by the statement

```
Aggregate( DailyDemand, Demand, TimeTable, 'summation' );
```

Assuming that the Stock is computed at the end of each period, the disaggregation (by interpolation) to daily values is accomplished by the statement

```
Disaggregate( Stock, DailyStock, TimeTable, 'interpolation', locus: 1 );
```

User-defined conversions

If your particular aggregation/disaggregation scheme is not covered by the predefined aggregation types available in AIMMS, it is usually not too difficult to implement a custom aggregation scheme yourself in AIMMS. For instance, the aggregation by summation from DailyDemand to Demand can be implemented as

```
Demand(h) := sum( d in TimeTable(h), DailyDemand(d) );
```

while the associated disaggregation rule becomes the statement

```
DailyDemand(d) := sum( h | d in TimeTable(h), Demand(h)/Card(TimeTable(per)) );
```

33.6 Implementing a model with a rolling horizon

Rolling horizons

The term *rolling horizon* is used to indicate that a time-dependent model is solved repeatedly, and in which the planning interval is moved forward in time during each solution step. With the facilities introduced in the previous sections setting up such a model is relatively easy. This section outlines the steps that are required to implement a model with a rolling horizon, without going into detail regarding the contents of the underlying model.

Two strategies

In this section you will find two strategies for implementing a rolling horizon. One is a simple strategy that will only work with certain restrictions. It requires just a single aggregation step and a single disaggregation step. The other is a generic strategy that will work in all cases. This strategy, however, requires that aggregation and disaggregation steps be performed between every two subsequent SOLVE statements.

Simple strategy

The simple strategy will work provided that

- all periods in the horizon are of equal length, and
- the horizon rolls from period boundary to period boundary.

It is then sufficient to make the horizon sufficiently large so as to cover the whole time range of interest.

Algorithm outline

The algorithm to implement the rolling horizon can be outlined as follows.

1. Select the current time slot and period, and create the global timetable.
2. Aggregate all calendar-based data into horizon-based data.
3. Solve the optimization model for a planning interval that is a subset of the complete horizon.

4. Move the current period to the next period boundary of interest, and repeat from steps until the time range of interest has passed.
5. Disaggregate the horizon-based solution into a calendar-based solution.

Assumptions

The examples below that illustrate both the simple and generic strategy make the following assumptions.

- The model contains the daily calendar `DailyCalendar`, the horizon `ModelPeriods` and the timetable `TimeTable` declared in Sections 33.2, 33.3 and 33.4, respectively.
- The model contains a time-dependent mathematical program `TimeDependentModel`, which produces a plan over the planning interval associated with `ModelPeriods`.
- The planning horizon, for which the model is to be solved, rolls along from `FirstWeekBegin` to `LastWeekBegin` in steps of one week. Both identifiers are element parameters in `DailyCalendar`.

Code outline

The outline of the simple strategy can be implemented as follows.

```
CurrentDay := FirstWeekBegin;
CreateTimeTable( TimeTable   , CurrentDay     , IntervalStart,
                 PeriodLength, LengthDominates,
                 InactiveDays, DelimiterDays  );

Aggregate( DailyDemand, Demand, TimeTable, 'summation' );
! ... along with any other aggregation required

repeat
    solve TimeDependentModel;

    CurrentDay    += 7;
    IntervalStart += 1;

    break when (not CurrentDay) or (CurrentDay > LastWeekBegin);
endrepeat;

Disaggregate( Stock     , DailyStock     , TimeTable, 'interpolation', locus: 1 );
Disaggregate( Production, DailyProduction, TimeTable, 'summation' );
! ... along with any other disaggregation required
```

Generic strategy

The simple strategy will not work

- whenever the lengths of periods in the horizon (expressed in time slots of the calendar) vary, or
- when the start of a new planning interval does not align with a future model period.

In both cases, the horizon-based solution obtained from a previous solve will not be accurate when you move the planning interval. Thus, you should follow a generic strategy which adds an additional disaggregation and aggregation step to every iteration.

Algorithm outline

The generic strategy for implementing a rolling horizon is outlined as follows.

1. Select the initial current time slot and period, and create the initial timetable.
2. Aggregate all calendar-based data into horizon-based data.
3. Solve the mathematical program.
4. Disaggregate all horizon-based variables to calendar-based identifiers.
5. Move the current time slot forward in time, and recreate the timetable.
6. Aggregate all identifiers disaggregated in step 4 back to the horizon using the updated timetable.
7. Repeat from step 2 until the time range of interest has passed.

Code outline

The outline of the generic strategy can be implemented as follows.

```
CurrentDay := FirstWeekBegin;
CreateTimeTable( TimeTable   , CurrentDay     , IntervalStart,
                 PeriodLength, LengthDominates,
                 InactiveDays, DelimiterDays  );

repeat
    Aggregate( DailyDemand, Demand, TimeTable, 'summation' );
    ! ... along with any other aggregation required

    solve TimeDependentModel;

    Disaggregate( Stock     , DailyStock     , TimeTable, 'interpolation', locus: 1 );
    Disaggregate( Production, DailyProduction, TimeTable, 'summation' );
    ! ... along with any other disaggregation required

    CurrentDay += 7;
    break when (not CurrentDay) or (CurrentDay > LastWeekBegin);
    CreateTimeTable( TimeTable   , CurrentDay     , IntervalStart,
                     PeriodLength, LengthDominates,
                     InactiveDays, DelimiterDays  );

    Aggregate( DailyStock     , Stock     , TimeTable, 'interpolation', locus: 1 );
    Aggregate( DailyProduction, Production, TimeTable, 'summation' );
    ! ... along with any other aggregation required
endrepeat;
```

33.7 Format of time slots and periods

Flexible time slot and period formats

While the BeginDate and EndDate attributes have to be specified using the fixed *reference date* format (see Section 33.2), AIMMS provides much more flexible formatting capabilities to describe

- time slots in a Calendar consisting of a single basic time unit (e.g. 1-day time slots),
- time slots in a Calendar consisting of multiple basic time units (e.g. 3-day time slots), and
- periods in a timetable consisting of multiple time slots.

The formatting capabilities described in this section are quite extensive, and allow for maximum flexibility.

Wizard support

In the Model Explorer, AIMMS provides a wizard to support you in constructing the appropriate formats. Through this wizard, you can not only select from a number of predefined formats (including some that use the regional settings of your computer), you also have the possibility of constructing a custom format, observing the result as you proceed.

Basic and extended format

AIMMS offers both a *basic* and an *extended format* for the description of time slots and periods. The basic format only refers to the beginning of a time slot or period. The extended format allows you to refer to both the first and last basic time unit contained in a time slot or period. Both the basic and extended formats are constructed according to the same rules.

Care is needed

The `TimeslotFormat` used in a `Calendar` must contain a reference to either its beginning, its end, or both. As the specified format is used to identify calendar elements when reading data from external data sources such as files and databases, you have to ensure that the specified format contains sufficient date and time references to uniquely identify each time slot in a calendar.

Example

For instance, the description "`January 1`" is sufficient to uniquely identify a time slot in a calendar with a range of one year. However, in a two-year calendar, corresponding days in the first and second year are identified using exactly the same element description. In such a case, you must make sure that the specified format contains a reference to a year.

Building blocks

A format description is a sequence of four types of components. These are

- predefined date components,
- predefined time components,
- predefined period references (extended format), and
- ordinary characters.

Ordinary characters

Predefined components begin with the `%` sign. Components that begin otherwise are interpreted as ordinary characters. To use a percent sign as an ordinary character, escape it with another percent sign, as in `%%`.

33.7.1 Date-specific components

Date-specific components

The date-specific components act as conversion specifiers to denote portions of a date description. They may seem rather cryptic at first, but you will find them useful and constructive when creating customized references to time. They are summarized in Table 33.6.

Conversion specifier	Meaning	Possible entries
%d	day	01,...,31
%m	month	01,...,12
%Am|*set-identifier*|	month	*element*
%y	year	00,...,99
%q	quarter	01,...,04
%Y	weekyear	00,...,99
%c	century	00,...,99
%C	weekcentury	00,...,99
%w	day of week	1,...,7
%Aw|*set-identifier*|	day of week	*element*
%W	week of year	01,...,53
%j	day of year	001,...,366

Table 33.6: Conversion specifiers for date components

Custom date-specific references

All date conversion specifiers allow only predefined numerical values, except for the specifiers %Am and %Aw. These allow you to specify references to sets. You can use %Am and %Aw to denote months and days by the elements in a specified set. These are typically the names of the months or days in your native language. AIMMS will interpret the elements by their ordinal number. The predefined identifiers AllMonths, AllAbbrMonths, AllWeekdays and AllAbbrWeekdays hold the full and abbreviated English names of both months and days.

Week year and century

The %Y and %C specifiers refer to the weekyear and weekcentury values of a specific date, as explained on page 554. You can use these if you want to refer to weekly calendar periods by their week number and year.

Omitting leading zeros

AIMMS can interpret numerical date-specific references with or without leading zeros when reading your input data. When writing data, AIMMS will insert all leading zeros to ensure a uniform length for date elements. If you do not want leading zeros for a specific component, you can insert the 's' modifier directly after the % sign. For instance, the string "%sd" will direct AIMMS to produce single-digit numbers for the first nine days.

Omitting trailing blanks

When using the %Am and %Aw specifiers, AIMMS will generate uniform length elements by adding sufficient trailing blanks to the shorter elements. As with leading zeros, you can use the s modifier to override the generation of these trailing blanks.

Example

The format "%Am|AllMonths| %sd, %c%y" will result in the generation of time slots such as 'January 1, 1996'. The date portion of the fixed reference date format used to specify the Begin and EndDate attributes of a calendar can be reproduced using the format "%c%y-%m-%d".

33.7.2 Time-specific components

Time-specific components

The conversion specifiers for time components are listed in Table 33.7. There are no custom time-specific references in this table, because the predefined numerical values are standard throughout the world.

Conversion specifier	Meaning	Possible entries
%h	hour	01,...,12
%H	hour	00,...,23
%M	minute	00,...,59
%S	second	00,...,59
%t	tick	00,...,99
%p	before or after noon	AM, PM

Table 33.7: Conversion specifiers for time components

Omitting leading zeros

AIMMS can interpret numerical time-specific references with or without leading zeros when reading your input data. When writing data, AIMMS will insert leading zeros to ensure a uniform length for time elements. If you do not want leading zeros for a specific component, you can insert the 's' modifier directly after the % sign. For instance, the string "%sh" will direct AIMMS to produce single-digit numbers for the first nine hours.

Example

The time slot format "%sAw|WeekDays| %sh:%M %p" will result in the generation of time slots such as 'Friday 11:00 PM', 'Friday 12:00 PM' and 'Saturday 1:00 AM'. The full reference date format is given by "%c%y-%m-%d %H:%M:%S".

33.7.3 Period-specific components

Use of period references

With period-specific conversion specifiers in either a time slot format or a period format you can indicate that you want AIMMS to display both the begin and end date/time of a time slot or period. You only need to use period-specific references in the following cases.

- The Unit attribute of your calendar consists of a multiple of one of the basic time units known to AIMMS (e.g. each time slot in your calendar

consists of 3 days), and you want to refer to the begin and end day of every time slot.
- You want to provide a description for a period in a timetable consisting of multiple time slots in the associated calendar using the function PeriodToString (see also Section 33.8), referring to both the first and last time slot in the period.

Period-specific components

By including a period-specific component in a time slot or period format, you indicate to AIMMS that any date, or time, specific component following it refers to either the beginning or the end of a time slot or period. The list of available period-specific conversion specifiers is given in Table 33.8.

Conversion specifier	Meaning
%B	begin of unit period
%b	begin of time slot
%I	end of period (inclusive)
%i	end of period (inclusive), but omitted when equal to begin of period
%E	end of period (exclusive)
%e	end of time slot

Table 33.8: Period-specific conversion specifiers.

Inclusive or exclusive

Through the "%I" and "%E" specifiers you can indicate whether you want any date/ time components used in the description of the end of a period (or time slot) to be included in that period or excluded from it. Inclusive behavior is common for date references, e.g. the description "Monday – Wednesday" usually means the period consisting of Monday, Tuesday *and* Wednesday. For time references exclusive behavior is used most commonly, i.e. "1:00 – 3:00 PM" usually means the period from 1:00 PM *until* 3:00 PM.

Limited resolution

After a conversion specifier that refers to the end of a period or time slot (i.e. "%E", "%I" or "%i") you should take care when using other date, or time, specific specifiers. AIMMS will only be able to discern time units that are larger than the basic time unit specified in the Unit attribute of the calendar at hand (or, when you use the function PeriodToString, of the calendar associated with the timetable at hand). For instance, when the time slots of a calendar consists of periods of 2 months, AIMMS will be able to distinguish the specific months at the beginning and end of each time slot, but will not know the specific week number, week day or month day at the end of each time slot. Thus, in this case you should avoid the use of the "%W", the "%w" and the "%d" specifiers after a "%E", "%I" or "%i" specifier.

The %i specifier

With the "%i" specifier you indicate inclusive behavior, and additionally you indicate that AIMMS must omit the remaining text when the basic time units (w.r.t. the underlying calendar) of begin and end slot of the period to which the specifier is applied, coincide. In practice, the "%i" specifier only makes sense when used in the function PeriodToString (see also Section 33.8), as time slots in a calendar always have a fixed length.

First example

The period description "Monday 12:00-15:00" contains three logical references, namely to a day, to the begin time in hours, and to the end time in hours. The day reference is intended to be shared by the begin and end times.

- The day reference is based on the elements of the (predefined) set AllWeekdays. The corresponding conversion specifier is "%Aw|AllWeekDays|"
- The descriptions of the begin and end times both use the conversion specifier "%H:%M". To denote the begin time of the period you must use the "%B" period reference. For the end time of the period, which is not included in the period, you must use "%E".

By combining these building blocks with a few ordinary characters you get the complete format string "%Aw|AllWeekDays| %B%H:%M-%E%H:%M". With this string AIMMS can correctly interpret the element "Monday 12:00-15:00" within a calendar covering no more than one week.

Second example

Consider the format "%B%Aw|AllWeekDays|%I - %Aw|AllWeekDays|" within a calendar with day as its basic time unit, and covering at most a week. Using this format string AIMMS will interpret the element "Monday - Wednesday" as the three-day period consisting of Monday, Tuesday, and Wednesday.

33.7.4 Support for time zones and daylight saving time

Support for daylight saving time

When your time zone has daylight saving time, and you are working with time slots or periods on an hourly basis, you may want to include an indicator into the time slot or period format to indicate whether daylight saving time is active during a particular time slot or period. Such an indicator enables you, for instance, to distinguish between the duplicate hour when the clock is set back at the end of daylight saving time.

Support for time zones

In addition, when your application has users located in different time zones, you may wish to present each user with calendar elements corresponding to their particular time zone. Or, when time-dependent data is stored in a database using UTC time (Universal Time Coordinate, or Greenwich Mean Time), a translation may be required to your own local time representation.

To support you in scenarios as described above, AIMMS provides

AIMMS support

- a special time zone conversion specifier, which can modify the representation of calendar elements based on specified time zone and daylight saving time, and
- a `TimeslotFormat` attribute in unit `Conventions` (see also Section 32.8), which you can use to override the time slot format of every calendar when the `Convention` is active.

With the conversion specifier `%TZ`, described in Table 33.9, you can accomplish the following `Calendar`-related tasks:

Time zone specifier

- create the `Calendar` elements between the given `BeginDate` and `EndDate` relative to a specified time zone, and
- specify the indicators that must be added to the `Calendar` elements when standard or daylight saving time is active.

Conversion specifier	**Meaning**
`%TZ(`*TimeZone*`)`	translation of calendar element to specified *TimeZone*, ignoring daylight saving time
`%TZ(`*TimeZone*`)|`*Std*`|`*Dst*`|`	translation of calendar element to specified *TimeZone*, plus string indicator for standard time (*Std*) and daylight saving time (*Dst*)

Table 33.9: Time zone conversion specifier

The *TimeZone* component of the `%TZ` conversion specifier that you must specify, is a time zone corresponding to the elements in your `Calendar`. You must specify the time zone as an explicit and quoted element of the predefined set `AllTimeZones` (explained below), or through a reference to an element parameter into that set. If you do not specify the *Std* and *Dst* indicators, AIMMS will ignore daylight saving time when generating the time slots, regardless whether daylight saving time is defined for that time zone. If you do not use the `%TZ` specifier to specify a time zone, AIMMS assumes that you intend to use the local time zone without daylight saving time.

Specifying the time zone

AIMMS provides you access to all time zones defined by your operating system through the predefined set `AllTimeZones`. The set `AllTimeZones` contains

The set `AllTimeZones`

- the fixed element `'Local'`, representing the local time zone without daylight saving time,
- the fixed element `'LocalDST'`, representing the local time zone with daylight saving time (if applicable),

- the fixed element 'UTC', representing the Universal Time Coordinate (or Greenwich Mean Time) time zone, and
- all time zones defined by your operating system.

Daylight saving time indicators

The remaining components of the %TZ specifier are two string indicators *Std* and *Dst*, which are displayed in all generated Calendar slots or period strings when standard time (i.e. no daylight saving time) or daylight saving time is active, respectively. Both indicators must be either quoted strings, or references to scalar string parameters. In addition, a run time error will occur when both indicators evaluate to the same string.

Effect on time slots

When you use the %TZ specifier, the date and time components of the generated time slots of a calendar may differ when you specify different time zones, but do not modify the reference dates specified in the BeginDate and EndDate attributes of the calendar. AIMMS always assumes that reference dates are specified in local time without daylight saving time (i.e. in the 'Local' time zone). Hence, all time slots will be shifted by the time differences between the specified time zone and the 'Local' time zone, plus any additional difference caused by daylight saving time.

Examples

Consider the following four Calendar declarations.

```
Calendar HourCalendarLocal {
    Index           :  hl;
    Unit            :  hour;
    BeginDate       :  "2001-03-25 00";
    EndDate         :  "2001-03-25 06";
    TimeslotFormat  :  "%c%y-%m-%d %H:00";
}
Calendar HourCalendarLocalIgnore {
    Index           :  hi;
    Unit            :  hour;
    BeginDate       :  "2001-03-25 00";
    EndDate         :  "2001-03-25 06";
    TimeslotFormat  :  "%c%y-%m-%d %H:00%TZ('LocalDST')";
}
Calendar HourCalendarLocalDST {
    Index           :  hd;
    Unit            :  hour;
    BeginDate       :  "2001-03-25 00";
    EndDate         :  "2001-03-25 06";
    TimeslotFormat  :  "%c%y-%m-%d %H:00%TZ('LocalDST')|\"\"|\" DST\"|";
}
Calendar HourCalendarUTC {
    Index           :  hc;
    Unit            :  hour;
    BeginDate       :  "2001-03-25 00";
    EndDate         :  "2001-03-25 06";
    TimeslotFormat  :  "%c%y-%m-%d %H:00%TZ('UTC')|\"\"|\" DST\"|";
}
```

Assuming that the 'Local' time zone has an offset of +1 hours compared to the 'UTC' time zone, this will result in the generation of the following time

slots for each of the calendars

```
!   HourCalendarLocal    HourCalendarIgnore   HourCalendarLocalDST     HourCalendarUTC
!   ------------------   ------------------   ---------------------   ------------------
    '2001-03-25 00:00'   '2001-03-25 00:00'   '2001-03-25 00:00'      '2001-03-24 23:00'
    '2001-03-25 01:00'   '2001-03-25 01:00'   '2001-03-25 01:00'      '2001-03-25 00:00'
    '2001-03-25 02:00'   '2001-03-25 02:00'   '2001-03-25 03:00 DST'  '2001-03-25 01:00'
    '2001-03-25 03:00'   '2001-03-25 03:00'   '2001-03-25 04:00 DST'  '2001-03-25 02:00'
    '2001-03-25 04:00'   '2001-03-25 04:00'   '2001-03-25 05:00 DST'  '2001-03-25 03:00'
    '2001-03-25 05:00'   '2001-03-25 05:00'   '2001-03-25 06:00 DST'  '2001-03-25 04:00'
    '2001-03-25 06:00'   '2001-03-25 06:00'   '2001-03-25 07:00 DST'  '2001-03-25 05:00'
```

Note that the time slots generated for `HourCalendarLocal` and `HourCalendarIgnore` are identical (although `'LocalDST'` supports daylight saving time). This is because daylight saving time is ignored when the `%TZ` specifier has no *Std* and *Dst* indicators. The time slots generated for `HourCalendarUTC` do not contain the specified daylight saving time indicator, because the `'UTC'` time zone has no daylight saving time.

33.8 Converting time slots and periods to strings

Converting time slots to strings

The following functions enable conversion between calendar slots and free format strings using the conversion specifiers discussed in the previous section. Their syntax is

- `TimeSlotToString`(*format-string, calendar, time-slot*)
- `StringToTimeSlot`(*format-string, calendar, moment-string*).

The result of the function `TimeSlotToString` is a description of the specified *time-slot* according to *format-string*. The result of `StringToTimeSlot` is the time slot in *calendar* in which the string *moment-string*, specified according to *format-string*, is contained.

Converting timetable periods to strings

With the function `PeriodToString` you can obtain a description of a period in a timetable that consists of multiple calendar slots.

- `PeriodToString`(*format-string, timetable, period*)

The result of the function is a description of the time span covered by a *period* in a horizon according to the specified *timetable* and *format-string*. The *format-string* argument can use period-specific conversion specifiers to generate a description referring to both the beginning and end of the period.

Obtaining the current time

The functions `CurrentToString` and `CurrentToTimeSlot` can be used to obtain the current time. Their syntax is

- `CurrentToString`(*format-string*)
- `CurrentToTimeSlot`(*calendar*)

The function CurrentToString will return the current time according to the specified format string. If you do not use the %TZ specifier in the format string of the CurrentToString function, AIMMS assumes daylight saving time by default. You can change this default behavior by setting the global option current_time_in_LocalDST to 0. The function CurrentToTimeSlot returns the time slot in *calendar* containing the current moment.

33.9 Working with elapsed time

Use of elapsed time

Sometimes you may find it easier to formulate your model in terms of (continuous) elapsed time with respect to some reference date rather than in terms of discrete time periods. For example, for a task in a schedule it is often more natural to store just the start and end time rather than to specify all of the time slots in a calendar during which the task will be executed. In addition, working with elapsed time allows you to store time references to any desired accuracy.

Input-output conversion

For data entry or for the generation of reports, however, elapsed time may not be your preferred format. In this event AIMMS offers a number of functions for the conversion of elapsed time to calendar strings (or set elements) and vice versa, using the conversion specifiers described in section 33.7.

Conversion to calendar elements

The following functions allow conversion between elapsed time and time slots in an existing calendar. Their syntax is

- MomentToTimeSlot(*calendar, reference-date, elapsed-time*)
- TimeSlotToMoment(*calendar, reference-date, time-slot*).

The *reference-date* argument must be a time slot in the specified calendar. The *elapsed-time* argument is the elapsed time from the *reference-date* measured in terms of the calendar's unit. The result of the function MomentToTimeSlot is the time slot containing the moment represented by the reference date plus the elapsed time. The result of the function TimeSlotToMoment is the elapsed time from the reference date to the value of the *time-slot* argument (measured in the calendar's unit).

Conversion to calendar strings

The following functions enable conversion between elapsed time and free format strings. Their syntax is

- MomentToString(*format-string, unit, reference-date, elapsed-time*)
- StringToMoment(*format-string, unit, reference-date, moment-string*).

The *reference-date* argument must be provided in the fixed format for reference dates, as described in Section 33.2. The *moment-string* argument must be a period in the format given by *format-string*. The *elapsed-time* argument is the elapsed time in *unit* with respect to the *reference-date* argument. The

result of the function MomentToString is a description of the corresponding moment according to *format-string*. Strictly spoken, the *unit* argument in MomentToString is only required when the option elapsed_time_is_unitless (see below) is set to on, and, consequently, *elapsed-time* is unitless. In the case that elapsed_time_is_unitless is set to off (the default), you are advised to set the *unit* argument equal to the associated unit of the *elapsed-time* argument. The result of the function StringToMoment is the elapsed time in *unit* between *reference-date* and *moment-string*.

Example

```
moment := MomentToString("%c%y-%Am|AllAbbrMonths|-%d (%sAw|AllWeekdays|) %H:%M",
                         [hour], "1996-01-01 14:00", 2.2 [hour] );
          ! result : "1996-Jan-01 (Monday) 16:12"

elapsed := StringToMoment("%c%y-%Am|AllAbbrMonths|-%d (%sAw|AllWeekdays|) %H:%M",
                          [hour], "1996-01-01 14:00", "1996-Jan-01 (Monday) 16:12" );
          ! result : 2.2 [hour]
```

Obtaining the current time

The function CurrentToMoment can be used to obtain the elapsed time since *reference-date* in the specified *unit* of the current time. Its syntax is

- CurrentToMoment(*unit*, *reference-date*).

Unitless result

By default, the result of the functions TimeSlotToMoment, StringToMoment and CurrentToMoment will have an associated unit, namely the unit specified in the *unit* argument. In addition, AIMMS expects the *elapsed-time* argument in the function MomentToTimeSlot and MomentToString to be of the same unit as its associated *unit* argument. If you want the result or arguments of these functions to be unitless, you can accomplish this by setting the compile time option elapsed_time_is_unitless to on. Note, however, that any change to this option affects all calls to these function throughout your model.

33.10 Working in multiple time zones

Multiple time zones

If your application uses external time-dependent data supplied by users worldwide, you are faced with the problem of converting all dates and times to the calendar in which your application is set up to run. The facilities described in this section help you with that task.

Obtaining time zone info

With the functions TimeZoneOffset, DaylightSavingStartDate and DayLightSavingEndDate you can obtain various time zone specific characteristics.

- TimeZoneOffset(*FromTimeZone*, *ToTimeZone*)
- DaylightSavingStartDate(*year*, *TimeZone*)
- DaylightSavingEndDate(*year*, *TimeZone*)

The function TimeZoneOffset returns the offset in minutes between the time zones *FromTimeZone* and *ToTimeZone*. You can use the function DaylightSa-

vingStartDate and DaylightSavingEndDate to retrieve the reference dates, within a particular time zone, corresponding to the begin and end date of daylight saving time.

Example

```
offset    := TimeZoneOffset('UTC', 'Eastern Standard Time');
             ! result: -300 [min]
startdate := DaylightSavingStartDate( '2001', 'Eastern Standard Time');
             ! result: "2001-04-01 02"
```

Converting reference dates

With the function ConvertReferenceDate you can convert reference dates to different time zones. Its syntax is:

- ConvertReferenceDate(*ReferenceDate*, *FromTimeZone*, *ToTimeZone*[, *IgnoreDST*])

With the function you can convert a reference date *ReferenceDate*, within the time zone *FromTimeZone*, to a reference date within the time zone *ToTimeZone*. If the optional argument *IgnoreDST* is set to 1, AIMMS will ignore daylight saving time for both input and output reference dates (see also Section 33.7.4). By default, AIMMS will not ignore daylight saving time.

Example

```
UTCdate := ConvertReferenceDate("2001-05-01 12", 'Local', 'UTC');
           ! result: "2001-05-01 11"
```

Conventions and time zones

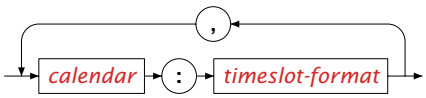
When your application needs to read data from or write data to a database or file with dates not specified in local time, some kind of conversion of all dates appearing in the data source is required during the data transfer. To support you with such date conversions, AIMMS allows you to override the default timeslot format of one or more calendars in your model through a unit Convention (see also Section 32.8).

The TimeslotFormat attribute

In the TimeslotFormat attribute of a Convention you can specify the time slot format (see also Section 33.7) to be used for each Calendar while communicating with an external source. The syntax of the TimeslotFormat attribute is:

Syntax

timeslot-format-list :



Use in graphical user interface

When an active Convention overrides the time slot format for a particular calendar, all calendar elements will be displayed according to the time slot format specified in the Convention. The time slot format specified in the calendar itself is then ignored.

Reading and writing to file

If you use a Convention to override the time slot format while writing data to a data file, report file or listing file, AIMMS will convert all calendar elements to the format specified in the TimeslotFormat for that calendar in the Convention prior to writing the data. Similarly, when reading data from a data file, AIMMS will interpret the calendar slots present in the file according to the specified time slot format, and convert them to the corresponding calendar elements in the model.

Database communication...

When communicating calendar data to a database, there is one additional issue that you have to take into account, namely the data type of the column in which the calendar data is stored in the database table. If calendar data is stored in a string column, AIMMS will transfer data according to the exact date-time format specified in the TimeslotFormat attribute of the Convention, including any indicators for specifying the time zone and/or daylight saving time.

... for date-time columns

However, if the data type of the column is a special date-time data type, AIMMS will always communicate calendar slots as reference dates, which is the standard date-time string representation used by ODBC. In translating calendar slots to reference dates, AIMMS will adhere to the format specified in the TimeslotFormat attribute of either the Calendar itself, or as overridden in the currently active Convention.

Generated reference dates

The reference dates are generated according to the time zone specified in the %TZ specifier of the currently active TIMESLOT SPECIFIER. These dates always ignore daylight saving time (i.e. shift back by one hour if necessary), as daylight saving time cannot be represented in the fixed reference date format. Specifiers in the TIMESLOT FORMAT other than the %TZ specifier are not used when mapping to date-time values in a database. If you do not specify a %TZ specifier in the TIMESLOT FORMAT, AIMMS will assume that all date-time columns in a database are represented in the 'Local' time zone (the default).

Example

Consider the calendar HourCalendarLocalDST defined below.

```
Calendar HourCalendarLocalDST {
    Index           :  hd;
    Unit            :  hour;
    BeginDate       :  "2001-03-25 00";
    EndDate         :  "2001-03-25 06";
    TimeslotFormat  :  "%c%y-%m-%d %H:00%TZ('LocalDST')|\"\"|\" DST\"|";
}
```

If you want to transfer data defined over this calendar with a database table in which all dates are represented in UTC time, you should define a convention defined as follows.

```
Convention UTCConvention {
    TimeslotFormat  : {
        HourCalendarLocalDST : "%c%y-%m-%d %H:00%TZ('UTC')"
    }
}
```

When this convention is active, AIMMS will represent all calendar slots of the calendar HourCalendarLocalDST as reference dates according to the UTC time zone, by shifting as many hours as dictated by the local time zone and/or daylight saving time if applicable. Hence, when you use the convention UTCConvention during data transfer with the database, all calendar data slots will be in the expected format.

Chapter 34

The AIMMS Programming Interface

This chapter

In addition to the capability to call external procedures and functions from within an AIMMS application, AIMMS also provides a generic Application Programming Interface (API). This chapter describes the semantics of the complete AIMMS API, and provides an extended example to familiarize you with its use. In addition, it discusses the concurrency aspects when multiple external applications are controlling a single AIMMS session. Note that this chapter assumes that you have some basic knowledge of the C programming language.

34.1 Introduction

Communication scenario's

One can think of several scenario's in which a path of communication needs to be set up between AIMMS and an external software component. The two most common scenario's are listed below.

- You have a collection of functions within an external DLL which you want to use to perform certain data manipulations within your AIMMS model through calls to an `ExternalProcedure` or `ExternalFunction`.
- From within your own application you want to open an AIMMS project, pass input data to it, solve an optimization model, and retrieve the solution.

Exchanging data

The most straightforward method to set up communication between AIMMS and an external DLL is by calling an external procedure or function from within your model. If, during such a call, the data of one or more scalar or low-dimensional indexed identifiers need to be passed to the DLL, the easiest way to exchange this data is by passing either a single scalar value or a (dense) array of scalar values as arguments to the corresponding DLL function. For higher-dimensional identifiers, however, the memory requirements for passing array arguments may grow out of hand, and additional control may be needed.

Application-controlled execution

With only the possibility to call external procedures and functions from within an AIMMS model, however, you have no possibility, from within an external application, to

- open an AIMMS project,

- initiate the exchange of data, or
- execute one or more procedures in your model.

The AIMMS *API*

The AIMMS Application Programming Interface (API) described in this chapter addresses both the drawbacks associated with dense data transfer, and the need to control the execution of an AIMMS model from within an external application. Figure 34.1 provides a schematic overview of the capabilities to communicate using both the concept of external procedures and functions and the AIMMS API. Left-to-right arrows are implemented through external proce-

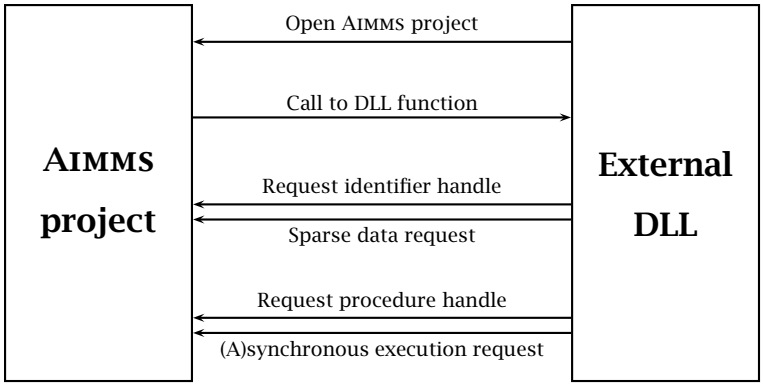


Figure 34.1: Interaction between AIMMS and an external DLL

dure and function calls within your model, while all right-to-left arrows are provided for by the AIMMS API.

Handles

Central to the AIMMS API is the concept of *handles*. Handles are represented by unique integer numbers, and provide indirect access to named identifiers and procedures within an AIMMS model. Access to the associated objects within the model is through the functions of the API. With every identifier or procedure in the model, multiple handles can be associated, each of which may behave differently when passed to a function in the AIMMS API depending on its declaration or on the sequency of API functions previously applied to it (e.g. during sparse data retrieval). Handles can be created by AIMMS and passed as arguments to a DLL function, or can be created from within an external application.

API functions

Through the functions in the AIMMS API, you can initiate further actions on a given identifier or procedure handle from within an external application. More specifically, the API functions allow you to

- obtain information about identifiers in the model, such as domain, range and type,

- set up sparse data communication between an identifier in the AIMMS model and an external application, and
- request either synchronous or asynchronous execution of a procedure within the AIMMS model.

C interface

AIMMS only provides a C interface to the functions in its API. When you are using a different language which requires a different interface, you should implement the required interface yourself in C/C++ or in a compatible language.

Single example

This remainder of this section will provide you with a simple `ExternalProcedure` declaration and the associated C function that illustrates the basic use of the AIMMS API and further familiarizes you with the basic concepts. Because of the many API functions and their interdependence, it is practically impossible to provide illustrative examples for each API function separately in the context of the this language reference. Therefore, the subsequent sections will only explain the semantics of each separate API function.

Example: printing identifier info

The following C function accepts the name of an AIMMS identifier with double-valued values. It queries AIMMS for a handle to that identifier, the corresponding domain and all associated values. For the sake of conciseness, the DLL function does not check all return values passed by the AIMMS API functions.

```
#include <stdio.h>
#include <string.h>
#include <aimmsapi.h>

DLL_EXPORT(void) print_double_aimms_identifier_info(char *name) {
   int  handle, full, sliced, domain[AIMMSAPI_MAX_DIMENSION],
        tuple[AIMMSAPI_MAX_DIMENSION], storage, i;
   char file[256], buffer[256];
   FILE *f;

   AimmsValue  value;
   AimmsString strvalue;

   /* Create a handle associated with the identifier name passed */
   AimmsIdentifierHandleCreate(name, NULL, NULL, 0, &handle);

   /* Get the dimension, domain and storage type of the identifier
      associated with the handle */
   AimmsAttributeDimension (handle, &full, &sliced);
   AimmsAttributeRootDomain(handle, domain);
   AimmsAttributeStorage   (handle, &storage);

   if ( storage != AIMMSAPI_STORAGE_DOUBLE ) return;

   /* Open a file consisting of the identifier name with the extension .def,
      and print the identifier's name and dimension */

   strcpy(file, name); strcat(file, ".def");
   if ( ! (f = fopen(file, "w")) ) return;
   fprintf(f, "Identifier name: %s\n", name);
   fprintf(f, "Dimension      : %d\n", full);
```

```
    /* Prepare strvalue to hold the locally declared buffer */
    strvalue.String = buffer;

    /* Print a header containing the names of the domain sets */
    fprintf(f, "\nData values    : \n");
    for ( i = 0; i < full; i++ ) {
        strvalue.Length = 256;
        AimmsAttributeName(domain[i], &strvalue); fprintf(f, "%17s", buffer);
    }
    fprintf(f,"%16s\n","Double value");
    for ( i = 0; i < full; i++ ) fprintf(f, "%17s", "----------------");
    fprintf(f,"\n");

    /* Print all tuples with nondefault data values */
    AimmsValueResetHandle(handle);
    while ( AimmsValueNext(handle, tuple, &value) ) {
        for ( i = 0; i < full; i++ ) {
            strvalue.Length = 256;
            AimmsSetElementToName(domain[i], tuple[i], &strvalue);
            fprintf(f,"%17s", buffer);
        }
        fprintf(f,"%17.5f\n", value.Double);
    }

    fclose(f);
}
```

If the DLL function is part of a DLL "Userfunc.dll", then it can be called from within AIMMS by the following ExternalProcedure declaration.

```
ExternalProcedure PrintParameterInfo {
    Arguments  : (param);
    DLLName    : "Userfunc.dll";
    BodyCall   : print_double_aimms_identifier_info(string scalar: param);
}
```

Its only argument is an element parameter into the predefined set AllIdentifiers. It can therefore be called with any identifier name.

```
ElementParameter param {
    Range      : AllIdentifiers;
    Property   : input;
}
```

Call example

Consider a two-dimensional parameter TransportCost(i,j) which contains the following data.

```
    TransportCost := DATA TABLE
                     Rotterdam       Antwerp         Berlin
    !                ---------       ---------       ---------
    Amsterdam         1.00            2.50           10.00
    Rotterdam                         1.20           10.00
    Antwerp                                          11.00
    ;
```

Then the procedure call PrintParameterInfo('TransportCost') will result in the creation of a file TransportCost.def with the following contents.

```
Identifier name: TransportCost
Dimension      : 2

Data values    :
Cities            Cities           Double value
----------------- ---------------- ---------------
Amsterdam         Rotterdam                1.00000
Amsterdam         Antwerp                  2.50000
Amsterdam         Berlin                  10.00000
Rotterdam         Antwerp                  1.20000
Rotterdam         Berlin                  10.00000
Antwerp           Berlin                  11.00000
```

aimmsapi.h header file

The prototypes of all the available AIMMS API functions, as well as all C macro definitions that are relevant for the execution of the API functions are provided in a single header file aimmsapi.h. You should include this header file in all your source files that make use of the AIMMS API functions.

Two flavors of API

The AIMMS API functions are provided in two flavors: ASCII and Unicode. For each of the functions mentioned in this chapter, there is an implementation postfixed with A for the ASCII flavor, and an implementation postfixed with W for the Unicode flavor. For instance, when UNICODE is defined, a call to AimmsIdentifierHandleCreate will be mapped to the implemented function AimmsIdentifierHandleCreateW. For the Unicode flavor, on Windows, double-byte character arrays are used to communicate strings corresponding to the UTF-16LE character encoding. For the Unicode flavor, on Linux, quadruple-byte character arrays are used to communicate strings corresponding to the UTF-32LE character encoding. This corresponds to the wchar_t* type on both platforms. Please make sure that the option external_string_character_encoding is set to corresponding encoding. For the ASCII flavor, both on Windows and on Linux, multibyte character arrays are used, and the encoding is determined by the option external_string_character_encoding.

libaimms3.lib import library

The AIMMS API functions are provided in the form of a Visual C/C++ import library libaimms3.lib to the libaimms3.dll DLL, which can be included in the link step of your external AIMMS DLL. When you are using the Visual C/C++ compiler, this import library will take care that all the relevant API functions are imported from the AIMMS executable when your AIMMS application loads the external DLL. For other compilers, you should consult the compiler documentation on how to import the functions in libaimms3.dll into your program.

Return values

All AIMMS API functions provide an integer return value. When the requested operation has succeeded, the value AIMMSAPI_SUCCESS is returned. When the operation has failed, AIMMS will return the value AIMMSAPI_FAILURE. In the latter case, you can obtain an error code and string through the API function AimmsAPILastError (see also Section 34.7).

Only identifiers with data

AIMMS will only allow you to pass or create handles for identifier types with which data is associated, i.e. sets, parameters and variables. In addition, you can pass or create handles to suffices of identifiers as long as the resulting suffix results in a set or parameter.

34.2 Obtaining identifier attributes

Obtaining attributes

For every identifier handle passed to (or created from within) an external function, AIMMS can provide additional attributes that are either related to the declaration of the identifier associated with the handle, or to the particular identifier slice that was passed as an argument in the external function call. Table 34.1 lists all AIMMS API functions which can be used to obtain these additional attributes.

`int AimmsAttributeName(int handle, AimmsString *name)`
`int AimmsAttributeType(int handle, int *type)`
`int AimmsAttributeStorage(int handle, int *storage)`
`int AimmsAttributeDefault(int handle, AimmsValue *value)`

`int AimmsAttributeSetUnit(int handle, char *unit, char *convention)` `int AimmsAttributeGetUnit(int handle, AimmsString *unitName)`

`int AimmsAttributeDimension(int handle, int *full, int *slice)`
`int AimmsAttributeRootDomain(int handle, int *domain)`
`int AimmsAttributeDeclarationDomain(int handle, int *domain)`
`int AimmsAttributeCallDomain(int handle, int *domain)`
`int AimmsAttributeRestriction(int handle, int *domainhandle)`
`int AimmsAttributeSlicing(int handle, int *slicing)`
`int AimmsAttributePermutation(int handle, int *permutation)`

`int AimmsAttributeFlagsSet(int handle, int flags)`
`int AimmsAttributeFlagsGet(int handle, int *flags)`
`int AimmsAttributeFlags(int handle, int *flags)`

`int AimmsAttributeElementRange(int handle, int *sethandle)`

Table 34.1: AIMMS API functions for obtaining handle attributes

Identifier name and type

With the functions `AimmsAttributeName` and `AimmsAttributeType` you can request the name and identifier type of the identifier associated with a handle. AIMMS passes the name of an identifier through an `AimmsString` structure (explained below). AIMMS only allows handles for identifier types with which data can be associated. More specifically, AIMMS distinguishes the following identifier types:

- simple root set,
- simple subset,
- relation,
- indexed set,
- numeric parameter,
- element parameter,
- string parameter,

- unit parameter,
- variable,
- element variable.

When the handle refers to a suffix of an identifier, the suffix type is appended to the identifier name separated by a dot.

Storage type

In addition to the identifier type, AIMMS also associates a storage type with each handle. It is the data type in which AIMMS expects the data values associated with the handle to be communicated. The function AimmsAttributeStorage returns the storage type. The possible storage types are:

- double (double),
- integer (int),
- binary (int, but only assuming 0-1 values),
- string (char *).

The complete list of identifier and storage type values returned by these functions can be found in the header file aimmsapi.h.

Default value

With the function AimmsAttributeDefault you can obtain the default value of the identifier associated with a handle. The default value can either be a double, integer or string value, depending on the storage type associated with the handle. Below you will find the convention used by AIMMS to pass such storage type dependent values back and forth.

Setting and getting units

Through the functions AimmsAttributeGetUnit and AimmsAttributeSetUnit you can get and set the units of measurement (see also Chapter 32) that will be used when passing data from and to AIMMS. When setting the unit to be used, you can specify both a unit and a convention. AIMMS will parse the given unit expression and use the specified convention to compute the appriopriate multiplication factors between the internal and external representation of the data of the identifier at hand.

Passing integer, double or string values

All transfer of integer, double or string values takes place through the record structures AimmsString and AimmsValue defined as follows.

```
typedef struct AimmsStringRecord {
  int   Length;
  char *String;
} AimmsString;
```

```
typedef union AimmsValueRecord {
  double      Double;
  int         Int;
  struct {
    int       Length;
    char     *String;
  }
} AimmsValue;
```

When value is such a structure, you can obtain an integer, double or string value through value.Int, value.Double or value.String, respectively.

Passing string lengths

For strings, you must set value.Length to the length of the string buffer passed through value.String before calling the API function. When AIMMS fills the value.String buffer, the actual length of the string passed back is assigned to value.Length. When the actual string length exceeds the buffer size, AIMMS truncates the string passed back through value to the indicated buffer size, and assigns the length of the actual string to value.Length.

Identifier dimensions

For each handle you can obtain the dimension of the associated identifier by calling the function AimmsAttributeDimension. The function returns:

- the *full* dimension of the identifier as given in its declaration, and
- the *slice* dimension, i.e. the resulting dimension of the actual identifier slice associated with the handle.

AIMMS uses tuples of length equal to the full dimension whenever information is communicated regarding the index domain of a handle or its slicing. When explicit data values associated with a handle are passed using the AIMMS API functions discussed in Section 34.4, AIMMS communicates such values using tuples of length equal to the slice dimension.

Set dimensions

For all data communication with external DLLs AIMMS considers sets to be represented by binary indicator parameters indexed over their respective root sets. For all elements in these root sets, such an indicator parameter assumes the value 1 if a root set element (or tuple of root set elements) is contained in the set at hand, or 0 otherwise. Since the default of these indicator parameters is 0, AIMMS only needs to communicate the nonzero values, i.e. exactly the tuples that are actually contained in the set. In connection with this representation, AIMMS returns the following (full or slice) dimensions for sets:

- the dimension of a *simple set* is 1,
- the dimension of a *relation* is the dimension of the Cartesian product of which the relation is a subset,
- the dimension of an *indexed set* is the dimension of the index domain of the set plus 1.

Identifier domains

The functions AimmsAttributeRootDomain, AimmsAttributeDeclarationDomain and AimmsAttributeCallDomain can be used to obtain an integer array containing handles to domain sets for every dimension of the identifier at hand. These domains play a different role in the sparse data communication, as explained below.

Root domain handles

The function AimmsAttributeRootDomain returns an array of handles to the respective root sets associated with the index domain specified in the identifier's declaration. You need these handles, for instance, to obtain a string representation of the element numbers returned by the data communication AIMMS API functions discussed in Section 34.4.

Declaration domain handles

The function AimmsAttributeDeclarationDomain returns an array of handles to the respective domain sets specified in the identifier's declaration. These domain sets can be equal to their corresponding root sets, or to subsets thereof. AIMMS will only pass data values for element tuples in the declaration domain, unless you have specified the raw translation modifier (see also Section 11.2) for a handle argument, or have created the handle yourself with the raw flag set (see also Section 34.3).

Call domain handles

The function AimmsAttributeCallDomain returns an array of handles to the particular subsets of the root sets (as returned in the root domain of the handle) to which data communication is restricted for this handle. The call domain can be different from the global domain if an actual external argument has been restricted to a subdomain of the root set in an external call (see also Section 10.3), or if you have created the handle with an explicit call domain yourself (see also Section 34.3). AIMMS will only pass data values associated with element tuples in just the call domain (raw flag set), or in the intersection of the call and declaration domain (raw flag not set).

Domain restriction

With the function AimmsAttributeRestriction you can obtain a handle to the global domain restriction of an indexed identifier as specified in its declaration and (dynamically) maintained by AIMMS as necessary. You may want to use this handle in conjunction with raw handles (explained in Section 34.4) to verify whether a particular element satisfies its domain restriction.

Example

Consider the following set and parameter declarations.

```
Set S_0 {
    Index        : i_0;
}
Set S_1 {
    SubsetOf     : S_0;
    Index        : i_1, j_1;
}
```

```
Set S_2 {
    SubsetOf     : S_1;
    Index        : i_2;
}
Parameter p {
    IndexDomain  : i_0;
}
Parameter q {
    IndexDomain  : (i_1, j_1) | p(i_1);
}
```

A handle to (in AIMMS notation) `q(i_1, i_2)` will return handles to

- `S_0` and `S_0` for the respective root domains,
- `S_1` and `S_1` for the respective declaration domains,
- `S_1` and `S_2` for the respective call domains, and
- `p(i_1)` for the domain restriction.

Slicing

As discussed in Section 10.3, the actual arguments in a procedure or function call can be slices of higher-dimensional identifiers within your model. When the slice dimension of a handle in an external call is less then its full dimension, you can use use the function `AimmsAttributeSlicing` to find out which dimensions of the associated AIMMS identifier have been sliced, and to which elements. The function returns an integer array containing, for every dimension, the element number (within the associated root set) to which the corresponding domain has been sliced, or the number `AIMMSAPI_NO_ELEMENT` if no slicing took place.

Domain permutations

Through the function `AimmsAttributePermutation` you can obtain the permutation of a permuted handle created with the function `AimmsAttributeHandleCreatePermuted`. The output `permutation` argument must be an integer array of length equal to the full dimension of the identifier. AIMMS returns the following values:

- if a dimension of the handle is sliced, the corresponding position in the `permutation` array will be 0,
- if a dimension is not sliced, the corresponding position in the `permutation` array will contain the sliced position (starting at 1, and numbered from 1 to the handle's slice dimension)
  - in which AIMMS will store elements of the corresponding dimension in a `tuple` returned by the functions `AimmsValueNext` and `AimmsValueNextMulti`, or
  - in which AIMMS expects such elements in calls to the functions `AimmsValueSearch` and `AimmsValueRetrieve`.

Getting ordered, special, raw and read-only flags

By specifying the input-output type and the ordered, retainspecials, elementsasordinals or raw translation modifiers for arguments in an external call (see also Section 11.2), you can influence the manner in which data is passed to an external function. With the AIMMS API function AimmsAttributeFlagsGet you obtain the active set of flags indicating whether

- the data associated with a handle is passed ordered (ordered flag),
- special values are passed unchanged or are translated (retainspecials flag),
- element tuples are passed by their element numbers (elementsasordinals flag),
- inactive data is passed (raw flag), and
- you can make assignments to the handle (input-output type).

The result is the *bitwise or* function of the individual flag values as defined in the aimmsapi.h header file.

Setting flags

Through the function AimmsAttributeFlagSet you can modify the flag settings for an existing handle. Note that the result of calls to AimmsValueNext may become unpredictable after modifying the ordered flag. In such a case, you are advised to reset the handle through the function AimmsHandleReset.

Element range

When a handle is associated with an element parameter within your application, you can use the function AimmsAttributeElementRange to obtain a handle to the set constituting the element range of the element parameter. You need this handle, for instance, when you want to obtain a string representation of the element numbers within the element range communicated by AIMMS in the AIMMS API functions discussed Section 34.4.

34.3 Managing identifier handles

Creation and data control

AIMMS offers the capability to dynamically create and delete handles to any desired identifier slice over any desired local subdomain from within a DLL. In addition, a subset of the AIMMS data control operators (as discussed in Section 25.3) can be called from within external DLLs. Table 34.2 lists all available AIMMS API functions for creating handles and performing data control operations.

Creating a handle

You can use the function AimmsIdentifierHandleCreate to dynamically create a handle to (a slice of) an AIMMS identifier or a suffix thereof within an external function or procedure. You can restrict the scope of a handle by

- specifying a *call* domain to which you want to restrict the handle, or
- by *slicing* one or more dimensions of the identifier.

```
int AimmsIdentifierHandleCreate(char *name, int *domain, int *slicing,
                                int flags, int *handle)
int AimmsIdentifierHandleCreatePermuted(char *name, int *domain, int *slicing,
                                        int *permutation, int flags, int *handle)
int AimmsIdentifierHandleDelete(int handle)
int AimmsIdentifierEmpty(int handle)
int AimmsIdentifierCleanup(int handle)
int AimmsIdentifierUpdate(int handle)
int AimmsIdentifierDataVersion(int handle, int *version)
```

Table 34.2: AIMMS API functions for handle management

Obtaining a suffix

If you want a handle to an identifier itself, the name passed to AimmsIdentifierHandleCreate should just be the identifier name. If you want a handle to a suffix of an identifier, you should pass the name of the identifier followed by a dot and the suffix name. Thus, for instance, you should pass the name "Transport.ReducedCost" if you want a handle to the reduced costs of the variable Transport.

Specifying a call domain

When you want to create a handle over the full root domain, you can simply pass a null pointer for the domain argument. If you want to specify an additional call domain, you must pass an integer array of length equal to the identifier's full dimension, each element containing a handle to the set to which you want to restrict the domain. If the raw flag is not set, passing a null pointer for the domain handle will effectively restrict the declaration domain of the identifier at hand, because of the semantics of the raw flag (see also Sections 34.2 and 34.4).

Specifying a slice

When you want to create a handle over the full dimension of an identifier, you can simply pass a null pointer for the slicing argument. If you want to create a handle to a slice, you must pass an integer array of length equal to the identifier's full dimension, each element containing either a null element for all the domains that you do not want to slice, or the element number of the element to which you want to slice.

Modification flags

With the flags argument in a call to AimmsIdentifierHandleCreate you can specify which modification flags should be set for the handle to be created. The format of the flags argument is the same as in the function AimmsAttributeFlags discussed in the previous section.

Creating a permuted handle

With the function AimmsIdentifierHandleCreatePermuted you can obtain a handle to a multidimensional identifier, for which the order in which element tuples are returned is permuted. Handles created by AimmsIdentifierHandleCreatePermuted are always read-only, i.e. cannot be used in the functions AimmsValueAssign and AimmsValueAssignMulti. The permutation argument must be

specified according to the rules explained for the function AimmsAttributePermutation.

Example

Consider an identifier p(i,j,k,l) for which you want to retrieve the values as if the identifier were defined as p(k,i,l,j). To retrieve all values of p in this order, the permutation array must be specified as [2,4,1,3].

Deleting handles

With the function AimmsIdentifierHandleDelete you can delete a dynamically created handle that is no longer needed. The function fails when you try to delete a handle that was passed as an argument to the DLL. After deletion the handle can no longer be used in conjunction with any AIMMS API function.

Empty, cleanup and update handles

The AIMMS API functions

- AimmsIdentifierEmpty,
- AimmsIdentifierCleanup, and
- AimmsIdentifierUpdate

can be called to perform the identical actions on a set or identifier (slice) from within an external DLL as can be accomplished by the data control operators EMPTY, CLEANUP and UPDATE from within AIMMS, respectively. The function AimmsIdentifierEmpty will empty the particular slice and subdomain of the identifier associated with the handle. The other two functions will cleanup or update the *entire* data set of the identifier associated with the handle, regardless of the specified slicing and local domain.

Data version

For every identifier within your model AIMMS maintains a version number of the data associated with the identifier. This number is incremented each time a data value of the identifier has been changed. You can use the function AimmsIdentifierDataVersion to retrieve this version number, for instance, to verify whether the data has changed relative to the last time you retrieved it.

Checking for global data changes

When you apply the function AimmsIdentifierDataVersion to the predefined handle value AIMMSAPI_MODEL_HANDLE, AIMMS will return a data version number based on the cases and datasets currently active within the model. AIMMS will update this number as soon as the combined configuration of the active case and/or datasets within the model has changed, as well as after a call to the CLEANDEPENDENTS operator. A change in this global data version number is a good indication that the contents of all or a number of domain sets may have changed, and must be retrieved again.

34.4 Communicating individual identifier values

Communicating identifier values

With every identifier handle AIMMS lets you retrieve all associated nondefault data values on an element-by-element basis. In addition, AIMMS lets you search whether a nondefault value exists for a particular element tuple, and make assignments to individual element tuples. Table 34.3 lists all the available AIMMS API functions for this purpose.

`int AimmsValueCard(int handle, int *card)`

`int AimmsValueResetHandle(int handle)`
`int AimmsValueSearch(int handle, int *tuple, AimmsValue *value)`
`int AimmsValueNext(int handle, int *tuple, AimmsValue *value)`
`int AimmsValueNextMulti(int handle, int *n, int *tuples, AimmsValue *values)`

`int AimmsValueRetrieve(int handle, int *tuple, AimmsValue *value)`
`int AimmsValueAssign(int handle, int *tuple, AimmsValue *value)`
`int AimmsValueAssignMulti(int handle, int n, int *tuples, AimmsValue *values)`

`int AimmsValueDoubleToMapval(double value, int *mapval)` `int AimmsValueMapvalToDouble(int mapval, double *value)`

Table 34.3: AIMMS API functions for sparse data communication

Cardinality

The function `AimmsValueCard` returns the cardinality of a handle, i.e. the number of nondefault elements of the associated identifier slice. You can call this function, for instance, when you need to allocate memory for the data structures in your own code before actually retrieving the data.

Retrieving nondefault values

The functions `AimmsValueResetHandle`, `AimmsValueSearch` and `AimmsValueNext` retrieve nondefault values associated with a handle on an element-by-element basis.

- The function `AimmsValueResetHandle` resets the handle to the position just before the first nondefault element.
- The function `AimmsValueSearch` expects an input tuple of element numbers (in the slice domain), and returns the first tuple for which a nondefault value exists on or following the input tuple.
- The function `AimmsValueNext` returns the first nondefault element directly following the element returned by the last call to `AimmsValueNext` or `AimmsValueSearch`, or the first element if the function `AimmsValueResetHandle` was called last. The function fails when there is no such element.

By calling `AimmsValueResetHandle` and subsequently `AimmsValueNext` it is possible to retrieve *all* nondefault values. By calling the function `AimmsValueSearch` you can directly skip to a particular element tuple if you have found that the intermediate tuples are not interesting anymore, and continue from there.

No scalar handles

The functions AimmsValueResetHandle, AimmsValueNext and AimmsValueSearch do not accept handles to scalar (i.e. 0-dimensional) identifier slices. To retrieve and assign scalar values you should use the functions AimmsValueRetrieve and AimmsValueAssign explained below.

Unordered versus ordered retrieval

The particular element returned by the functions AimmsValueSearch and AimmsValueNext may differ depending on the setting of the ordered flag for the handle. If the handle has been created unordered (default), the values returned successively are ordered by increasing element number in a right-to- left tuple order. If the handle has been created ordered, AIMMS will return values in accordance with the ordering principles imposed on all *local* tuple domains.

Raw data retrieval

By default, AIMMS will only pass values for element tuples that lie within the *current* contents of the intersection of the call domain and declaration domain of an identifier. Thus, the values that get passed may depend on a dynamically changing domain restriction that is part of the index domain in the declaration of an identifier. When the raw modification flag is set for a handle, AIMMS will pass *all* available data values in the call domain, regardless of the domain restrictions.

Return tuple and value

All data retrieval functions return a tuple and the associated nondefault value. The interpretation of the value argument for all possible storage types was discussed on page 580. The tuple argument must be an integer array of length equal to the *slice* dimension of the handle. Upon success, the tuple contains the element numbers in the *global* domain sets for every non-sliced dimension.

Element or ordinal numbers

By setting the flag elementsasordinals during the creation of a handle, you can modify the default tuple representation. If this flag is set, the tuples returned by AIMMS will contain ordinal numbers corresponding to the respective call domains associated with the handle. Similarly, AIMMS expects tuples that are passed to it, to contain ordinal numbers as well, when this flag is set.

Rationale

While at first sight the choice for representing tuples by their element numbers in the global domain of a handle may seem less convenient than ordinal numbers in its call domain, you must be aware that the latter representation is not invariant under changes in the contents of the call domain. Alternatively to setting the flag elementsasordinals, you can also convert the returned element numbers into these formats using the AIMMS API functions discussed in Section 34.5.

Value types

The expected storage type of the data values returned by the data retrieve functions can be obtained using the function AimmsAttributeStorage. The possible storage types for the various identifier types are listed below:

- numeric parameters and variables return double or integer values,

- all set types return binary values,
- element parameters return integer element numbers, and
- string and unit parameters return string values.

Element parameter values

The element numbers returned for element parameters are relative to the set handle returned by the function AimmsAttributeElementRange. You can use the AIMMS API functions of Section 34.5 to obtain the associated ordinal numbers or string representations.

Set values

For sets (either simple, relation or indexed), the data retrieval functions return the binary value 1 for just those elements (or element tuples) that are contained in the set. For indexed sets, AIMMS returns tuples for which the last component is the element number of an element contained in the set slice associated with all but the last tuple components.

Converting special numbers

When a handle to a numeric parameter or variable has been created with the special flag set, the data retrieval functions will pass any special number value associated with the handle as is (see also Sections 11.2 and 34.2). AIMMS represents special numbers as double precision floating point numbers outside AIMMS' ordinary range of computation. The function AimmsValueDoubleToMapval returns the MapVal value associated with any double value (see also Table 6.1), while the function AimmsValueMapvalToDouble returns the double representation associated with any type of special number.

Retrieving specific values

The function AimmsValueRetrieve returns the value for a specific element tuple in the slice domain. This value can be either the default value or a nondefault value. The tuple must consist of element numbers in the corresponding domain sets. When the raw flag is not set, the function fails (but still returns the default value of the associated identifier) for any tuple outside of the index domain of the handle. When the raw flag is set, the function fails only when there is no data for the tuple.

Assigning values

The function AimmsValueAssign lets you assign a new value to a particular element tuple in the slice domain. If you want to assign the default value you can either pass a null pointer for value, or a pointer to the appropriate default value. The function fails if you try to assign a value to an element tuple outside the contents of the call domain of the handle. When the raw flag is not set, the function will also fail if the assigned tuple lies outside of the current (active) contents of the declaration domain.

Exchanging multiple values

When a particular identifier handle requires the exchange of a large amount of values, you are strongly encouraged to use the functions AimmsValueNextMulti and AimmsValueAssignMulti instead of the functions AimmsValueNext and AimmsValueAssign. In general, AIMMS can perform the simultaneous exchange of

multiple values much more efficient than the equivalent sequence of single exchanges. For both functions, the tuples array must be an integer array of length n times the *slice* dimension of the handle, while the values array must be the corresponding AimmsValue array of length n.

- In the function AimmsValueNextMulti, AIMMS will fill the tuples array with the respective tuples for which nondefault values are returned in the values array. Upon return, the n argument will contain the actual number of values passed.
- In the function AimmsValueAssignMulti, the tuples array must be filled sequentially with the respective tuples to which the assignments take place via the values array.

When your data transfer involves the addition of a large amount of set elements to an AIMMS set as well, you may also want to consider using the function AimmsSetAddElementMulti (see Section 34.5).

Communicating scalar values

When a handle corresponds to a 0-dimensional (i.e. scalar) identifier slice, you can still use the AimmsValueRetrieve and AimmsValueAssign to retrieve its value or assign a value to it. In this case, the tuple argument is ignored.

Assigning set values

When you want to delete or add an existing element or element tuple to a set, you must assign the value 0 or 1 to the associated tuple respectively. If you want to add a tuple of nonexisting simple elements, you must first add these elements to the corresponding global simple domain sets using the function AimmsSetAddElement discussed below.

34.5 Accessing sets and set elements

Accessing sets

The AIMMS API functions discussed in the previous section allow you to retrieve and assign individual values of (slices of) indexed identifiers associated with tuples of set element numbers used by AIMMS internally. The AIMMS API functions discussed in this section allow you to add elements to simple sets, and let you convert element numbers into ordinal numbers and element names, or vice versa. Table 34.4 presents all set related API functions.

Adding elements to simple sets

The function AimmsSetAddElement allows you to add new element names to a simple set. AIMMS will return with the internal element number assigned to the element, which you can use for further references to the element. The function fails if an element with the specified name already exists, but still sets element to the corresponding element number.

```
int AimmsSetAddElement(int handle, char *name, int *element)
int AimmsSetAddElementMulti(int handle, int n, int *elementNumbers)
int AimmsSetAddElementRecursive(int handle, char *name, int *element)
int AimmsSetRenameElement(int handle, int element, char *name)
int AimmsSetDeleteElement(int handle, int element)
```

```
int AimmsSetElementNumber(int handle, char *name, int allowCreate,
                          int *elementNumber, int *isCreated)
int AimmsSetAddElementMulti(int handle, int n, int *elementNumbers)
int AimmsSetAddElementRecursiveMulti(int handle, int n, int *elementNumbers)
```

```
int AimmsSetElementToOrdinal(int handle, int element, int *ordinal)
int AimmsSetElementToName(int handle, int element, AimmsString *name)
int AimmsSetOrdinalToElement(int handle, int ordinal, int *element)
int AimmsSetOrdinalToName(int handle, int ordinal, AimmsString *name)
int AimmsSetNameToElement(int handle, char *name, int *element)
int AimmsSetNameToOrdinal(int handle, char *name, int *ordinal)
```

Table 34.4: AIMMS API functions for passing set data

Adding element to subsets

If the set is a subset, AIMMS will add the element to that subset only. Thus, the function will fail and return no element number if the corresponding element does not already exist in the associated root set. If the element is present in the root set, but not in the domain of the subset, the functions will fail but still return the element number corresponding to the presented string. With the function AimmsSetAddElementRecursive you can add an element to a subset itself as well as to all its supersets, up to the associated root set.

Renaming set elements

Through the function AimmsSetRenameElement you can provide a new name for an element number associated with an existing element in a set. The change in name does not imply any change in the data previously defined over the element. However, the element will be displayed according to its new name in the graphical user interface, or in data exchange with external data sources.

Deleting set elements

With the function AimmsSetDeleteElement you can delete the element with the given element number from a simple set. If the set is a *root* set, any remaining data defined over the element in subsets parameters and variables will become inactive. To remove such inactive references to the deleted element, you can use the API function AimmsIdentifierCleanup (see also Section 34.3).

Modifying subset contents

Alternatively to applying the functions AimmsSetAddElement and AimmsSetDeleteElement to subsets, you can also use the function AimmsValueAssign to modify the contents of a subset. In that case, you should assign the value 1 to the tuple that should be added to the subset, or 0 to a tuple that should be removed (as discussed in the previous section). The function AimmsValueAssign will also work on indexed sets and relations.

Adding multiple set elements to a simple set...

When, as part of a large data transfer from an external data source to AIMMS, you have to add a large amount of (non-existing) set elements to a simple AIMMS set, the use of the functions AimmsSetElement and AimmsSetElementRecursive may become a performance bottleneck compared to any bulk data transfer of multidimensional data defined over these set elements. The reason for this is that the function AimmsSetElementAdd adds elements one at a time, and may need to extend the internal data structures used to store set data many times, which is a relatively expensive action.

... in an efficient manner

As an alternative, AIMMS offers a different set of functions that combined allow you to add multiple set elements much more efficiently, at the expense of a slightly more complex sequence of actions. The functions are:

- the function AimmsSetElementNumber, which retrieves an existing, or creates a new, set element number for a given element name, *but does not add it yet to any set*, and
- the functions AimmsSetAddElementMulti and AimmsSetAddElementRecursiveMulti, which add multiple elements to a set or a hierarchy of sets simultaneously by passing an array of set element numbers created through the function AimmsSetElementNumber.

Set element representations

The functions

- AimmsSetElementToOrdinal,
- AimmsSetElementToName,
- AimmsSetOrdinalToElement,
- AimmsSetOrdinalToName,
- AimmsSetNameToElement, and
- AimmsSetNameToOrdinal

allow you to convert AIMMS' element numbers into ordinal numbers within a particular subset, and element names and vice versa. The functions will fail when the input representation does not correspond to an existing element.

Ordinal numbers may change

In working with ordinal numbers, you should be aware that ordinal numbers are not invariant under changes to a set. When an element is added to or deleted from a set, or when the ordering of the set has changed, the ordinal numbers of some or all of its elements may have changed. In contrast, the element numbers and names of elements remain constant as long as the case used by the AIMMS model has not changed, or when the CLEANDEPENDENTS operator has not been applied to one or more root sets. You can verify the latter condition with a call to the function AimmsIdentifierDataVersion (see also Section 34.3).

34.6 Executing AIMMS procedures

Running AIMMS *procedures*

The AIMMS API allows you to execute procedures contained in the AIMMS model from within an external application. Both procedures with and without arguments can be executed, and scalar output results can be directly passed back to the external application. Table 34.5 lists the AIMMS API functions offered to obtain procedure handles, to execute AIMMS procedures or to schedule AIMMS procedures for later execution.

```
int AimmsProcedureHandleCreate(char *procedure, int *handle, int *nargs, int *argtype)
int AimmsProcedureHandleDelete(int handle)
int AimmsProcedureRun(int handle, int *argtype, AimmsValue *arglist, int *result)
int AimmsProcedureArgumentHandleCreate(int prochandle, int argnumber, int *arghandle)
int AimmsProcedureAsyncRunCreate(int handle, int *argtype,
                                 AimmsValue *arglist, int *request)
int AimmsProcedureAsyncRunDelete(int request)
int AimmsProcedureAsyncRunStatus(int request, int *status, int *result)
int AimmsExecutionInterrupt(void)
```

Table 34.5: AIMMS API functions for execution requests

Obtaining procedure handles

With the function AimmsProcedureHandleCreate you can obtain a handle to a procedure with the given name within the model. In addition, AIMMS will return the number of arguments of the procedure, as well as the type of each argument. The possible argument types are:

- one of the storage types *double*, *integer*, *binary* or *string* (discussed in Section 34.2) for scalar formal arguments, or
- a *handle* for non-scalar formal arguments.

In addition to indicating the storage type of each argument, the argtype argument will also indicate whether an argument is input, output, or input- output. Through the function AimmsProcedureHandleDelete you can delete procedure handles created with AimmsProcedureHandleCreate.

Calling procedures

You can use the function AimmsProcedureRun to run the AIMMS procedure associated with a given handle. If the AIMMS procedure has arguments, then you have to provide these, together with their types, through the arglist and argtype arguments. The (integer) return value of the procedure (see also pages 139 and 147) is returned through the result argument. If AIMMS is already executing another procedure (started by another thread), the call to AimmsProcedureRun blocks until the other execution request has finished. Section 34.10 explains how to prevent this blocking behavior by obtaining exclusive control over AIMMS.

Passing arguments

For each argument of the AIMMS procedure you have to provide both the type and value through the `argtype` and `arglist` arguments in the call to `AimmsProcedureRun`. You have the following possibilities.

- If the argument is scalar, the argument type can be
  - the storage type returned by the function `AimmsProcedureHandleCreate`, in which case the argument value must be a pointer to a buffer of the indicated type containing the argument, or
  - a handle, in which case the argument value must be a handle associated with a scalar AIMMS identifier (slice) that you want to pass.
- If the argument is non-scalar, the argument type can only be a handle, and the argument value must be a handle corresponding to the identifier (slice) that you want to pass.

If you pass an argument as an identifier handle, this can either be a handle to a global identifier defined within the model, or a local argument handle obtained through a call to the function `AimmsProcedureArgumentHandleCreate` (see below).

Output values

When the input-output type of one or more of the arguments is `inout` or `output`, AIMMS will update the values associated with any handle argument, or, if a buffer containing a scalar value was passed, fill the buffer with the new value of the argument.

Obtaining argument handles

Through the function `AimmsProcedureArgumentHandleCreate` you can obtain a handle to the local arguments of procedures within your model. After creating these handles you can pass them as arguments to the function `AimmsProcedureRun`. The following rules apply.

- After creation, handles created by `AimmsProcedureArgumentHandleCreate` have no associated data.
- If the handle corresponds to an `Input` argument of the procedure, you can supply data prior to calling the procedure, and AIMMS will empty the handle after the execution of the procedure has completed.
- If the handle corresponds to an `InOut` or `Output` argument of the procedure, AIMMS will not empty the handle after completion of the procedure. If you want to supply data to a handle corresponding to an `InOut` argument in subsequent calls, you have to make sure to empty the handle (through the function `AimmsIdentifierEmpty`) prior to supplying the input data.

Requesting asynchronous execution

With the function `AimmsProcedureAsyncRunCreate` you can request asynchronous execution of a particular AIMMS procedure. The function returns an integer request handle for further reference. AIMMS will execute a requested procedure as soon as there are no other execution requests currently being executed or waiting to be executed. *Note that you should make sure that the `AimmsValue` array passed to* AIMMS *stays alive during the asynchronous execution of the*

procedure. Failure to do so, may result in illegal memory references during the actual execution of the AIMMS procedure. This is especially true when the array contains references to scalar integer, double or string `InOut` or `Output` buffers within your application to be filled by the AIMMS procedure.

Obtaining the status

Through the function `AimmsProcedureAsyncRunStatus` you can obtain the status of an outstanding asynchronous execution request. The status of such a request can be

- pending,
- running,
- finished,
- deleted, or
- unknown (for an invalid request handle).

When the request is in the finished state, the return value of the AIMMS procedure will be returned via the `result` argument.

Deleting a request

You should make sure to delete all asynchronous execution handles requested during a session using the function `AimmsProcedureAsyncRunDelete`. *Failure to delete all finished requests may result in a serious memory leak if your external DLL generates many small asynchronous execution requests.* If you delete a pending request, AIMMS will remove the request from the current execution queue. The function will fail if you try to delete a request that is currently being executed.

Interrupting an existing run

When an AIMMS procedure has been started by a separate thread in your program you can interrupt it using the function `AimmsExecutionInterrupt`. This function returns `AIMMSAPI_SUCCESS` when AIMMS was idle and `AIMMSAPI_FAILURE` was executing a procedure.

34.7 Passing errors and messages

Passing errors and messages

The AIMMS API functions in Table 34.6 let you send error and warning messages to AIMMS and get the current AIMMS status. In addition, you can obtain the error number and description of the last AIMMS API error.

```
int AimmsAPIPassMessage(int severity, char *message)
int AimmsAPIStatus(int *status)
int AimmsAPILastError(int *code, char *message)
```

Table 34.6: AIMMS API functions for error messages

Passing errors and messages

With the function AimmsAPIPassMessage you can send error and warning messages to the end-user of your DLL in AIMMS. Such errors and warnings are displayed to the end-user in the AIMMS message window. For every message you must indicate a severity code, the complete list of which is included in the aimmsapi.h header file. When AIMMS receives a message with error severity, a run-time error is generated. The end-user of an application can set execution options to filter out those warning messages which are below a certain severity threshold.

Setting CurrentErrorMessage

If a function in your DLL is called from within an AIMMS project, and you want to pass back an error message to the model without automatically opening the AIMMS message window, you should not use the function AimmsAPIPassMessage, but instead assign the message to the predefined AIMMS string parameter CurrentErrorMessage. To assign a value to it, you should create a handle to it via the function AimmsIdentifierHandleCreate and assign the message using the function AimmsValueAssign. It is then up to the model developer calling your function, whether the message stored in CurrentErrorMessage should be displayed (e.g. in the AIMMS message window).

Obtaining the execution status

Through the function AimmsAPIStatus you can obtain the current status of the AIMMS execution engine, such as executing, solving, ready, etc. The complete list of possible status codes and their meaning is included in the aimmsapi.h header file.

Obtaining API errors

Whenever a call to an AIMMS API function fails, the function returns AIMMSAPI_FAILURE as its return value. In such a case, you can obtain the precise error code and a message describing the error through the function AimmsAPILastError. The complete list of error codes is contained in the aimmsapi.h header file. By modifying the API-related execution options, you can also enforce that every API error is listed in the AIMMS message window.

34.8 Raising and handling errors

Raising and handling errors

The error passing described in the previous section is retained in AIMMS 3 in order not to break existing applications. The use of the error handling described in this section, however, is encouraged as it is more in line with the error handling framework described in Section 8.4. In addition, all errors, including all their parts, can be retrieved. The AIMMS API functions in Table 34.7 enable the raising and handling of errors and retrieving the current AIMMS status.

int AimmsErrorStatus(void) int AimmsErrorCount(void)
char *AimmsErrorMessage(int errNo) int AimmsErrorSeverity(int errNo) char *AimmsErrorCode(int errNo) char *AimmsErrorCategory(int errNo) int AimmsErrorNumberOfLocations(int errNo) char *AimmsErrorFilename(int errNo) char *AimmsErrorNode(int errNo, int pos) char *AimmsErrorAttributeName(int errNo, int pos) int AimmsErrorLine(int errNo, int pos) int AimmsErrorColumn(int errNo) time_t AimmsErrorCreationTime(int errNo)
int AimmsErrorDelete(int errNo) int AimmsErrorClear(void)
int AimmsErrorRaise(int severity, char *message, char *code)

Table 34.7: AIMMS Raising and handling errors in the AIMMS API

Global error collector manipulation

The functions AimmsErrorStatus, AimmsErrorCount, AimmsErrorGet, AimmsErrorDelete and AimmsErrorClear all manipulate the global error collector. The global error collector is described in Section 8.4. The function AimmsErrorStatus scans the contents of the global error collector and returns

- **AIMMSAPI_SEVERITY_CODE_NEVER:** if the global error collector is empty,
- **AIMMSAPI_SEVERITY_CODE_WARNING:** if it contains only warnings, or
- **AIMMSAPI_SEVERITY_CODE_ERROR:** if it contains at least one error.

The function AimmsErrorCount does not return a status code, instead it directly returns the number of errors and warnings in the global error collector. With the functions AimmsErrorMessage, AimmsErrorSeverity, AimmsErrorCategory, AimmsErrorCode, AimmsErrorNumberOfLocations, AimmsErrorLine, AimmsErrorNode, AimmsErrorAttributeName, AimmsErrorFilename, AimmsErrorColumn, and AimmsErrorCreationTime actual error information is obtained. In these functions the errNo argument should be in the range {1..AimmsErrorCount()} and the pos argument should be in the range {1..AimmsErrorNumberOfLocations(errNo)}.

Example for API calls

None of the AIMMS API functions throws an exception, nor do any of them let an exception pass through. The example below serves as a simple template to call AimmsProcedureRun and handle all errors occurring during that execution run.

```
int ErrCount, errNo, apr_stat ;

apr_stat = AimmsProcedureRun( procHandle, ... );
ErrCount = AimmsErrorCount();
if ( ErrCount ) {
    for ( errNo = 1 ; errNo <= ErrCount ; errNo ++ ) {

        // Handle the error; replace the next line as
        // appropriate for the application at hand.
        printf( "Error %d: %s\n", errNo, AimmsErrorMessage(errNo) );
```

```
    }
    AimmsErrorClear();
} else if ( apr_stat == AIMMSAPI_FAILURE ) {
    printf("Aimms failed for an unknown reason.\n");
}
```

Raising an error

The function AimmsErrorRaise(severity, message, code) can raise errors and warnings. These errors will be handled by the currently active error handler as described in Section 8.4. If there is no currently active error handler, the error is directly placed in the global error collector. The call to this function is similar to the RAISE statement, see Section 8.4.2. The code argument is optional. The severity argument should be either

- **AIMMSAPI_SEVERITY_CODE_WARNING:** indicating a warning or
- **AIMMSAPI_SEVERITY_CODE_ERROR:** indicating an error.

The category of an error raised by AimmsErrorRaise is fixed to 'User'.

34.9 Opening and closing a project

Opening and closing a project

The AIMMS API functions in Table 34.8 allow you to open and close an AIMMS project from within your own application.

int AimmsProjectOpen(char *commandline, int *handle)
int AimmsServerProjectOpen(char *commandline, int *handle)
int AimmsProjectClose(int handle, int interactive)
int AimmsProjectWindow(HWND *window)

Table 34.8: AIMMS API functions for opening and closing projects

Opening a project

If you want to use AIMMS as an optimization engine from within an external program, you can use the function AimmsProjectOpen to open the AIMMS project which contains the model that you want to connect to. To open an AIMMS project you must specify the command line containing the project file name as well as any other command line options with which you want to run AIMMS, *but without the name of the* AIMMS *executable*. If the project is not in the current working directory, the directory in which the project is contained must be appended to the project file name. On success, you obtain a project handle which must be used to close the project. Because a single AIMMS instance can only run a single project, the function fails if a project is already running in this AIMMS instance.

Opening a server project

When you are running an AIMMS project as a server application, you do not want windows or message boxes to appear on the server desktop under any circumstances. Although you can open an AIMMS project in a minimized or hidden fashion through commandline arguments passed to the AimmsProjectOpen function, AIMMS can still present some message boxes, e.g. to report licensing problems during startup. With the function AimmsServerProjectOpen you can open an AIMMS project absolutely without any windowing support. If AIMMS encounters any problems during startup, the function will fail and you can retrieve the error message through the function AimmsAPILastError.

License required

When you open an AIMMS project from within your own application through the AIMMS API, the normal AIMMS licensing arrangements apply. When no valid AIMMS license is available on the host computer, a call to either AimmsProjectOpen or AimmsServerProjectOpen will fail.

Closing a project

With the function AimmsProjectClose you can request AIMMS to close the current project, and, subsequently, to terminate itself. With the interactive argument you can indicate whether the project must be closed in an interactive manner (i.e. whether the user must be able to answer any additional dialog box that may appear), or that the default response is assumed. The request will fail if the project handle is not equal to the project handle returned by the function AimmsProjectOpen, thus disallowing you to close a project that was not opened by yourself.

Obtaining the AIMMS window

Through the function AimmsProjectWindow you can obtain the WIN32 window handle associated with the current AIMMS project. You can use the window handle in any WIN32 function call inside your DLL that requires the AIMMS window handle to function properly.

34.10 Thread synchronization

Multiple threads

The AIMMS API allows multiple DLLs to be active within the context of a single project. While some of these DLLs may only be useful when called from within your AIMMS project itself, you may want other DLLs to run independently in a separate thread of execution. Such behavior may be necessary, for instance, when

- you want to link AIMMS to an online data source, where an independent DLL collects the online data and passes it on to AIMMS whenever appropriate, or
- you want to call AIMMS as an independent optimization engine from within your own program and need to pass data to AIMMS whenever necessary.

AIMMS *thread*

When you open an AIMMS project by calling the function AimmsProjectOpen from within your own application, AIMMS will create a new thread. This AIMMS thread will deal with

- all end-user interaction initiated from within the AIMMS end-user interface (which is created as part of opening the project), and
- all asynchronous execution requests that are initiated either from within your application, another external DLL linked to your AIMMS project, or from within the model itself.

Thread initialization

Whenever you want to call AIMMS API functions from within a thread started by yourself, you must make sure that the thread is well-equipped to do so by calling the AIMMS thread (un)initialization functions listed in Table 34.9.

```
int AimmsThreadAttach(void)
int AimmsThreadDetach(void)
```

Table 34.9: AIMMS API functions for thread initialization

Initializing threads...

Prior to calling any other AIMMS API function from within a newly created thread, you should call the function AimmsThreadAttach. It will make sure that any thread-specific initialization required for calling the AIMMS execution engine is performed properly. Similarly, you should call the function AimmsThreadDetach just prior to exiting the thread.

... includes COM initialization

Among others, a call to AimmsThreadAttach will initialize the Microsoft COM library in a manner compatible with the COM apartment model employed by AIMMS. Therefore, if you are using COM interfaces within your thread, you should not call the COM SDK functions CoInitialize (or CoInitializeEx) and CoUninitialize directly to initialize the COM library, but rather call AimmsThreadAttach and AimmsThreadDetach.

Thread synchronization

Whenever an AIMMS project runs in a multi-threaded environment, synchronization of the execution and data retrieval requests becomes of the utmost importance. By default, AIMMS will make sure that no two execution or data retrieval requests initiated from different threads are dealt with simultaneously. However, this default synchronization scheme does not preclude that the execution of two subsequent requests from one thread is interrupted by a request from another thread.

Obtaining exclusive control

When the proper functioning of your application requires that your execution and data retrieval requests to AIMMS are not interrupted by requests from competing threads, you can use the functions listed in Table 34.10 to obtain exclusive control over the AIMMS execution engine.

```
int AimmsControlGet(int timeout)
int AimmsControlRelease(void)
```

Table 34.10: AIMMS API functions for obtaining exclusive control

Obtaining and releasing control

With the function AimmsControlGet you can restrict control over the current AIMMS session exclusively to the thread calling AimmsControlGet. Execution and data retrieval requests from any thread other than this controlling thread (including the AIMMS thread itself) will block until the controlling thread has released the control. The function AimmsControlRelease releases the exclusive control over the AIMMS session. *Note that every successful call to AimmsControlGet must be followed by a corresponding call to AimmsControlRelease*, or AIMMS will be inaccessible to all other threads for the remainder of the session. AimmsControlRelease fails when the calling thread does not have exclusive control.

Waiting for the control

When another thread has exclusive control over AIMMS, either obtained explicitly through a call to AimmsControlGet or implicitly through an execution or data retrieval request, the function AimmsControlGet will block *timeout* milliseconds before returning with a failure. By choosing a *timeout* of WAIT_INFINITE, the function AimmsControlGet will block until it gets exclusive control.

Nonblocking execution

If you want to make sure that a subsequent execution request will never block, you can

- call AimmsControlGet with a timeout of 0 milliseconds,
- perform the execution request when successful, and
- subsequently release the control.

The call to AimmsControlGet has the effect of verifying that no other thread is using AIMMS at the moment. If you cannot get exclusive control, you must store the request for later execution.

34.11 Interrupts

Interrupts

During the execution of a procedure in your model, the AIMMS thread will block. This effectively prevents you from interrupting the AIMMS execution, for instance, to update data in the model, or just to abort the current procedure execution. Likewise, while a function in your DLL that was executed from

within an AIMMS procedure is still running, AIMMS cannot service any end-user requests. The functions listed in this section allow your DLL and AIMMS to work together in a cooperative manner in such situations.

Handling interrupts

You can use the functions listed in Table 34.11 to handle two-way interrupts.

```
int AimmsInterruptCallbackInstall(AimmsInterruptCallback cb)
int AimmsInterruptPending(void)
```

Table 34.11: AIMMS API functions for handling interrupts

Installing a callback

With the function AimmsInterruptCallbackInstall you can pass a function pointer with a prescribed prototype to AIMMS, which AIMMS will call on a regular basis during subsequent execution of an AIMMS procedure. Note that the installed callback is *thread-local*, i.e., AIMMS will only call the callback procedure from within an AIMMS procedure that is executing in the same thread in which you called AimmsInterruptCallbackInstall to install the callback.

Within a callback

Within a callback function AIMMS allows you to request or modify model data, or to run model procedures, which would normally be prohibited because calls to the AIMMS API block when AIMMS is executing (see also Section 34.10). Through the argument of the callback function AIMMS passes its current state (just executing, or within a solve), while you can indicate, through the return value of the callback function, whether you

- want to interrupt the current solve but continue the remainder of the current execution,
- want to interrupt the current execution all together, or
- do not want to interrupt the current execution at all.

Because AIMMS will call a callback procedure quite regularly, it is advisory to keep the actions executed within it to a minimum, or AIMMS could be slowed down unacceptably.

Uninstalling the callback

You can uninstall a previously installed callback function by simply calling the function AimmsInterruptCallbackInstall with a null pointer as the callback function argument. Note that it is even possible to uninstall a callback function—or modify a callback function—during a call (by AIMMS) to the currently installed callback function.

Pending interrupts

When AIMMS calls a function within an external DLL, this would normally prevent AIMMS from servicing end-user requests to update end-user pages, modify model data, or even to interrupt the execution of the current AIMMS execution, i.e. the execution of your function. This is not a problem when a call to your function only takes a small amount of time to execute, but might be unacceptable when your function takes a long time to complete. In such situations, you might consider to insert calls to the function AimmsInterruptPending at strategic places in your source code. With it, you allow AIMMS to service such requests, and to call any callback functions installed by other DLLs. On return, AimmsInterruptPending returns

- AIMMSAPI_TRUE when AIMMS received a request to interrupt the current execution, or
- AIMMSAPI_FALSE when there was no interrupt request.

When an interrupt was requested you should abort the execution of your external function as soon as possible.

34.12 Model Edit Functions

AIMMS Model Edit Functions

The AIMMS API supports Model Edit Functions allowing external applications to inspect, modify, or even construct AIMMS models. In this section, the model edit functions are introduced using a small example. Subsequently, after briefly describing the relation to runtime libraries plus the conventions used, several tables containing model edit functions, are presented and described. Finally, the limitations of using model edit functions through the AIMMS API are described briefly.

Small example

In the following example, an element parameter nextCity is created with a simple definition. Model editing is done using **model editor** handles. These handles provide access to the identifiers in the model, and should not be confused with the data handles and procedure handles described elsewhere in this chapter.
The model editor handle int dsMEH refers to a declaration section, whereas the model editor handle int ncMEH refers to the parameter nextCity.

```
1  AimmsMeCreateNode("nextCity", AIMMSAPI_ME_IDTYPE_ELEMENT_PARAMETER, dsMEH, 0, &ncMEH);
2  AimmsMeSetAttribute(ncMEH, AIMMSAPI_ME_ATTR_INDEX_DOMAIN, "i");
3  AimmsMeSetAttribute(ncMEH, AIMMSAPI_ME_ATTR_RANGE, "Cities");
4  AimmsMeSetAttribute(ncMEH, AIMMSAPI_ME_ATTR_DEFINITION,
                "if i == last(Cities) then first(Cities) "
                "else Element(Cities,ord(i)+1) endif");
5  AimmsMeCompile(ncMEH);
6  AimmsMeCloseNode(ncMEH);
```

A line by line explanation of this example follows below. For the sake of brevity, error handling, such as suggested in Section 34.8, is omitted here.

- **Line 1:** Creates an element parameter named nextCity. The fourth argument of AimmsMeCreateNode is the position within the section. A 0 indicates that it should be placed at the end of the section.
- **Lines 2-4:** Set the attributes IndexDomain, Range, and Definition of this parameter. Note that only the text is passed, these calls do not use the AIMMS compiler to compile them.
- **Line 5:** Compiles the element parameter nextCity. Only now is the text of the attributes actually checked and compiled.
- **Line 6:** The function AimmsMeCloseHandle de-allocates the handle ncMEH but the created identifier nextCity remains in the model.

Relation to runtime libraries

Section 35.6 describes the model editing facility available in the AIMMS language using runtime libraries. The advantage of using the AIMMS API, instead of runtime libraries, for model editing is that the entire model can be edited, including the main model, provided that there is no AIMMS procedure active while executing a model edit function from within the AIMMS API. The cost is that multiple languages have to be used.

Conventions for model edit functions

The model edit functions follow the following conventions:

- Each function starts with AimmsMe.
- These functions return either AIMMSAPI_SUCCESS or AIMMSAPI_FAILURE.
- A model editor handle MEH has to be closed by either AimmsMeCloseNode or AimmsMeDestroyNode.
- No distinction is made between identifiers and nodes in the model editor tree, they are both called "nodes".
- String output arguments use the type AimmsString as is explained in Section 34.2.

Model roots

Table 34.12 lists the functions available for manipulating model editor roots. The number of roots available for model editing is stored by the function AimmsMeRootCount(count) in its output argument count. Note that count is always at least 1, since there is always the main model. The root Predeclared identifiers is not included in this count. As the root Predeclared identifiers and its sub-nodes cannot be changed, it is not included in this count. To obtain a handle for an existing root, the function int AimmsMeOpenRoot(pos, MEH) can be used. A model editor handle is then created and stored in MEH. If pos is 0, the main model root is opened. If pos is in the range { 1 .. count-1 } then a library is opened. If pos equals count then the predeclared root Predeclared identifiers is opened. The root Predeclared identifiers and its sub-nodes are read only. In order to create a new runtime library the function AimmsMeCreateRuntimeLibrary(name, prefix, MEH) can be used. The position of the new library is at the end of the existing libraries.

int AimmsMeRootCount(int *count)
int AimmsMeOpenRoot(int pos, int *MEH)
int AimmsMeCreateRuntimeLibrary(char *name, char *prefix, int *MEH)
int AimmsMeNodeExists(char*name, int nMEH, int *exists)
int AimmsMeOpenNode(char*name, int nMEH, int *MEH)
int AimmsMeCreateNode(char *name, int idtype, int pMEH, int pos, int *MEH)
int AimmsMeCloseNode(int MEH)
int AimmsMeDestroyNode(int MEH)

Table 34.12: Model edit functions for roots and nodes

Opening or creating a node

The function AimmsMeNodeExists(name, nMEH, exists) can be used to test if an identifier exists. This function returns AIMMSAPI_FAILURE when nMEH does not indicate a valid namespace, or when name is not a valid identifier name. If the name is a declared identifier in namespace nMEH, then exists is set to 1, and if not to 0. The function AimmsMeOpenNode(name, nMEH, MEH) creates a handle to the node with name name in the namespace determined by the model editor handle nMEH. If successful, a model editor handle is created and stored in the output argument MEH. If nMEH equals AIMMSAPI_NULL_HANDLE_NUMBER, then the namespace of the main model is used. A new node with name name and type idtype can be created using the function AimmsMeCreateNode(name, idtype, pMEH, pos, MEH). The value of idtype must be one of the constants defined in aimmsapi.h starting with AIMMSAPI_ME_IDTYPE_. The parent node of the new node is determined by the model editor handle pMEH. The value pos determines the new position of the node within the parent node. If pos is outside the range of existing children {1..n}, the new identifier is placed at the end, otherwise the existing children at positions pos .. n are shifted to positions pos+1 .. n+1 where n was the old number of children of pMEH.

Closing or destroying a node

Table 34.12 not only lists the functions to open or create nodes, but also shows the complementary functions to close or destroy nodes. The function AimmsMeCloseNode(MEH) de-allocates the handle MEH but leaves the corresponding node in the model intact. The function AimmsMeDestroyNode(MEH) destroys the node corresponding to MEH and all nodes below that node in the model, and subsequently deallocates the handle MEH.

The name of a node

Table 34.13 lists the functions that return the name of a node. The function AimmsMeName(MEH, name) stores the name of the node to which MEH refers without any prefixes in the output argument name. The function AimmsMeRelativeName(MEH, rMEH, rName) stores the name of MEH such as it should be used from within the node rMEH in the output argument rName. A fully qualified name is stored in rName when MEH is the AIMMSAPI_ME_NULL_HANDLE_NUMBER handle.

```
int AimmsMeName(int MEH, AimmsString *name)
int AimmsMeRelativeName(int MEH, int rMEH, AimmsString *rName)
int AimmsMeType(int MEH, int *meType)
int AimmsMeTypeName(int typeNo, AimmsString *tName)
int AimmsMeAllowedChildTypes(int MEH, int *typeBuf, int typeBufsize, int *maxTypes)
```

Table 34.13: Model edit functions for name and type

The type of a node

In addition, Table 34.13 lists the functions for the type of a node. The function AimmsMeType(MEH, meType) stores the type of the node MEH in the output argument meType. The value of meType refers to one of the constants in aimmsapi.h starting with AIMMSAPI_ME_IDTYPE_. The function AimmsMeAllowedChildTypes(MEH, typeBuf, typeBufsize, maxTypes) stores the types of children allowed below the node MEH in the buffer typeBuf while respecting its size typeBufsize. The maximum number of child types below MEH is stored in the output argument maxTypes. The utility function AimmsMeTypeName(typeNo, tName) stores the name of the type typeNo in the output argument tName.

```
int AimmsMeGetAttribute(int MEH, int attr, AimmsString *text)
int AimmsMeSetAttribute(int MEH, int attr, const char *txt)
int AimmsMeAttributes(int MEH, int attrsBuf[], int attrBufSize, int *maxNoAttrs)
int AimmsMeAttributeName(int attr, AimmsString *name)
```

Table 34.14: Model edit functions for attributes

The attributes of a node

Table 34.14 lists the functions available for handling the attributes of a node. All attributes correspond to constants in the aimmsapi.h file. These constants start with AIMMSAPI_ME_ATTR_. The function AimmsMeGetAttribute(MEH,attr,text) stores the contents of attribute attr of node MEH in the output argument text. The function AimmsMeSetAttribute(MEH,attr,txt) sets the contents of attribute attr of node MEH to txt. This function will fail if attribute attr is not applicable to identifier MEH, but the text itself is not checked for errors. The function AimmsMeAttributes(MEH, attrsBuf, attrBufSize, maxNoAttrs) provides the applicable attributes for these two functions. It will store the constants corresponding to the attributes available to node MEH in attrBuf while respecting the size of that buffer attrBufSize. The maximum number of attributes available to node MEH is stored in maxNoAttrs. The function AimmsMeAttributeName(attr, name) stores the name of attr in name.

Basic node manipulations

The functions that support changing the aspects of a node such as name, location, and type of a node are also shown in Table 34.15. The function AimmsMeNodeRename(MEH, newName) changes the name of a node, and the name-change is applied to the attribute texts that reference this node. An entry is appended to the name change file if the node is not a runtime node. The function AimmsMeNodeMove(MEH, pMEH, pos) moves the node MEH to child position

```
int AimmsMeNodeRename(int MEH, char *newName)
int AimmsMeNodeMove(int MEH, int pMEH, int pos)
int AimmsMeNodeChangeType(int MEH, int newType)
int AimmsMeNodeAllowedTypes(int MEH, int* typeBuf, int typeBufsize, int *maxNoTypes)
```

Table 34.15: Model edit functions for node manipulations

pos of node pMEH. If this results in a change of namespace, the corresponding namechange is applied to the attributes that reference this node. In addition, an entry is appended to the corresponding name change file if this node is not a runtime node. Moves from one library to another are not supported, nor is a move in or out of the main model. The function AimmsMeNodeChangeType(MEH, newType) changes the type of a node. It will retain available attributes whenever possible. The function AimmsMeNodeAllowedTypes can be used to query which types, if any, a particular node can be changed to. The function AimmsMeNode-AllowedTypes(MEH, typeBuf, typeBufsize, maxNoTypes) will store all the types into which node MEH can be changed in a buffer typeBuf that respects the size typeBufsize. The maximum number of types into which MEH can be changed is stored in maxNoTypes.

Tree walk of the model

Table 34.16 lists the functions that permit walking all nodes in the model editor tree. The function AimmsMeParent(MEH, pMEH) creates a model editor handle to the parent of MEH, and stores this handle in the output argument pMEH. The function AimmsMeFirst(MEH, fMEH) creates a model editor handle to the first child of MEH, and stores this handle in the output argument fMEH. The function AimmsMeNext(MEH, nMEH) creates a model editor handle to the node next to MEH, and stores this handle in the output argument nMEH. If such a parent, first child, or next node does not exist the AIMMSAPI_ME_NULL_HANDLE_NUMBER handle is stored in the output argument although the corresponding function does not fail.

```
int AimmsMeParent(int MEH, int *pMEH)
int AimmsMeFirst(int MEH, int *fMEH)
int AimmsMeNext(int MEH, int *nMEH)
int AimmsMeImportNode(int MEH, char *fn, const char *pwd)
int AimmsMeExportNode(int MEH, char *fn, const char *pwd)
```

Table 34.16: Reading, writing and tree walking a model editor tree

Reading and writing (portions of) a model

The functions that allow the reading of an AIMMS section from a file, or writing a section to a file are also listed in Table 34.16. They use the Text .ams file format. The function AimmsMeImportNode(MEH, fn, pwd) reads a file fn and stores the resulting model structure at node MEH. The function AimmsMeExportNode(MEH, fn, pwd) writes the model structure at node MEH to file fn. If MEH does not refer to an AIMMS section, module, library, or model, the functions AimmsMeImportNode and AimmsMeExportNode will fail.

Compilation

The model edit functions available for compilation and model status queries are listed in Table 34.17. The central function `AimmsMeCompile (MEH)` compiles the node `MEH` and all its sub-nodes. The entire application (main model and libraries) is compiled if the argument `MEH` equals `AIMMSAPI_ME_NULL_HANDLE_NUMBER`. If this compilation step is successful then the procedures are runnable. The function `AimmsMeIsRunnable(MEH, r)` stores 1 in the output argument `r` if the procedure referenced by `MEH` is runnable. The function `AimmsMeIsReadOnly(MEH, r)` stores 1 in the output argument `r` if the node resides in a read-only library, such as the `predeclared identifiers`, or a library that was read from a read only file.

```
int AimmsMeCompile(int MEH)
int AimmsMeIsRunnable(int MEH, int *r)
int AimmsMeIsReadOnly(int MEH, int *r)
```

Table 34.17: Model edit functions for compilation and status queries

Limitations

The following limitations apply to model edit functions from within the AIMMS API:

1. The `SourceFile` attribute is not supported.
2. The current maximum number of identifiers is thirty thousand.

Further, when an AIMMS procedure is running, the identifiers in the main application can not be modified as explained in Section 35.6.

Chapter 35

Model Structure and Modules

Model and sections

This chapter discusses the common structuring components of a model, namely the *main model* and model *sections*. With the use of sections, you can provide depth to the model tree in the AIMMS **Model Explorer**, which allows you to structure your model in any logical manner that makes sense to you. Imposing a clear and logical structure to your model will strongly add to the overall maintainability of your modeling application.

Modules

The next concept introduced in this chapter is that of a *module*, which is basically a model section with its own, separate, namespace. Modules allow you to share sections of model source between multiple models, without the risk of running into name clashes. AIMMS uses modules to implement those parts of its functionality that can be best expressed in the AIMMS language itself. The available AIMMS system modules include

- a (customizable) implementation of the outer approximation algorithm,
- a scenario generation module for stochastic programming, and
- sets of constants used in the graphical 2D- and 3D-chart objects.

Library modules

Finally, this chapter discusses the concept of a *library module*, which is the source module associated with a library project (see Section 3.1 of the User's Guide). Library modules can only be added to an AIMMS model through the **Library Manager**, and are always displayed as a separate root in the model tree.

35.1 Introduction

Support for large models

When a model grows larger, the need for a clear and logical storage structure of all its constituting components also grows. In the absence of such a logical storage structure, you will find that it becomes increasingly hard to find your way in the model source because of the huge amount of information it contains. To support you in structuring your model, AIMMS offers several development tools and language constructs just for this purpose.

The model tree

To support you in structuring your model, AIMMS lets you organize all identifier declarations and procedures of your model in the form of a tree, called the *model tree*. You can access the model tree in a graphical manner, using the **Model Explorer** tool (see also Chapter 4 of the User's Guide). Several language constructs of the AIMMS language, such as collections of identifiers declarations, or the procedures and functions included in your model, are visible as separate nodes within the model tree.

Main model node

All model declarations in an AIMMS model are located underneath the root node of the model tree, the `Model` node. The `Model` node is always present in the **Model Explorer**, even when you start a new AIMMS project, and cannot be deleted. For the `Model` node itself, you can specify several attributes that have a global impact, such as the licensing arrangements for your model, or the unit convention (if any) that is applicable to your application as a whole. The attributes of the `Model` node are discussed in full detail in Section 35.2.

Model sections ...

As you start adding identifier declarations, procedures and functions to a new model, you will soon notice that storing all these declarations directly underneath the `Model` node will result in a nearly unmanageable list of declarations. Finding information in such a (linear) list soon becomes a daunting task. To support you in adding additional structure to your model tree AIMMS provides `Section` nodes, which allow you to add depth to the model tree, much like directories add depth to a file system.

... to structure a model

By adding `Section` nodes with meaningful names to the model tree, and storing all model declarations that you find relevant for these section underneath them, you can impose any logical structure on your model tree that you find useful. Because AIMMS allows identifiers to be used prior to their declaration, you do not even have to worry about the declaration order when you reorganize the model tree in this manner.

Separate source file

In addition to providing the structuring capabilities described above, the contents of a `Section` node can also be stored in a separate source file. You can import the contents of such a source file into a section of another model, or permanently link the contents of a section to the contents of the source file. This allows you to reuse part of one model within similar applications. This method of sharing functionality, however, has its limitations. Name clashes can occur when an imported section redeclares an identifier already declared in the main model. If you run into these limitations, you are advised to use the `Module` concept discussed below.

Section attributes

The attributes of a Section node allow you to specify such issues as whether its contents needs to be stored in a separate source file, and if the usage of such a source file needs to be licensed. The attributes of a Section node are discussed in full detail in Section 35.3.

Complex modeling environments

When the development of modeling applications becomes the core business of an organization, this will almost certainly lead to a multitude of related modeling projects, collaborating developers, and various end-user types, all subject to frequent changes over time. Projects evolve naturally due to feedback from end-users, changing application environments, and rotating personnel. Changes in the pool of model developers are inevitable, and may cause major fluctuations in application knowledge, experience, and modeling skills. End-users of applications also change jobs, which may result in new requirements and customization requests from the newcomers.

Need for modularization

In such a dynamic modeling world, the exchange of information becomes a crucial element to avoid unnecessary duplication. When projects are customized for different end-users, there is apt to be quite a bit of commonality between these projects. If these commonalities are not exchanged properly, there will be multiple and differing versions of essentially the same model segments. As a result, extensive and costly human resources will be needed to maintain these multiple related models. Modularization can help to overcome these problems.

Collections of functions and procedures

In Chapter 10 you were introduced to functions and procedures as the initial tools to modularize the functionality within an AIMMS project. As explained above, collections of functions and procedures, along with the required identifier declarations, can be stored in model sections. These can be exported to separate .ams files, and can subsequently be imported by, or linked into, any other AIMMS project. Every time an AIMMS project is started containing a section link, it will automatically pick up the latest version of the file. This means that when such a collection of functions, procedures and identifier declarations at a customer's site need to be updated, only their corresponding files need to be replaced.

Name clashes

One problem that you are likely to run into with the above approach, however, is the occurrence of name clashes. Some of the identifier names of procedures, functions, and identifiers in a model section may also occur in the model in which the section is to be included. Such name clashes will effectively prevent AIMMS from importing or linking the section into your model. A possible solution to this problem would be to rename the offending identifiers, either in your model or in the section to be included. However, using either approach, the same problems are likely to return when you get an updated version of the included model section.

Modules ...

A more structural solution to the name clash problem is provided by the concept of *modules* in AIMMS, which allow you to share common model source into multiple models, without the risk of running into name clashes. Modules are inserted into the model tree by means of `Module` nodes. These nodes are essentially `Section` nodes with a separate namespace, along with attributes to manipulate the global model namespace. The attributes of `Module` nodes are discussed in full detail in Section 35.4.

... avoid name clashes

Like `Section` nodes, `Module` nodes can be exported to a separate source file, which can be imported or linked into another model. However, because all identifiers declared within the `Module` node only live in its associated namespace, importing a module into another project will not lead to name clashes anymore.

Dividing a project into sub-projects...

When a project becomes larger, the operational demands and sheer amount of work involved in implementing the project, may become too demanding for a single modeler to keep up with. It is then time to divide the project into a number of manageable sub-projects, on which individual developers can work more or less independently.

... unsuitable for modules

Modules, as discussed above, are not necessarily the most suitable instrument to facilitate a division into sub-projects. This is mainly due to the fact that the module concept does not allow identifiers in the module to be strictly private to that module. Because of this, other developers can, in principle, refer to all identifiers in the module, and, consequently, the chances of a single structural change in any of the modules breaking the entire application are considerable.

Library projects

To address the problem of allowing multiple developers to work independently on manageable sub-projects of a big AIMMS project more thoroughly, AIMMS supports the concept of *library projects*. Library projects go far beyond modules—they do not only support independent model development, but a developer can also create end-user pages and menus as part of the library project. When a library project is included in a main project, the associated overall application can then be composed by combining the model source, pages, and menus created as part of all its included libraries. Library projects are discussed in full detail in Chapter 3 of the User's Guide.

Library modules

Library modules are the source code modules associated with library projects. They can only be added to your model through the **Library Manager** discussed in Section 3.1 of the User's Guide. AIMMS will insert library modules into the model tree as a separate `LibraryModule` root node. The attributes of `LibraryModule` nodes are discussed in full detail in Section 35.5.

Library interface

As with ordinary modules, library modules have an associated namespace, which helps to avoid name clashes when including a library project into an AIMMS project. In addition, however, library modules provide a public *interface* to the rest of the model. Within the library project, all identifiers declared in the library can be freely used in the source of the library module, its pages and menus. The main project, and all other library projects included in the main project, however, can only access the identifiers that are part of the interface of the library. This allows a developer of a library to freely change any declaration that is not part of the library interface, without the risk of breaking the entire application.

35.2 `Model` declaration and attributes

`Model` declaration and attributes

The `Model` node defines the root node of the entire model tree associated with an AIMMS modeling project. The attributes of the main `Model` node are listed in Table 35.1. All attributes of the `Model` node have a global impact on the entire modeling project.

Attribute	Value-type	See also page
`Convention`	*convention, element-parameter*	
`Comment`	*comment string*	19

Table 35.1: `Model` attributes

The `Convention` attribute

With the `Convention` attribute you can indicate that all I/O with respect to identifiers in your model is to take place according to the unit convention specified in this attribute. The value of this attribute must be either a direct reference to a convention declared in your model, an element parameter into the set `AllConventions` or a string parameter holding the name of a convention within your model. You can find more detailed information about unit conventions and their usage in Section 32.8

35.3 `Section` declaration and attributes

`Section` declaration and attributes

`Section` nodes provide depth to the model tree, and offer facilities to store parts of your model in separate source files. A `Section` node is always a child of the `Model` node, of another `Section` node, or of a `Module` node. The attributes of `Section` nodes are listed in Table 35.2.

Attribute	Value-type	See also page
SourceFile	*string*	
Property	NoSave	
Comment	*comment string*	19

Table 35.2: Section attributes

The SourceFile attribute

With the SourceFile attribute you can indicate that the contents of a Section node in your model is linked to the specified source file. As a consequence, AIMMS will read the contents of the Section node from the specified file during compilation of the model. Any modifications to the part of the model contained in such a Section node will also be stored in this source file when you save the model. When you select an existing source file for the SourceFile attribute of a Section node in the **Model Explorer** (see also Section 4.2 of the User's Guide), any previous contents of that section will be lost.

The Property attribute

In the property attribute the NoSave property can be specified. When the property NoSave is set, none of the identifiers declared in the section will be saved in cases.

Section names as identifier subsets

Whenever you add a Section node to the model tree, the name of the Section node (with spaces replaced by underscores), will also be available within your model as an implicit subset of the predeclared set AllIdentifiers. The contents of this subset is fixed, and is defined as the set of all identifiers declared within the subtree corresponding to the Section node. You can use this implicitly created set, for instance, in the EMPTY statement to empty all section identifiers using only a single statement.

35.4 Module declaration and attributes

Module declaration and attributes

Module nodes create a subtree of the model tree along with a separate namespace for all identifier declarations in that subtree. Like Section nodes, the model contents associated with a Module node can be stored in a separate source file. A Module node is always a child of the main Model node, of a Section node, or of another Module node. The attributes of Module nodes are listed in Table 35.3.

The SourceFile attribute

Like with ordinary Section nodes, the contents of a Module node can also be stored in a separate source file, dynamically linked into a Module node in your model through the use of the SourceFile attribute.

Attribute	Value-type	See also page
SourceFile	*string*	614
Property	NoSave	614
Prefix Public Protected	*identifier* *identifier-list* *identifier-list*	
Comment	*comment string*	19

Table 35.3: Module attributes

Modules and namespaces

The distinguishing feature of modules is that each module is supplied with a separate namespace. This means that all identifiers, procedures and functions declared within a module are, without using the module prefix, only visible within that module. In addition, within a module it is possible to redeclare identifier names that have already been declared outside the module.

Nested modules

Modules in an AIMMS model can be nested. This implies that with each AIMMS model containing one or more Module nodes, one can associate a corresponding tree of nested namespaces. This tree of namespaces starts with the global namespace of the Model node as the root node. As a consequence, you can associate a path of namespaces with every identifier, procedure or function declaration in the model tree. This path of namespaces starts with the global namespace down to the namespace associated with the module in which the declaration is contained.

Scoping rules

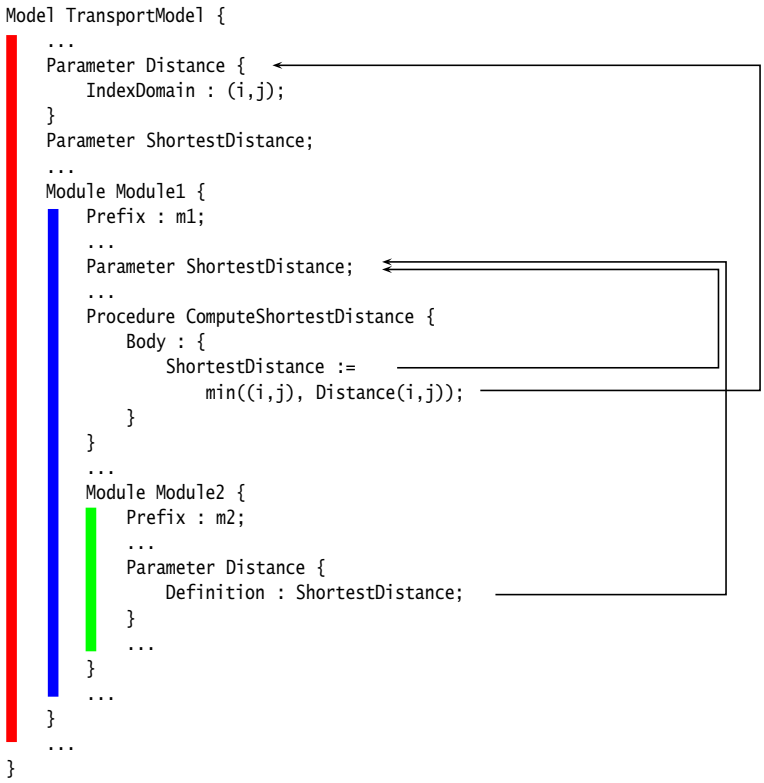
When AIMMS encounters an identifier reference during the compilation of a procedure or function body or in one of the attributes of an identifier declaration, AIMMS will search for a declaration of the identifier at hand in the following order.

- If the referenced identifier is declared in the namespace associated with the Module (or Model) in which the procedure, function or identifier is contained, AIMMS will use that particular declaration.
- If the referenced identifier cannot be found, AIMMS will repeatedly search the next higher namespace until a declaration for the identifier is found.

Consequences

As a result of these scoping rules, whenever the corresponding identifier name is referenced within a module, AIMMS has will always to refer to the identifier declaration within the same module rather than to a possibly contradicting declaration for an identifier with the same name anywhere higher up, or sideways, in the model tree. This feature enables multiple developers to work truly independently on different modules used within a model.

Consider the following model with two (nested) modules, called Module1 and Module2. The following can be concluded by applying the scoping rules listed *Example*



above.

- The reference to ShortestDistance in the procedure ComputeShortestDistance in the module Module1 refers to the declaration ShortestDistance within that module, and *not* to the declaration ShortestDistance in the main model.
- The reference to Distance in the procedure ComputeShortestDistance in the module Module1 refers to the declaration Distance(i,j) in the main model, and *not* to the scalar declaration Distance within the nested module Module2.
- The reference to ShortestDistance in the module Module2 refers to the declaration ShortestDistance within the module Module1, and *not* to the declaration ShortestDistance in the main model.
- The parameter Distance in the module Module2 does not conflict with the declaration of Distance(i,j) in the main model, because the former is only visible within the scope of the module Module2.

Accessing protected identifiers

The separate namespace of every module actively prevents identifiers within a module from being "seen" outside the module. For this reason, identifiers declared within a module are also referred to as *protected* identifiers. AIMMS, however, still allows you to reference protected identifiers anywhere else in your model through the use of the *namespace resolution operator* ::. In combination with a module-specific prefix, the :: operator accurately lets you indicate that you are referring to a protected identifier declared in the particular module associated with the prefix.

The Prefix attribute

With the *mandatory* Prefix attribute of a Module node, you must specify a module-specific prefix to be used in conjunction with the :: operator. The value of the Prefix attribute should be a unique name within the namespace of the surrounding module (or main model), and will subsequently be added to this namespace. In conjunction with the :: operator the prefix unambiguously identifies the namespace from which a particular identifier should be taken.

The :: namespace resolution operator

With the *namespace resolution operator* :: you instruct AIMMS to look for the identifier directly following the :: operator within the module associated with the prefix in front it. The :: operator may be optionally surrounded with spaces. By stacked use of the :: operator you can indicate that you want to refer to an identifier declared in a nested module. Each next prefix should refer to the Prefix attribute of the module declared directly within the module associated with the previous prefix.

Using global identifiers in ModuleS

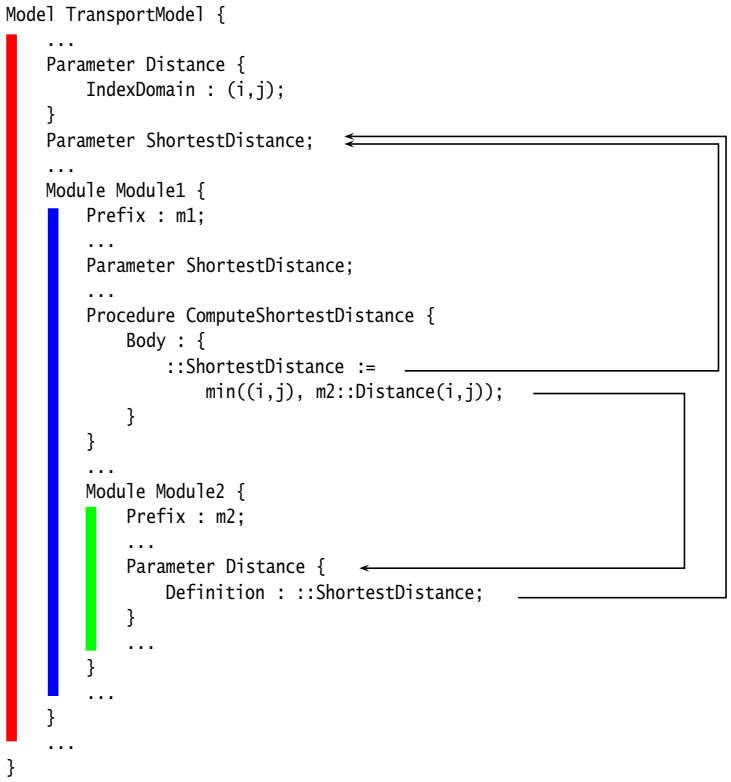
If you want to refer to an identifier in the main model, that is also declared elsewhere along the path from the current module to the main model, you can use the :: operator *without a prefix*. This indicates to AIMMS that you are interested in an identifier declared in the global namespace associated with the main model.

Examples

Consider the model outlined in the example above.

- Within the main model, a reference m1::ShortestDistance would refer to the parameter ShortestDistance declared within the module Module1, and not to the parameter ShortestDistance declared in the main model itself.
- Within the main model, a reference m1::m2::Distance would refer to the parameter Distance declared in the module Module2 nested within the module Module1.
- Within the module Module1, a reference to ::ShortestDistance would refer to the parameter ShortestDistance declared in the main model, and not to the parameter ShortestDistance declared in Module1.
- Within the module Module2, a reference to ::Distance would refer to the parameter Distance declared in the main model, and not to the parameter Distance declared in Module2.

The following model outline, which is a variation of the model outline of the previous example, further illustrates the consequences of the use of the :: operator.



The Public attribute

Through the Public attribute you can indicate that a set of identifiers declared within the module is public. These identifiers can then be referenced without the :: operator within the importing module (or main model). The value of the Public attribute must be a constant set expression. You might consider the identifiers specified in the Public attribute as the public interface of a module. As a result, AIMMS will effectively add the names of these identifiers to the namespace of the importing module, as if they were declared within the importing module itself.

Example

Consider the model outline of the first example, and assume that the declaration of module Module2 is augmented as follows.

```
Module Module2 {
    Prefix : m2;
    Public : {
        data { Distance }
    }
    ...
    Parameter Distance {
```

```
            Definition : ShortestDistance;
        }
        ...
    }
```

As a result of the Public attribute, Distance will be added to the namespace of Module1, and the compilation of the procedure ComputeShortestDistance will fail because Distance will now refer the scalar declaration in Module2 rather than to the 2-dimensional declaration in the main model. In addition, it is possible, within the main model, to refer to the parameter Distance in Module2 through the expression m1::Distance, because Distance has been effectively added to the namespace of module Module1.

Propagation of public identifiers

When an identifier is added to the Public attribute of an imported module, it is, as explained above, effectively added to the namespace of the importing module. This creates the possibility to add a public identifier of an imported module to the Public attribute of the importing module as well. In this way you can propagate the public character of such an identifier to the next outer namespace. For example, by adding the identifier Distance in the example above, to the Public attribute of the module Module1 as well, it would also become public in the main model. Obviously, in this case, adding Distance to the Public attribute of Module1 would cause a name clash with the global identifier Distance(i,j).

The Protected attribute

Once you import a module into an existing AIMMS application, one or more identifiers in the public interface of the imported module can cause name clashes with existing identifiers in the application, like Distance in the example of previous paragraph. When you run into such a problem, AIMMS allows you to override the Public status of one or more identifiers of a module through its Protected attribute. The value of the Protected attribute must be a constant set expression, and its contents must be a subset of the set of identifiers specified in the Public attribute. By adding an identifier to the Protected attribute, it is, again, only accessible outside of the module by using the :: operator.

Public versus Protected responsibilities

The responsibilities for specifying the Public and Protected attributes are substantially different, and result in a different storage of the values of these attributes. This is similar to the SouceFile-related attributes discussed earlier in this chapter. The following rules apply.

- The Public attribute is intended for the *developer* of a module to define a public interface to the module. If the module is stored in a separate .amb file, to be imported by other AIMMS applications, the contents of the Public attribute is stored inside the module-specific .amb file.
- The Protected attribute is intended for the *user* of a module to override the public character of certain identifiers as specified by the developer of the module. As the contents of the Protected attribute is not an integral part of the module, but may be specified differently by every user of the

module, it is never stored in a module-specific .amb file, but rather in the importing module or main model.

Unique global representation

For each identifier in an AIMMS model, there is a unique global representation. If the identifier is contained in the global namespace of the main model, the global representation is the identifier name itself. If an identifier is only contained in the namespace of a particular module, its unique representation based on the namespace Prefix of the module and the :: operator. Thus, for the first example of this section (without Public attributes), the unique global representations of all identifiers are:

- Distance(i,j)
- ShortestDistance
- m1::ShortestDistance
- m1::ComputeShortestDistance
- m1::m2::Distance

With the Public attribute of Module2 defined as in the previous example, the unique global representation of the parameter Distance in Module2 becomes m1::Distance, as it effectively causes Distance to be contained in the namespace of Module1.

Display and data transfer

Whenever AIMMS is requested to DISPLAY or WRITE the contents of one or more identifiers in your model, it will use the unique global representation discussed in the previous paragraph. Also, when you READ data from a file, AIMMS expects all identifiers for which data is provided in the file to be identified by their unique global representation.

35.5 LibraryModule declaration and attributes

LibraryModule declaration and attributes

LibraryModule nodes create a separate tree in the model tree along with a separate namespace for all identifier declarations in that subtree. The model contents associated with a LibraryModule is always stored in a separate source file. Contrary to Section and Module nodes, the name of this source file cannot be specified in the LibraryModule declaration. Rather, it is specified when you add the library to your project using the **Library Manager**, as discussed in Section 3.1 of the User's Guide. The attributes of LibraryModule nodes are listed in Table 35.4.

Library modules and namespaces

Like a normal module, each LibraryModule is supplied with a separate namespace. Compared to normal modules, however, the visibility rules for identifiers in a library modules are different. They are more in line with the intended use of libraries, i.e. to enable a single developer to work independently on the model source of a library.

Attribute	Value-type	See also page
Prefix	*identifier*	
Interface	*identifier-list*	
Property	NoSave	614
Comment	*comment string*	19

Table 35.4: LibraryModule attributes

The Interface attribute

Through the Interface attribute of a LibraryModule you can specify the list of identifiers in the module that you want to be part of its public interface. Only identifiers in the library interface can be accessed in model declarations, pages and menu items that are not part of the library at hand. Library identifiers not in the interface are strictly private to the library, and can never be used outside of the library.

The Prefix attribute

With the *mandatory* Prefix attribute of a LibraryModule node, you must specify a module-specific prefix to be used in conjunction with the :: operator. The value of the Prefix attribute should be a unique name within the main model.

No propagation to global namespace

Even though identifiers in the interface of the library are visible outside of the library, AIMMS *always* requires the use of the library prefix to reference such identifiers. Library modules do not support the Public attribute of ordinary modules to propagate identifiers to the global namespace.

Library initialization and termination

When creating a new library, AIMMS will automatically add LibraryInitialization, PostLibraryInitialization, PreLibraryTermination and LibraryTermination procedures to it. These procedures will be executed during the initialization and termination of your model. The distinction between these steps are explained in more detail in Section 25.1.

35.6 Runtime Libraries and the Model Edit Functions

Runtime libraries and the AIMMS Model Edit Functions permit applications to adapt to modern flexibility requirements of the model; at runtime you can create identifiers and subsequently use them. A few use cases, in which the need for flexibility in the model grows, are briefly outlined below.

Use case: automating modeling tasks

You may want to improve the maintainability of your application by

- Generating similar statements that act on dynamic selections of identifiers, or

- Generate necessary parameters and database table identifiers with their mapping attributes by querying a relational database schema when setting up a database link with your model.

Use case: Cooperative model development

Another example, in cooperative model development, a model is developed together with the users of that model. For instance, an existing application framework is demonstrated to the users and, subsequently, the suggestions from these users are taken into account. A suggestion might be to add structural nodes or arcs, or might be to add a particular restriction on existing nodes and arcs.

Use case: Proprietary user knowledge

Further, not all structural information may be available at the time of model development; some users may need to add their proprietary knowledge to the model at runtime. Examples of such proprietary knowledge are:

- Pricing rules for the valuation of portfolios.
- Blending rules for the prediction of property values of blends.

Use case: ad hoc user queries

A final example of a modern flexibility requirement is a user who has additional questions only when the results are actually presented. Such a user wants to question the model in order to understand a particular result. This person is only able to formulate the question after the unexpected result presents itself.

Runtime editing of identifiers

In the above use cases, applications create, manipulate, check, use, and destroy AIMMS identifiers at runtime. Such operations are performed by the Model Edit Functions. Such applications need to:

1. Have a place to store these AIMMS identifiers and to retrieve them from. Such a place is called an AIMMS runtime library.
2. Have functions and procedures available to create, modify, check, and destroy these AIMMS identifiers. Together, these functions and procedures form the Model Edit Functions.
3. Have a way to use these identifiers inside the model.
4. *And be able to continue execution in the presence of errors.* This fourth requirement is an essential aspect of all the other requirements and is central to the design of the AIMMS Runtime libraries and AIMMS Model Edit Functions. Global and local error handling is described in Section 8.4.1.

Runtime identifiers and libraries

The identifiers created, modified, checked, used, and destroyed at runtime are called runtime identifiers. These runtime identifiers are declared within a runtime library. A runtime library is itself also a runtime identifier: it can also be created, modified, checked, used, and destroyed at runtime. A runtime identifier can have any AIMMS type, except for quantity.

Separation between main application and runtime libraries

Model edit functions are only allowed to operate on runtime identifiers. Runtime identifiers exist at runtime but do not yet exist at compile time; the names of runtime identifiers cannot be used directly in the main model. This enforces a separation between identifiers in the main application and runtime identifiers as depicted in Figure 35.1. On the left side of this architecture there is a main application consisting of a main model and zero, one or more libraries. On the right there are zero, one or more runtime libraries. Compilation errors can occur within runtime libraries at runtime. The identifiers inside the main application are not affected by such an error; that is, provided it has local error handling, any procedure inside the main application can continue execution in the presence of compilation errors on identifiers in a runtime library. This is an important advantage of the separation: for several of the use cases presented above, this separation enables continuation in the presence of errors.

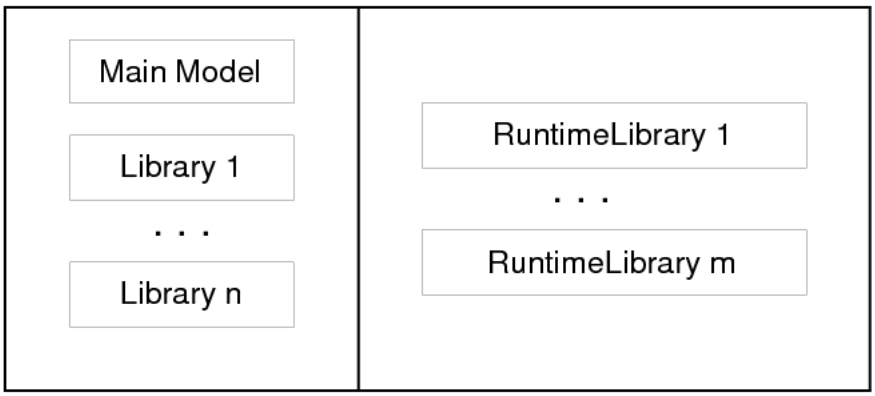


Figure 35.1: Separation between main application and runtime libraries

Example of creating an identifier

In this example, a runtime procedure `rp` is created and its body specified. This procedure is created in the runtime library `MyRuntimeLibrary1` with prefix `mrl`. The purpose of the runtime procedure `rp` is to write out the runtime parameter `P` declared in the same runtime library. This example assumes that both the runtime library `MyRuntimeLibrary1` and the runtime parameter `P` already exist.

```
Procedure DisplayDataOfRuntimeIdentifierTabular {
    ElementParameter erp {
        Default    : 'MainExecution';
        Range      : AllIdentifiers;
    }
    StringParameter str;
    ElementParameter err {
        Range      : errh::PendingErrors;
    }
    ElementParameter err2 {
        Range      : errh::PendingErrors;
    }
    Body {
 1      block
```

```
 2          erp := me::Create("rp", 'procedure', 'MyRuntimeLibrary1', 0);
 3          me::SetAttribute(erp, 'body', "display { P } ;");
 4          me::Compile(erp);
 5          me::Compile('MyRuntimeLibrary1');
 6          Apply(erp);
 7          me::Delete(erp);
 8      onerror err do
 9          if erp then
10              block
11                  me::Delete(erp);
12              onerror err2 do
13                  if errh::Severity(err2) = 'Severe' then
14                      DialogMessage(errh::Message(err2) +
15                          "; not prepared to handle severe errors " +
16                          "and halting execution");
17                      halt ;
18                  else
19                      errh::MarkAsHandled(err2) ;
20                  endif ;
21              endblock ;
22              erp := '' ;
23          endif ;
24          errh::MarkAsHandled(err);
25          DialogMessage("Creating and executing rp failed; " + errh::Message(err) );
26      endblock ;
    }
```

A line by line explanation of this example follows below.

- **Lines 1, 8, 25:** In order to handle the errors during a group of model edit actions, a `BLOCK` statement with an `ONERROR` clause is used.
- **Lines 2 - 7:** Contain the calls to the model edit functions. Note that these are formulated without any concern for errors because these errors are handled in line 9 - 25.
- **Line 2:** Create the procedure `rp` as the final procedure in the runtime library `MyRuntimeLibrary1`. The prefix of the library will be prefixed to the name of the identifier created; and after this statement the value of the element parameter `erp` is `'mrl::rp'`.
- **Line 3:** Sets the contents of the body of that procedure. Here it is to display the parameter `P` in tabular format.
- **Line 4:** Checks the procedure `mrl::rp` for errors.
- **Line 5:** Compiles the entire runtime library `MyRuntimeLibrary1` which will make the procedures inside that library runnable.
- **Line 6:** Executes the procedure just created.
- **Line 7:** Delete the procedure just created.
- **Lines 9 - 23:** Try to delete `erp` (`mrl::rp`) if it has not already been deleted.
- **Lines 13 - 20:** Ignore all errors during the deletion except for severe internal errors.
- **Line 24:** Mark the error `err2` as handled.
- **Line 25:** Finally notifies the application user that something has gone wrong.

Model Edit Functions

Model editing is available from within the language itself with intrinsic functions and procedures to view, create, modify, move, rename, compile, and delete identifiers. An intrinsic function or procedure that modifies the application is called a Model Edit Function. These functions and procedures reside in the predeclared module `ModelEditFunctions` with the prefix `me`. The table below lists the Model Edit Functions and briefly describes them.

`me::CreateLibrary`(*libraryName*, *prefixName*)→`AllIdentifiers`
`me::Create`(*name*, *newType*, *parentId*, *pos*)→`AllIdentifiers`
`me::Delete`(*runtimeId*)

`me::ImportLibrary`(*filename*[, *password*])→`AllIdentifiers` `me::ImportNode`(*esection*, *filename*[, *password*]) `me::ExportNode`(*esection*, *filename*[, *password*])
`me::Parent`(*runtimeId*)→`AllIdentifiers` `me::Children`(*runtimeId*, *runtimeChildren(i)*)
`me::ChildTypeAllowed`(*runtimeId*, *newType*) `me::TypeChangeAllowed`(*runtimeId*, *newType*) `me::TypeChange`(*runtimeId*, *newType*)
`me::GetAttribute`(*runtimeId*, *attr*) `me::SetAttribute`(*runtimeId*, *attr*, *txt*) `me::AllowedAttribute`(*runtimeId*, *attr*)
`me::Rename`(*runtimeId*, *newname*) `me::Move`(*runtimeId*, *parentId*, *pos*)
`me::IsRunnable`(*runtimeId*)
`me::Compile`(*runtimeId*)

Table 35.5: Model Edit Functions for runtime libraries

Creating and deleting

Table 35.5 lists the Model Edit Functions. A new runtime library can be created using the function `me::CreateLibrary`. if successful this function returns the library as an element in `AllSymbols`. The function `me::Create` creates a new node or identifier with name `name` of type `type` in section `ep_sec` at position `pos`. The return value is an element in `AllSymbols`. If inserted at position i ($i > 0$), the declarations previously at positions i .. n are moved to positions $i + 1$ .. $n + 1$. If inserted at position 0, the identifier is placed at the end. The procedure `me::Delete` can be used to delete both a runtime library and a runtime identifier in a library. All subnodes of `ep` in the runtime library are also deleted.

Reading and writing

The procedure `me::ImportNode` reads a section, module, or library into node `ep`. If `ep` is a runtime library, an entire library is read, replacing the existing prefix. `me::ExportNode` writes the contents of the model editor tree referenced by `ep` to a file. These two procedures use the text `.ams` file format.

Inspecting the tree

The function `me::Parent(ep)` returns the parent of `ep`, or the empty element if `ep` is a root. The function `me::Children(ep, epc(i))` returns the children of `ep` in `epc(i)` in which `i` is an index over a subset of `Integers`.

Node types

The function `me::ChildTypeAllowed(ep, et)` returns 1 if an identifier of type `et` is allowed as a child of `ep`. The function `me::TypeChangeAllowed(ep, et)` returns 1 if the identifier `ep` is allowed to change into type `et`. The procedure `me::TypeChange(ep,et)` performs a type change while attempting to retain as many attributes as possible.

Attributes

The function `me::GetAttribute(ep, attr)` returns the contents of the attribute `attr` of identifier or node `ep`. The complementary procedure `me::SetAttribute (ep,attr,str)` specifies these contents. The function `me::AllowedAttribute(ep, attr)` returns 1 if attribute `attr` of identifier `ep` is allowed to have text.

Changing name or location

The procedure `me::Rename(ep, newname)` gives `ep` a new name `newname`. The text inside the library is adapted, but a corresponding entry in the namechange file is not created. The procedure `me::Move(ep, ep_p, pos)` moves an identifier from one location to another. When an identifier changes its namespace, this is a change of name, and the text in the runtime library is adapted correspondingly, but no entry in the namechange file is created. Runtime identifiers can not be moved from one runtime library to another.

Querying runtime library status

The function `me::IsRunnable(ep)` returns 1 if `ep` is inside a succesfully compiled runtime library.

Compilation

The function `me::Compile(ep)` compiles the node `ep` and all its subnodes. If `ep` is the empty element, all runtime libraries are compiled. See also Section 25.4 on working with `AllIdentifiers`.

Runtime identifiers are like data

To the main application, runtime identifiers are like data. Data operations such as creation, modification, destruction, read, and write are also applicable to runtime identifiers. When saving a project, the runtime libraries are **not** saved. Runtime libraries, including the data of runtime identifiers, can be saved in two ways: as separate files or in cases.

Storing runtime libraries in separate files

The runtime libraries themselves can be saved in text or binary model files using the function `me::ExportNode`. They can subsequently be read back using the functions `me::ImportLibrary` and `me::ImportNode` (see the function reference for more details on these functions). The data of the runtime identifiers can be written using a `write to file` statement and be read back using a `read from file` statement, see also Section 26.1.1.

Storing runtime libraries in cases

When saving a case, a snapshot of the data in a model, or a selection thereof (casetype), is saved. The data of a model include the runtime libraries. However, the names of the runtime identifiers can vary and therefore they cannot be part of a casetype. Whether runtime libraries are saved in a case is controlled by a global option, named `Case contains runtime libraries`. When loading a case saved with this option switched on, the previously created runtime libraries will be first destroyed and then the stored runtime libraries will be recreated, both their structure and data. When loading a case saved while this option was off, or a case saved with AIMMS 3.10 or earlier, any existing runtime libraries will be left intact. Datasets never contain runtime libraries.

The NoSave property

When the `NoSave` property is specified for a runtime library, this runtime library will not be saved in cases.

The AIMMS model explorer

To the AIMMS model explorer, the runtime libraries are read only; it can copy runtime identifiers into the main application, but it cannot modify runtime identifiers. This is because, if the AIMMS model explorer could modify runtime identifiers, the state information maintained by the main application regarding the runtime identifiers might become inconsistent with the actual state of these runtime identifiers.

Visualizing the data of runtime identifiers

When AIMMS is in developer mode, data pages of the runtime identifiers can be opened, just like data pages of ordinary identifiers. The data of runtime identifiers can also be visualized on the AIMMS pages in the following two ways:

- The safest way is to create a subset of `AllIdentifiers` containing the selected runtime identifiers, and use this subset as "implicit identifiers" in a pivot table. If the runtime identifiers referenced in this set do not yet exist, they will simply not be displayed.
- The runtime identifiers can also be directly visualized in other page objects. Care should then be taken that the visualized runtime identifiers are created with the proper index domain before a page is opened containing these identifiers; if an identifier does not exist, a page containing a reference to such an identifier will not open correctly. In order to avoid the inadvertent use of runtime identifiers on pages, they are not selectable using point and click in the identifier selector, but the identifier selector accepts them when typed in.

Limitations

The following limitations apply:

- Local declarations are not supported; only global identifiers corresponding to elements in `AllIdentifiers`.
- Quantities are not supported.
- The `source file`, `module code` and `user data` attributes are not supported.
- The current maximum number of identifiers is thirty thousand.

Appendices

Appendix A

Distributions, statistical operators and histogram functions

This chapter

This chapter provides a more elaborate description of the distributions and distribution and sample operators listed in Tables 6.5, 6.6 and 6.7. You can use this information when you want to set up an experiment around your (optimization-based) AIMMS model.

Description of distributions

For each of the available distributions we describe

- its parameters, mean and variance,
- the unit relationship between its parameters and result,
- its shape, and
- its typical use in applications.

Such information may be useful in the selection and use of a distribution to describe the particular statistical behavior of input data of experiments that you want to perform on top of your model. However, a general guideline for choosing the right might be in order and is provided in the next paragraph.

Choosing the right distribution

Whenever your experiment counts a number of occurrences, you should first make a distinction between experiments with replacement (i.e. throwing dice), experiments without replacement (i.e. drawing cards from a deck), or experiments in which independent occurrences take place at random moments (i.e. customers appearing at a desk). Having made this distinction, Table A.1 will help you to select the right distribution for your experiment. In any other case the `Normal` distribution should be considered first. Although this distribution is unbounded, it is declining so rapidly that it can often be used even when the result should be bounded. If the `Normal` distribution does not suffice, the primary selection criterium is existence of bounds: AIMMS provides the user with distributions with no bounds, one (lower) bound and two (upper and lower) bounds. See section A.2 (continuous distributions) for details.

Description of distribution operators

For each of the available distribution and sample operators we provide

- the interpretation of its result, and
- the formula for the computation of the operator.

Such information may be useful when you want to perform an analysis of the results of your experiments.

Option for backward compatibility

All distribution operators that are listed in Section A.3 have been introduced in AIMMS 3.4, although the DistributionCumulative and DistributionInverseCumulative operator were already available under the names CumulativeDistribution and InverseCumulativeDistribution, respectively. Furthermore, in order to obtain a consistent set of distribution functions the prototype for some of them has been slightly adapted. Section A.2 discusses the function prototype of the continuous distribution functions in full detail. Both the old and the new function prototypes are discussed in the AIMMS Function Reference. To make sure that models using distribution functions and developed in an older version of AIMMS are working correctly, you should set the option Distribution_compatibility to 'AIMMS 3.0'.

A.1 Discrete distributions

Discrete distributions

We start our discussion with the discrete distributions available in AIMMS. They are

- the Binomial distribution,
- the HyperGeometric distribution,
- the Poisson distribution,
- the Negative Binomial distribution, and
- the Geometric distribution.

Discrete distributions describing successes

The Binomial, HyperGeometric and Poisson distributions describe the number of times that a particular outcome (referred to as "success") occurs. In the Binomial distribution, the underlying assumption is a fixed number of trials and a constant likelihood of success. In the HyperGeometric distribution, the underlying assumption is "sampling without replacement": A fixed number of trials are taken from a population. Each element of this population denotes a success or failure and cannot occur more than once. In the Poisson distribution the number of trials is not fixed. Instead we assume that successes occur independently of each other and with equal chance for all intervals with the same duration.

	with replacement	**without replacement**	**independent occurrences at random moments**
example	throwing dice	drawing cards	serving customers
# trials until first success / time until first occurrence	`Geometric`	not supported in AIMMS	`Exponential` (continuous)
# trials until n-th success / time until n-th occurrence	`Negative Binomial`	not supported in AIMMS	`Gamma` (continuous)
# successes in fixed # trials / # successes in fixed time	`Binomial`	`Hypergeometric`	`Poisson`

Table A.1: Overview of discrete distributions in AIMMS

Distributions describing trials

The `Negative Binomial` distribution describes the number of failures before a specified number of successes have occurred. It assumes a constant chance of success for each trial, so it is linked to the `Binomial` distribution. Similarly, the distribution linked to `Poisson` distribution that describes the amount of time until a certain number of successes have occurred is known as the `Gamma` distribution and is discussed in Section A.2. The `Negative Binomial` distribution is a special case of the `Geometric` distribution and describes the number of failures before the first success occurs. Similarly, the `Exponetial` distribution is a special case of the `Gamma` distribution and describes the amount of time until the first occurrence.

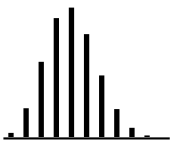
Discrete distributions overview

Table A.1 shows the relation between the discrete distributions. The continuous `Exponential` and `Gamma` distribution naturally fit in this table as they represent the distribution of the time it takes before the first/n-th occurrence (given the average time between two consecutive occurrences).

Binomial distribution

The `Binomial`(p, n) distribution:

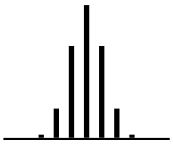
- **Input parameters** : Probability of success p and number of trials n
- **Input check** : integer $n > 0$ and $0 < p < 1$
- **Permitted values** : $\{i \mid i = 0, 1, \ldots, n\}$
- **Formula** : $P(X = i) = \binom{n}{i} p^i (1-p)^{n-i}$
- **Mean** : np
- **Variance** : $np(1-p)$
- **Remarks** : `Binomial`(p, n) = `HyperGeometric`(p, n, ∞)



A typical example for this distribution is the number of defectives in a batch of manufactured products where a fixed percentage was found to be defective in previously produced batches. Another example is the number of persons in a group voting yes instead of no, where the probability of yes has been determined on the basis of a sample.

HyperGeometric distribution

The HyperGeometric(p,n,N) distribution:

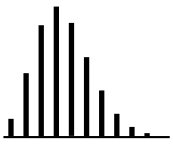


- **Input parameters** : Known initial probability of success p, number of trials n and population size N
- **Input check** : integer $n,N : 0 < n \leq N$, and $p \in \frac{1}{N}, \frac{2}{N}, \ldots, \frac{N-1}{N}$
- **Permitted values** : $\{i \mid i = 0,1,\ldots,n\}$
- **Formula** : $P(X=i) = \dfrac{\binom{Np}{i}\binom{N(1-p)}{n-i}}{\binom{N}{n}}$
- **Mean** : np
- **Variance** : $np(1-p)\frac{N-n}{N-1}$

As an example of this distribution, consider a set of 1000 books of which 30 are faulty When considering an order containing 50 books from this set, the HyperGeometric(0.03,50,1000) distribution shows the probability of observing i $(i = 0,1,\ldots,n)$ faulty books in this subset.

Poisson distribution

The Poisson(λ) distribution:

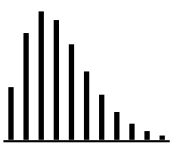


- **Input parameters** : Average number of occurrences λ
- **Input check** : $\lambda > 0$
- **Permitted values** : $\{i \mid i = 0,1,\ldots\}$
- **Formula** : $P(X=i) = \dfrac{\lambda^i}{i!}e^{-\lambda}$
- **Mean** : λ
- **Variance** : λ
- **Remarks** : Poisson$(\lambda) = \lim_{p \downarrow 0}$Binomial$(p, \lambda/p)$

The Poisson distribution should be used when there is an constant chance of a 'success' over time or (as an approximation) when there are many occurrences with a very small individual chance of 'success'. Typical examples are the number of visitors in a day, the number of errors in a document, the number of defects in a large batch, the number of telephone calls in a minute, etc.

Negative Binomial distribution

The NegativeBinomial(p,r) distribution:

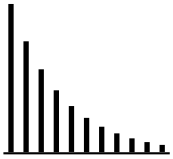


- **Input parameters** : Success probability p and number of successes r
- **Input check** : $0 < p < 1$ and $r = 1,2,\ldots$
- **Permitted values** : $\{i \mid i = 0,1,\ldots\}$
- **Formula** : $P(X=i) = \binom{r+i-1}{i} p^r (1-p)^i$
- **Mean** : $r/p - r$
- **Variance** : $r(1-p)/p^2$

Whenever there is a repetition of the same activity, and you are interested in observing the r-th occurrence of a particular outcome, then the Negative Binomial distribution might be applicable. A typical situation is going from door-to-door until you have made r sales, where the probability of making a sale has been determined on the basis of previous experience. Note that the NegativeBinomial distribution describes the number of *failures* before the r-th success. The distribution of the number of *trials* i before the r-th success is given by $P_{\texttt{NegativeBinomial(p,r)}}(X = i - r)$.

Geometric distribution

The Geometric(p) distribution:



- **Input parameters** : Probability of success p
- **Input check** : $0 < p < 1$
- **Permitted values** : $\{i \mid i = 0, 1, \ldots\}$
- **Formula** : $P(X = i) = (1 - p)^i p$
- **Mean** : $1/p - 1$
- **Variance** : $(1 - p)/p^2$
- **Remarks** : Geometric(p) = NegativeBinomial(p, 1)

The Geometric distribution is a special case of the NegativeBinomial distribution. So it can be used for the same type of problems (the number of visited doors before the first sale). Another example is an oil company drilling wells until a producing well is found, where the probability of success is based on measurements around the site and comparing them with measurements from other similar sites.

A.2 Continuous distributions

Continuous distributions

In this section we discuss the set of continuous distributions available in AIMMS.

The three distributions with both lower and upper bound are

- the Uniform distribution,
- the Triangular distribution, and
- the Beta distribution.

The five distributions with only a lower bound are

- the LogNormal distribution,
- the Exponential distribution,
- the Gamma distribution,
- the Weibull distribution, and
- the Pareto distribution.

The three unbounded distributions are

- the Normal distribution,

- the `Logistic` distribution, and
- the `Extreme Value` distribution.

Parameters of continuous distributions

Every parameter of a continuous distributions can be characterized as either a *shape* parameter β, a *location* parameter l, or a *scale* parameter s. While the presence and meaning of a shape parameter is usually distribution-dependent, location and scale parameters find their origin in the common transformation

$$x \mapsto \frac{x - l}{s}$$

to shift and stretch a given distribution. By choosing $l = 0$ and $s = 1$ the standard form of a distribution is obtained. If a certain distribution has n shape parameters ($n \geq 0$), these shape parameters will be passed as the first n parameters to AIMMS. The shape parameters are then followed by two optional parameters, with default values 0 and 1 respectively. For double-bounded distributions these two optional parameters can be interpreted as a lower and upper bound (the value of the location parameter l for these distributions is equal to the lower bound and the value of the scale parameter s is equal to the difference between the upper and lower bound). For single-bounded distributions the bound value is often used as the location parameter l. In this section, whenever the location parameter can be interpreted as a mean value or whenever the scale parameter can be interpreted as the deviation of a distribution, these more meaningful names are used to refer to the parameters. Note that the `LogNormal`, `Gamma` and `Exponential` distributions are distrubutions that will mostly be used with location parameter equal to 0.

Transformation to standard form

When transforming a distribution to standard form, distribution operators change. Section A.5 (scaling of statistical operators) gives the relationships between distribution operators working on random variables $X(l, s)$ and $X(0, 1)$.

Units of measurement

When a random variable representing some real-life quantity with a given unit of measurement (see also Chapter 32) is distributed according to a particular distribution, some parameters of that distribution are also naturally expressed in terms of this same unit while other parameters are expected to be unitless. In particular, the location and scale parameters of a distribution are measured in the same unit of measurement as the corresponding random variable, while shape parameters (within AIMMS) are implemented as unitless parameters.

Unit notation in this appendix

When you use a distribution function, AIMMS will perform a unit consistency check on its parameters and result, whenever your model contains one or more `QUANTITY` declarations. In the description of the continuous distributions below, the expected units of the distribution parameters are denoted in square brackets. Throughout the sequel, $[x]$ denotes that the parameter should have the same unit of measurement as the random variable X and [–] denotes that a parameter should be unitless.

A commonly used distribution

In practice, the Normal distribution is used quite frequently. Such widespread use is due to a number of pleasant properties:

- the Normal distribution has no shape parameters and is symmetrical,
- random values are more likely as they are closer to the mean value,
- it can be directly evaluated for any given mean and standard deviation because it is fully specified through the mean and standard deviation parameter,
- it can be used as a good approximation for distributions on a finite interval, because its probability density is declining fast enough (when moving away from the mean),
- the mean and sum of any number of uncorrelated Normal distributions are Normal distributed themselves, and thus have the same shape, and
- the mean and sum of a large number of uncorrelated distributions are always approximately Normal distributed.

Distributions for double bounded variables

For random variables that have a known lower and upper bound, AIMMS provides three continuous distributions on a finite interval: the Uniform, Triangular and Beta distribution. The Uniform (no shape parameters) and Triangular (one shape parameter) distributions should be sufficient for most experiments. For all remaining experiments, the user might consider the highly configurable Beta (two shape parameters) distribution.

Distributions for single bounded variables

When your random variable only has a single bound, you should first check whether the Gamma distribution can be used or whether the Normal distribution is accurate enough. The LogNormal distribution should be considered if the most likely value is near but not at the bound. The Weibull or Gamma distribution ($\beta > 1$), or even the ExtremeValue distribution are alternatives, while the Weibull or Gamma distribution ($\beta \leq 1$) or Pareto distribution should be considered if the bound is the most likely value.

The Gamma distribution

The Gamma (and as a special case thereof the Exponential) distribution is widely used for its special meaning. It answers the question: how long does it take for a success to occur, when you only know the average number of occurrences (like in the Poisson distribution). The Exponential distribution gives the time to the first occurrence, and its generalization, the Gamma(β) distribution gives the time to the β-th occurrence. Note that the sum of a Gamma(β_1, l_1, s) and Gamma(β_2, l_2, s) distribution has a Gamma($\beta_1 + \beta_2, l_1 + l_2, s$) distribution.

The LogNormal distribution

If you assume the logarithm of a variable to be Normal distributed, the variable itself is LogNormal-distributed. As a result, it can be shown that the chance of an outcome in the interval $[x \cdot c_1, x \cdot c_2]$ is equal to the chance of an outcome in the interval $[x/c_2, x/c_1]$ for some x. This might be a reasonable assumption in price developments, for example.

Uniform distribution

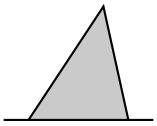
The Uniform(*min*,*max*) distribution:

- **Input parameters** : $min\,[x]$, $max\,[x]$
- **Input check** : $min < max$
- **Permitted values** : $\{x \mid min \le x \le max\}$
- **Standard density** : $f_{(0,1)}(x) = 1$
- **Mean** : $1/2$
- **Variance** : $1/12$

In the Uniform distribution all values of the random variable occur between a fixed minimum and a fixed maximum with equal likelihood. It is quite common to use the Uniform distribution when you have little knowledge about an uncertain parameter in your model except that its value has to lie anywhere within fixed bounds. For instance, after talking to a few appraisers you might conclude that their single appraisals of your property vary anywhere between a fixed pessimistic and a fixed optimistic value.

Triangular distribution

The Triangular(β,*min*,*max*) distribution:

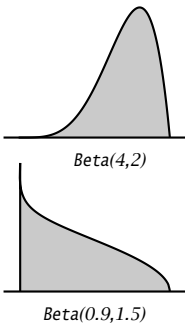


- **Input parameters** : shape β [−],$min\,[x]$, $max\,[x]$
- **Input check** : $min < max$, $0 < \beta < 1$
- **Permitted values** : $\{x \mid min \le x \le max\}$
- **Standard density** : $f_{(\beta,0,1)}(x) = \begin{cases} 2x/\beta & \text{for } 0 \le x \le \beta \\ 2(1-x)/(1-\beta) & \text{for } \beta < x \le 1 \end{cases}$
- **Mean** : $(\beta + 1)/3$
- **Variance** : $(1 - \beta + \beta^2)/18$
- **Remarks** : The shape parameter β indicates the position of the peak in relation to the range, i.e. $\beta = \frac{peak-min}{max-min}$

In the Triangular distribution all values of the random variable occur between a fixed minimum and a fixed maximum, but not with equal likelihood as in the Uniform distribution. Instead, there is a most likely value, and its position is not necessarily in the middle of the interval. It is quite common to use the Triangular distribution when you have little knowledge about an uncertain parameter in your model except that its value has to lie anywhere within fixed bounds and that there is a most likely value. For instance, assume that a few appraisers each quote an optimistic as well as a pessimistic value of your property. Summarizing their input you might conclude that their quotes provide not only a well-defined interval but also an indication of the most likely value of your property.

Beta distribution

The Beta(α,β,*min*,*max*) distribution:



- **Input parameters** : shape α [–], shape β [–], *min* [x], *max* [x]
- **Input check** : $\alpha > 0, \beta > 0, min < max$
- **Permitted values** : $\{x \mid min < x < max\}$
- **Standard density** : $f_{(\alpha,\beta,0,1)}(x) = \dfrac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1}$
  where $B(\alpha,\beta)$ is the Beta function
- **Mean** : $\alpha/(\alpha+\beta)$
- **Variance** : $\alpha\beta(\alpha+\beta)^{-2}(\alpha+\beta+1)^{-1}$
- **Remarks** : Beta(1,1,*min*,*max*)=Uniform(*min*,*max*)

The Beta distribution is a very flexible distribution whose two shape parameters allow for a good approximation of almost any distribution on a finite interval. The distribution can be made symmetrical, positively skewed, negatively skewed, etc. It has been used to describe empirical data and predict the random behavior of percentages and fractions. Note that for $\alpha < 1$ a singularity occurs at $x = min$ and for $\beta < 1$ at $x = max$.

LogNormal distribution

The LogNormal(β,*min*,*s*) distribution:

- **Input parameters** : shape β [–], lowerbound *min* [x] and scale s [x]
- **Input check** : $\beta > 0$ and $s > 0$
- **Permitted values** : $\{x \mid min < x < \infty\}$
- **Standard density** : $f_{(\beta,0,1)}(x) = \dfrac{1}{\sqrt{2\pi}x\ln(\beta^2+1)} e^{\frac{-(\ln(x^2(\beta^2+1)))}{2\ln(\beta^2+1)}}$
- **Mean** : 1
- **Variance** : β^2

If you assume the logarithm of the variable to be Normal(μ,σ)-distributed, then the variable itself is LogNormal($\sqrt{e^{\sigma^2}-1}, 0, e^{\mu-\sigma^2/2}$)-distributed. This parameterization is used for its simple expressions for mean and variance. A typical example is formed by real estate prices and stock prices. They all cannot drop below zero, but they can grow to be very high. However, most values tend to stay within a particular range. You usually can form some expected value of a real estate price or a stock price, and estimate the standard deviation of the prices on the basis of historical data.

Exponential distribution

The Exponential(*min*,*s*) distribution:

- **Input parameters** : lowerbound *min* [x] and scale s [x]
- **Input check** : $s > 0$
- **Permitted values** : $\{x \mid min \le x < \infty\}$
- **Standard density** : $f_{(0,1)}(x) = \lambda e^{-x}$
- **Mean** : 1
- **Variance** : 1
- **Remarks** : Exponential(*min*,*s*) = Gamma(1,*min*,*s*)
  Exponential(*min*,*s*) = Weibull(1,*min*,*s*)

Assume that you are observing a sequence of independent events with a constant chance of occurring in time, with s being the average time between occurrences. (in accordance with the `Poisson` distribution) The `Exponential`$(0, s)$ distribution gives answer to the question: how long a time do you need to wait until you observe the first occurrence of an event. Typical examples are time between failures of equipment, and time between arrivals of customers at a service desk (bank, hospital, etc.).

Gamma distribution

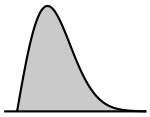
The `Gamma`(β,*min*,*s*) distribution:

- **Input parameters** : shape β [–], lowerbound *min* [x] and scale s [x]
- **Input check** : $s > 0$ and $\beta > 0$
- **Permitted values** : $\{x \mid min < x < \infty\}$
- **Standard density** : $f_{(\beta,0,1)}(x) = x^{\beta-1}e^{-x}/\Gamma(\beta)$ where $\Gamma(\beta)$ is the Gamma function
- **Mean** : β
- **Variance** : β

The `Gamma` distribution gives answer to the question: how long a time do you need to wait until you observe the β-th occurrence of an event (instead of the first occurrence as in the `Exponential` distribution). Note that it is possible to use non-integer values for β and a location parameter. In these cases there is no natural interpretation of the distribution and for $\beta < 1$ a singularity exists at $x = min$, so one should be very careful in using the `Gamma` distribution this way.

Weibull distribution

The `Weibull`(β,*min*,*s*) distribution:

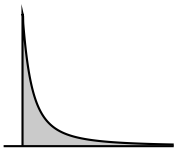


- **Input parameters** : shape β [–], lowerbound *min* [x] and scale s [x]
- **Input check** : $\beta > 0$ and $s > 0$
- **Permitted values** : $\{x \mid min \leq x < \infty\}$
- **Standard density** : $f_{(\beta,0,1)}(x) = \beta x^{\beta-1}e^{-x^{\beta}}$
- **Mean** : $\Gamma(1 + 1/\beta)$
- **Variance** : $\Gamma(1 + 2/\beta) - \Gamma^2(1 + 1/\beta)$

The `Weibull` distribution is another generalization of the `Exponential` distribution. It has been successfully used to describe failure time in reliability studies, and the breaking strengths of items in quality control testing. By using a value of the shape parameter that is less than 1, the `Weibull` distribution becomes steeply declining and could be of interest to a manufacturer testing failures of items during their initial period of use. Note that in that case there is a singularity at $x = min$.

The Pareto(β,l,s) distribution:

Pareto distribution

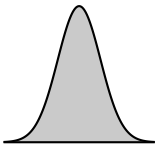


- **Input parameters** : shape β [–], location l [x] and scale s [x]
- **Input check** : $s > 0$ and $\beta > 0$
- **Permitted values** : $\{x \mid l + s < x < \infty\}$
- **Standard density** : $f_{(\beta,0,1)}(x) = \beta / x^{\beta+1}$
- **Mean** : for $\beta > 1$: $\beta/(\beta-1), \infty$ otherwise
- **Variance** : for $\beta > 2$: $\beta(\beta-1)^{-2}(\beta-2)^{-1}, \infty$ otherwise

The Pareto distribution has been used to describe the sizes of such phenomena as human population, companies, incomes, stock fluctuations, etc.

The Normal(μ,σ) distribution:

Normal distribution

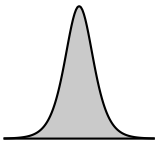


- **Input parameters** : Mean μ [x] and standard deviation σ [x]
- **Input check** : $\sigma > 0$
- **Permitted values** : $\{x \mid -\infty < x < \infty\}$
- **Standard density** : $f_{(0,1)}(x) = e^{-x^2/2}/\sqrt{2\pi}$
- **Mean** : 0
- **Variance** : 1
- **Remarks** : Location μ, scale σ

The Normal distribution is frequently used in practical applications as it describes many phenomena observed in real life. Typical examples are attributes such as length, IQ, etc. Note that while the values in these examples are naturally bounded, a close fit between such data values and normally distributed values is quite common in practice, because the likelihood of extreme values away from the mean is essentially zero in the Normal distribution.

The Logistic(μ,s) distribution:

Logistic distribution



- **Input parameters** : mean μ [x] and scale s [x]
- **Input check** : $s > 0$
- **Permitted values** : $\{x \mid -\infty < x < \infty\}$
- **Standard density** : $f_{(0,1)}(x) = (e^x + e^{-x} + 2)^{-1}$
- **Mean** : 0
- **Variance** : $\pi^2/3$

The Logistic distribution has been used to describe growth of a population over time, chemical reactions, and similar processes. Extreme values are more common than in the somewhat similar Normal distribution

The Extreme Value(l,s) distribution:

Extreme Value distribution

- **Input parameters** : Location l [x] and scale s [x]
- **Input check** : $s > 0$
- **Permitted values** : $\{x \mid -\infty < x < \infty\}$
- **Standard density** : $f_{(0,1)}(x) = e^x e^{-e^x}$

- **Mean** : $\gamma = 0.5772\ldots$ (Euler's constant)
- **Variance** : $\pi^2/6$

`Extreme Value` distributions have been used to describe the largest values of phenomena observed over time: water levels, rainfall, etc. Other applications include material strength, construction design or any other application in which extreme values are of interest. In literature the `Extreme Value` distribution that is provided by AIMMS is known as a type 1 Gumbel distribution.

A.3 Distribution operators

Distribution operators

The distribution operators discussed in this section can help you to analyze the results of an experiment. For example, it is expected that the sample mean of a sequence of observations gets closer to the mean of the distribution that was used during the observations as the number of observations increases. To compute statistics over a sample, you can use the sample operators discussed in Section A.4 or you can use the histogram functions that are explained in Section **??** of the Language Reference. The following distribution operators are available in AIMMS:

- the `DistributionCumulative(`*distr,x*`)` operator,
- the `DistributionInverseCumulative(`*distr,*α`)` operator,
- the `DistributionDensity(`*distr,x*`)` operator,
- the `DistributionInverseDensity(`*distr,*α`)` operator,
- the `DistributionMean(`*distr*`)` operator,
- the `DistributionDeviation(`*distr*`)` operator,
- the `DistributionVariance(`*distr*`)` operator,
- the `DistributionSkewness(`*distr*`)` operator, and
- the `DistributionKurtosis(`*distr*`)` operator.

Cumulative distributions ...

`DistributionCumulative(`*distr,x*`)` computes the probability that a random variable X drawn from the distribution *distr* is less or equal than x. Its inverse, `DistributionInverseCumulative(`*distr*$,\alpha$`)`, computes the smallest x such that the probability that a variable X is greater than or equal to x does not exceed α.

... and their derivatives

The `DistributionDensity(`*distr,x*`)` expresses the expected density around x of sample points drawn from a *distr* distribution and is in fact the derivative of `DistributionCumulative(`*distr,x*`)`. The `DistributionInverseDensity(`*distr*$,\alpha$`)` is the derivative of `DistributionInverseCumulative(`*distr*$,\alpha$`)`. Given a random variable X, the `DistributionInverseDensity` can be used to answer the question of how much a given value x should be increased such that the probability $P(X \leq x)$ is increased with α (for small values of α).

... for discrete distributions

For continuous distributions *distr*, $\alpha \in [0,1]$, and $x =$ DistributionInverseCumulative(*distr*, α) it holds that

$$\begin{aligned} \texttt{DistributionDensity}(\textit{distr}, x) &= \partial\alpha/\partial x \\ \texttt{DistributionInverseDensity}(\textit{distr}, \alpha) &= \partial x/\partial\alpha \end{aligned}$$

Note that the above two relations make it possible to express DistributionInverseDensity in terms of DistributionDensity. Through this relation the DistributionInverseDensity is also defined for discrete distributions.

Distribution statistics

The operators DistributionMean, DistributionDeviation, DistributionVariance, DistributionSkewness and DistributionKurtosis provide the mean, standard deviation, variance, skewness and kurtosis of a given distribution. Note that the values computed using the sample operators converges to the values computed using the corresponding distribution operators as the size of the sample increases (the law of large numbers).

A.4 Sample operators

Sample operators

The statistical sample operators discussed in this section can help you to analyze the results of an experiment. The following operators are available in AIMMS:

- the Mean operator,
- the GeometricMean operator,
- the HarmonicMean operator,
- the RootMeanSquare operator,
- the Median operator,
- the SampleDeviation operator,
- the PopulationDeviation operator,
- the Skewness operator,
- the Kurtosis operator,
- the Correlation operator, and
- the RankCorrelation operator.

Associated units

The results of the Skewness, Kurtosis, Correlation and RankCorrelation operator are unitless. The results of the other sample operators listed above should have the same unit of measurement as the expression on which the statistical computation is performed. Whenever your model contains one or more QUANTITY declarations, AIMMS will perform a unit consistency check on arguments of the statistical operators and their result.

Mean

The following mean computation methods are supported: (arithmetic) mean or average, geometric mean, harmonic mean and root mean square (RMS). The first method is well known and has the property that it is an unbiased estimate of the expectation of a distribution. The geometric mean is defined as the N-th root of the product of N values. The harmonic mean is the reciprocal of the arithmetic mean of the reciprocals. The root mean square is defined as the square root of the arithmetic mean of the squares. It is mostly used for averaging the measurements of a physical process.

- **Operator :** `Mean(`*domain*`,`*expression*`)`
- **Formula :** $\frac{1}{n}\sum_{i=1}^{n} x_i$

- **Operator :** `GeometricMean(`*domain*`,`*expression*`)`
- **Formula :** $\sqrt[n]{\prod_{i=1}^{n} x_i}$

- **Operator :** `HarmonicMean(`*domain*`,`*expression*`)`
- **Formula :** $\frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$

- **Operator :** `RootMeanSquare(`*domain*`,`*expression*`)`
- **Formula :** $\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$

Median

The median is the middle value of a sorted group of values. In case of an odd number of values the median is equal to the middle value. If the number of values is even, the median is the mean of the two middle values.

- **Operator :** `Median(`*domain*`,`*expression*`)`
- **Formula :** $median = \begin{cases} x_{\frac{N+1}{2}} & \text{if } N \text{ is odd} \\ \frac{1}{2}\left(x_{\frac{N}{2}} + x_{\frac{N+2}{2}}\right) & \text{if } N \text{ is even} \end{cases}$

Standard deviation

The standard deviation is a measure of dispersion about the mean. It is defined as the root mean square of the distance of a set of values from the mean. There are two kinds of standard deviation: the standard deviation of a sample of a population, also known as σ_{n-1} or s, and the standard deviation of a population, which is denoted by σ_n. The relation between these two standard deviations is that the first kind is an unbiased estimate of the second kind. This implies that for large n $\sigma_{n-1} \approx \sigma_n$. The standard deviation of an sample of a population can be computed by means of

- **Operator :** `SampleDeviation(`*domain*`,`*expression*`)`

■ **Formula** : $\sqrt{\frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)}$

whereas the standard deviation of a population can be determined by

■ **Operator** : PopulationDeviation(*domain*,*expression*)

■ **Formula** : $\sqrt{\frac{1}{n}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)}$

Skewness

The skewness is a measure of the symmetry of a distribution. Two kinds of skewness are distinguished: positive and negative. A positive skewness means that the tail of the distribution curve on the right side of the central maximum is longer than the tail on the left side (skewed "to the right"). A distribution is said to have a negative skewness if the tail on the left side of the central maximum is longer than the tail on the right side (skewed "to the left"). In general one can say that a skewness value greater than 1 of less than -1 indicates a highly skewed distribution. Whenever the value is between 0.5 and 1 or -0.5 and -1, the distribution is considered to be moderately skewed. A value between -0.5 and 0.5 indicates that the distribution is fairly symmetrical.

■ **Operator** : Skewness(*domain*,*expression*)

■ **Formula** : $\frac{\sum_{i=1}^{n}(x_i-\mu)^3}{\sigma_{n-1}^3}$

where μ denotes the mean and σ_{n-1} denotes the standard deviation.

Kurtosis

The kurtosis coefficient is a measure for the peakedness of a distribution. If a distribution is fairly peaked, it will have a high kurtosis coefficient. On the other hand, a low kurtosis coefficient indicates that a distribution has a flat peak. It is common practice to use the kurtosis coefficient of the standard Normal distribution, equal to 3, as a standard of reference. Distributions which have a kurtosis coefficient less than 3 are considered to be platykurtic (meaning flat), whereas distributions with a kurtosis coefficient greater than 3 are leptokurtic (meaning peaked). Be aware that in literature also an alternative definition of kurtosis is used in which 3 is subtracted from the formula used here.

■ **Operator** : Kurtosis(*domain*,*expression*)

■ **Formula** : $\frac{\sum_{i=1}^{n}(x_i-\mu)^4}{\sigma_{n-1}^4}$

where μ denotes the mean and σ_{n-1} denotes the standard deviation.

Correlation coefficient

The correlation coefficient is a measurement for the relationship between two variables. Two variables are positive correlated with each other when the correlation coefficient lies between 0 and 1. If the correlation coefficient lies between -1 and 0, the variables are negative correlated. In case the correlation coefficient is 0, the variables are considered to be unrelated to one another. Positive correlation means that if one variable increases, the other variable increases also. Negative correlation means that if one variable increases, the other variable decreases.

- **Operator :** `Correlation(`*domain*`,`*x_expression*`,` *y_expression*`)`
- **Formula :**

$$\frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sqrt{\left(n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2\right)\left(n\sum_{i=1}^{n} y_i^2 - \left(\sum_{i=1}^{n} y_i\right)^2\right)}}$$

Rank correlation

If one wants to determine the relationship between two variables, but their distributions are not equal or the precision of the data is not trusted, one can use the rank correlation coefficient to determine their relationship. In order to compute the rank correlation coefficient the data is ranked by their value using the numbers $1, 2, \ldots, n$. These rank numbers are used to compute the rank correlation coefficient.

- **Operator :** `RankCorrelation(`*domain*`,`*x_expression*`,` *y_expression*`)`
- **Formula :**

$$1 - \frac{6\sum_{i=1}^{n} (\mathrm{Rank}(x_i) - \mathrm{Rank}(y_i))^2}{n(n^2-1)}$$

A.5 Scaling of statistical operators

Transforming distributions

Shifting and scaling distribution has an effect on the distribution operators, and on sample operators when the samples are from a specified distribution. Location and scale parameters find their origin in a common transformation

$$x \mapsto \frac{x-l}{s}$$

to shift and stretch a given distribution. By choosing $l = 0$ and $s = 1$ one obtains the standard form of a given distribution, and the relation of operators working on the general and standard form of distributions is as follows:

$$
\begin{aligned}
X(l,s) &= l + sX(0,1) \\
\texttt{DistributionDensity}(x;l,s) &= \tfrac{1}{s}\,\texttt{DistributionDensity}(\tfrac{x-l}{s};0,1) \\
\texttt{DistributionInversDensity}(\alpha;l,s) &= s \cdot \texttt{DistributionInversDensity}(\alpha;0,1) \\
\texttt{DistributionCumulative}(x;l,s) &= \texttt{DistributionCumulative}(\tfrac{x-l}{s};0,1) \\
\texttt{DistributionInverseCumulative}(\alpha;l,s) &= l + s \cdot \texttt{DistributionInverseCumulative}(\alpha;0,1) \\
\texttt{Mean}(X(l,s)) &= l + s \cdot \texttt{Mean}(X(0,1)) \\
\texttt{Median}(X(l,s)) &= l + s \cdot \texttt{Median}(X(0,1)) \\
\texttt{Deviation}(X(l,s)) &= s \cdot \texttt{Deviation}(X(0,1)) \\
\texttt{DistributionVariance}(X(l,s)) &= s^2 \cdot \texttt{DistributionVariance}(X(0,1)) \\
\texttt{Skewness}(X(l,s)) &= \texttt{Skewness}(X(0,1)) \\
\texttt{Kurtosis}(X(l,s)) &= \texttt{Kurtosis}(X(0,1)) \\
\texttt{(Rank)Correlation}(X(l,s),Y) &= \texttt{(Rank)Correlation}(X(0,1),Y)
\end{aligned}
$$

Transformation of the mean

The transformation formula for the Mean holds for both the DistributionMean and the Mean of a sample. However, for the GeometricMean, the HarmonicMean and the RootMeanSquare, only the scale factor can be propagated easily during the transformation. Thus, for a sample taken from a distribution X and a any mean operator M from the GeometricMean, HarmonicMean or RootMeanSquare, it holds that

$$M(X(l,s)) = s \cdot M(X(l,1))$$

but in general

$$M(X(l,s)) \neq l + M(X(0,s))$$

Transformation of the other moments

The transformation formula for the deviation is valid for the DistributionDeviation, the SampleDeviation and PopulationDeviation, while the transformation formulae for the Skewness and Kurtosis hold for both the distribution and sample operators.

A.6 Creating histograms

Histogram

The term histogram typically refers to a picture of a number of observations. The observations are divided over equal-length intervals, and the number of observed values in each interval is counted. Each count is referred to as a frequency, and the corresponding interval is called a frequency interval. The picture of a number of observations is then constructed by drawing, for each frequency interval, the corresponding frequency as a bar. A histogram can thus be viewed as a bar chart of frequencies.

Histogram support

The procedures and functions discussed in this section allow you to create histograms based on a large number of trials in an experiment conducted from within your model. You can set up such an experiment by making use of random data for each trial drawn from one or more of the distributions discussed in the AIMMS Language Reference. The histogram frequencies, created through the functions and procedures discussed in this section, can be displayed graphically using the standard AIMMS bar chart object.

Histogram functions and procedures

AIMMS provides the following procedure and functions for creating and computing histograms.

- `HistogramCreate`(*histogram-id*[,*integer-histogram*][,*sample-buffer-size*])
- `HistogramDelete`(*histogram-id*)
- `HistogramSetDomain`(*histogram-id*,*intervals*[,*left*,*width*]
  [,*left-tail*][,*right-tail*])
- `HistogramAddObservation`(*histogram-id*,*value*)
- `HistogramAddObservations`(*histogram-id*,*values-parameter*)
- `HistogramGetFrequencies`(*histogram-id*,*frequency-parameter*)
- `HistogramGetBounds`(*histogram-id*,*left-bound*,*right-bound*)
- `HistogramGetObservationCount`(*histogram-id*)
- `HistogramGetAverage`(*histogram-id*)
- `HistogramGetDeviation`(*histogram-id*)
- `HistogramGetSkewness`(*histogram-id*)
- `HistogramGetKurtosis`(*histogram-id*)

The *histogram-id* argument assumes an integer value. The arguments *frequency-parameter*, *left-bound* and *right-bound* must be one- dimensional parameters (defined over a set of intervals declared in your model). The optional arguments *integer-histogram* (default 0), *left-tail* (default 1) and *right-tail* (default 1) must be either 0 or 1. The optional argument *sample-buffer-size* must be a positive integer, and defaults to 512.

Creating and deleting histograms

Through the procedures `HistogramCreate` and `HistogramDelete` you can create and delete the internal data structures associated with each individual histogram in your experiment. Upon success, the procedure `HistogramCreate` passes back a unique integer number, the *histogram-id*. This reference is required in the remaining procedures and functions to identify the histogram at hand. The observations corresponding to a histogram can be either continuous or integer-valued. AIMMS assumes continuous observations by default. Through the optional *integer-histogram* argument you can indicate that the observations corresponding to a histogram are integer-valued.

Sample buffer size

For every histogram, AIMMS will allocate a certain amount of memory for storing observations. By default, AIMMS allocates space to store samples of 512 observations at most. Using the optional *sample-buffer-size* argument, you can override the default maximum sample size. As long as the number of observations is still smaller than the sample buffer size, all observations will be stored individually. As soon as the actual number of observations exceeds the sample buffer size, AIMMS will no longer store the individual observations. Instead, all observations are then used to determine the frequencies of frequency intervals. These intervals are determined on the basis of the sample collected so far, unless you have specified interval ranges through the procedure `HistogramSetDomain`.

Setting the interval domain

You can use the function `HistogramSetDomain` to define frequency intervals manually. You do so by specifying

- the number of fixed-width *intervals*,
- the lower bound of the *left*-most interval (not including a left-tail interval) together with the (fixed) *width* of intervals to be created (optional),
- whether a *left-tail* interval must be created (optional), and
- whether a *right-tail* interval must be created (optional).

The default for the *left* argument is `-INF`. *Note that the* left *argument is ignored unless the* width *argument is strictly greater than 0.* Note that the selection of one or both of the tail intervals causes a corresponding increase in the number of frequency intervals to be created.

Use of tail intervals

Whenever an observed value is smaller than the lower bound of the left-most fixed-width interval, AIMMS will update the frequency count of the left-tail interval. If the left-tail interval is not present, then the observed value is lost and the procedure `HistogramAddObservation` and `HistogramAddObservations` (to be discussed below) will have a return value of 0. Similarly, AIMMS will update the frequency count of the right-tail interval, when an observation lies beyond the right-most fixed-width interval.

Adjusting the interval domain

Whenever, during the course of an experiment, the number of added observations is still below the sample buffer size, you are allowed to modify the interval ranges. As soon as the number of observations exceeds the sample buffer size, AIMMS will have fixed the settings for the interval ranges, and the function `HistogramSetDomain` will fail. This function will also fail when previous observations cannot be placed in accordance with the specified interval ranges.

Adding observations

You can use the procedure `HistogramAddObservation` to add a new observed value (or `HistogramAddObservations` to add a set of values) to a histogram. Non-integer observations for integer-valued histograms will be rounded to the nearest integer value. The procedure will fail, if the observed value cannot be placed in accordance with the specified interval ranges.

Obtaining frequencies

With the procedure HistogramGetFrequencies, you can request AIMMS to fill a one-dimensional parameter (slice) in your model with the observed frequencies. The cardinality of the index domain of the frequency parameter must be at least as large as the total number of frequency intervals (including the tail interval(s) if created). The first element of the domain set is associated with the left-tail interval, if created, or else the left-most fixed-width interval.

Interval determination

If you have provided the number of intervals through the procedure HistogramSetDomain, AIMMS will create this number of frequency intervals plus at most two tail intervals. Without a custom-specified number of intervals, AIMMS will create 16 fixed-width intervals plus two tail intervals. If you have not provided interval ranges, AIMMS will determine these on the basis of the collected observations. As long as the sample buffer size of the histogram has not yet been reached, you are still allowed to modify the number of intervals prior to any subsequent call to the procedure HistogramGetFrequencies.

Obtaining interval bounds

Through the procedure HistogramGetBounds you can obtain the left and right bound of each frequency interval. The bound parameters must be one-dimensional, and the cardinality of the corresponding domain set must be at least the number of intervals (including possible left- and right-tail intervals). The lower bound of a left-tail interval will be -INF, the upper bound of a right-tail interval will be INF.

Obtaining statistical information

The following functions provided statistical information:

- HistogramGetObservationCount The total number of observations,
- HistogramGetAverage the arithmetic mean,
- HistogramGetDeviation standard deviation,
- HistogramGetSkewness skewness, and
- HistogramGetKurtosis kurtosis coefficient.

Example

In the following example, a number of observable outputs o of a mathematical program are obtained as the result of changes in a single uniformly distributed input parameter InputRate. The interval range of every histogram is set to the interval [0,100] in 10 steps, and it is assumed that the set associated with index i has at least 12 elements.

```
for (o) do
    HistogramCreate( HistogramID(o) );
    HistogramSetDomain( HistogramID(o), intervals: 10, left: 0.0, width: 10.0 );
endfor;

while ( LoopCount <= TrialSize ) do
    InputRate := Uniform(0,1);
    solve MathematicalProgram;
    for (o) do
        HistogramAddObservation( HistogramID(o), ObservableOutput(o) );
    endfor;
endwhile;
```

```
for (o) do
    HistogramGetFrequencies( HistogramID(o), Frequencies(o,i) );
    HistogramGetBounds( HistogramID(o), LeftBound(o,i), RightBound(o,i) );
    HistogramDelete( HistogramID(o) );
endfor;
```

Appendix B

Additional Separation Procedures for Benders' Decomposition

This chapter

In Section 21.4 we showed the implementation of the procedure SeparationOptimalityAndFeasibilityDual as used by the textbook algorithm. The Benders' module implements also three other separation procedure that can be used by the Benders' decomposition algorithm depending on the setting of the control parameters UseDual and FeasibilityOnly. In this chapter we explain the implementation of these procedures.

The procedure SeparationFeasibilityOnly

The procedure SeparationFeasibilityOnly is called by the Benders' decomposition algorithm in case the primal of the Benders' subproblem is used (parameter UseDual equals 0) and if only feasibility cuts can be generated by the algorithm (parameter FeasibilityOnly equals 1). This procedure creates the feasibility problem for the (primal) subproblem if it does not exist yet. The feasibility problem is updated and solved. If its optimal objective value equals 0 (or is negative) then we have found an optimal solution for the original problem and the algorithm terminates. If the optimal objective value is larger than 0, indicating that the subproblem would have been infeasible, we add a feasibility cut to the master problem. The feasibility cut is created using the dual solution of the feasibility problem. By the dual solution we mean the shadow prices of the constraints and the reduced costs of the variables in the feasibility subproblem.

```
return when ( BendersAlgorithmFinished );

! Create feasibility problem corresponding to Subproblem (if it does not exist yet).
if ( not FeasibilityProblemCreated ) then
    gmpF := GMP::Instance::CreateFeasibility( gmpS, "FeasProb",
                              useMinMax : UseMinMaxForFeasibilityProblem );
    solsesF := GMP::Instance::CreateSolverSession( gmpF ) ;
    FeasibilityProblemCreated := 1;
endif;

! Update feasibility problem corresponding to Subproblem and solve it.
GMP::Benders::UpdateSubProblem( gmpF, gmpM, 1, round : 1 );

GMP::SolverSession::Execute( solsesF ) ;
GMP::Solution::RetrieveFromSolverSession( solsesF, 1 ) ;
```

```
! Check whether objective is 0 in which case optimality condition is satisfied.
ObjectiveFeasProblem := GMP::SolverSession::GetObjective( solsesF );

if ( ObjectiveFeasProblem <= BendersOptimalityTolerance ) then
    if ( MasterHasBeenSolved ) then
        return AlgorithmTerminate( 'Optimal' );
    endif;
endif;

! Add feasibility cut to the Master problem.
NumberOfFeasibilityCuts += 1;
GMP::Benders::AddFeasibilityCut( gmpM, gmpF, 1, NumberOfFeasibilityCuts );
```

The procedure SeparationOptimalityAndFeasibility

The procedure SeparationOptimalityAndFeasibility is called by the Benders' decomposition algorithm in case the primal of the Benders' subproblem is used (parameter UseDual equals 0) and if both optimality and feasibility cuts can be generated by the algorithm (parameter FeasibilityOnly equals 0). This procedure updates the primal subproblem and solves it. If the primal subproblem is infeasible then this procedure creates a feasibility problem for the subproblem if it does not exist yet. The feasibility problem is updated and solved, and its dual solution is used to created a feasibility cut which is added to the master problem. If the primal subproblem is bounded and optimal then the objective value of the subproblem is compared to the objective value of the master problem to check whether the algorithm has found an optimal solution for the original problem. If the solution is not optimal yet then an optimality cut is added to the master problem, using the dual solution of the primal subproblem.

```
return when ( BendersAlgorithmFinished );

! Update Subproblem and solve it.
GMP::Benders::UpdateSubProblem( gmpS, gmpM, 1, round : 1 );

GMP::SolverSession::Execute( solsesS ) ;
GMP::Solution::RetrieveFromSolverSession( solsesS, 1 ) ;

ProgramStatus := GMP::Solution::GetProgramStatus( gmpS, 1 ) ;

if ( ProgramStatus = 'Unbounded' ) then
    return AlgorithmTerminate( 'Unbounded' );
endif;

if ( ProgramStatus = 'Infeasible' ) then

    ! Create (if it does not exist yet) and update feasibility problem corresponding to
    ! Subproblem, and solve it to create feasibility cut for the Master problem.
    if ( not FeasibilityProblemCreated ) then
        gmpF := GMP::Instance::CreateFeasibility( gmpS, "FeasProb",
                                  useMinMax : UseMinMaxForFeasibilityProblem );
        solsesF := GMP::Instance::CreateSolverSession( gmpF ) ;
        FeasibilityProblemCreated := 1;
    endif;

    GMP::Benders::UpdateSubProblem( gmpF, gmpM, 1, round : 1 );
```

```
    GMP::SolverSession::Execute( solsesF ) ;
    GMP::Solution::RetrieveFromSolverSession( solsesF, 1 ) ;

    ! Add feasibility cut to the Master problem.
    NumberOfFeasibilityCuts += 1;
    GMP::Benders::AddFeasibilityCut( gmpM, gmpF, 1, NumberOfFeasibilityCuts );

else

    ! Check whether optimality condition is satisfied.
    ObjectiveSubProblem := GMP::SolverSession::GetObjective( solsesS );

    if ( SolutionImprovement( ObjectiveSubProblem, BestObjective ) ) then
        BestObjective := ObjectiveSubProblem;
    endif;

    if ( SolutionIsOptimal( ObjectiveSubProblem, ObjectiveMaster ) ) then
        return AlgorithmTerminate( 'Optimal' );
    endif;

    ! Add optimality cut to the Master problem.
    NumberOfOptimalityCuts += 1;
    GMP::Benders::AddOptimalityCut( gmpM, gmpS, 1, NumberOfOptimalityCuts );

endif;
```

The procedure SeparationFeasibilityOnlyDual

The procedure SeparationFeasibilityOnlyDual is called by the Benders' decomposition algorithm in case the dual of the Benders' subproblem is used (parameter UseDual equals 1) and if only feasibility cuts can be generated by the algorithm (parameter FeasibilityOnly equals 1). This procedure updates the dual subproblem and solves it. If its optimal objective value equals 0 (or is negative) then we have found an optimal solution for the original problem and the algorithm terminates. If the optimal objective value is larger than 0 then we create a feasibility cut using the level values of the variables in the solution of the dual subproblem. This feasibility cut is added to the master problem.

```
return when ( BendersAlgorithmFinished );

! Update Subproblem and solve it.
GMP::Benders::UpdateSubProblem( gmpS, gmpM, 1, round : 1 );

GMP::SolverSession::Execute( solsesS ) ;
GMP::Solution::RetrieveFromSolverSession( solsesS, 1 ) ;

! Check whether objective is 0 in which case optimality condition is satisfied.
ObjectiveSubproblem := GMP::SolverSession::GetObjective( solsesS );

if ( ObjectiveSubproblem <= BendersOptimalityTolerance ) then
    if ( MasterHasBeenSolved ) then
        return AlgorithmTerminate( 'Optimal' );
    endif;
else
    ! Add feasibility cut to the Master problem.
    NumberOfFeasibilityCuts += 1;
    GMP::Benders::AddFeasibilityCut( gmpM, gmpS, 1, NumberOfFeasibilityCuts );
endif;
```

Index

Symbols

A

D

E

F

G

H

K

L

N

O

P

S

V

W

X

Z

Bibliography

[Al03] F. Altenstedt, *Memory consumption versus computational time in nested benders decompostion for stochastic linear programmings*, Tech. report, Chalmers University of Technology, Göteborg, Sweden, 2003. 19.4.2

[Ba01] P. Baptiste, C. Le Pape, and W. Nuijten, *Constraint-based scheduling*, Kluwer Academic Publishers, 2001. 22

[Be62] J.F. Benders, *Partitioning procedures for solving mixed-variables programming problems*, Numerische Mathematic **4** (1962), 238–252. 21

[Bi97] J.R. Birge and F. Louveaux, *Introduction to stochastic programming*, Springer, New York, 1997. 19

[BT09] A. Ben-Tal, L. El Ghaoui, and A. Nemirovski, *Robust optimization*, Princeton University Press, Princeton, N.J., 2009. 20, 20.3, 20.3

[Ch04] J.W. Chinneck, *The constraint consensus method for finding approximately feasible points in nonlinear programs*, INFORMS Journal on Computing **16** (2004), 255–265. 16.2, 17.2, 17.3.9

[De98] M. Dempster and R. Thompson, *Parallelization and aggregation of nested benders decomposition*, Annals of Operations Research **81** (1998), 163–187. 19.4.2

[Du86] M.A. Duran and I.E. Grossmann, *An outer-approximation algorithm for a class of mixed-integer nonlinear programs*, Mathematical Programming **36** (1986), 307–339. 18, 18.2

[Fi10] M. Fischetti, D. Salvagni, and A. Zanette, *A note on the selection of benders' cuts*, Mathematical Programming B **124** (2010), 175–182. 21.2, 21.5.3

[Fo94] R. Fourer and D.M. Gay, *Experience with a primal presolve algorithm*, Large Scale Optimization: State of the Art (W.W. Hager, D.W. Hearn, and P.M. Pardalos, eds.), Kluwer Academic Publishers, 1994. 17.1

[Ka87] A.H.G. Rinnooy Kan and G.T. Timmer, *Stochastic global optimization methods; part II: Multi level methods*, Mathematical Programming **37** (1987), 57–78. 17.2

[Ka05] P. Kall and J. Mayer, *Stochastic linear programming: Models, theory, and computation*, Springer, New York, 2005. 19

[Ma99] R.K. Martin, *Large scale linear and integer optimization: A unified approach*, Kluwer Academic Publishers, Norwell, 1999. 21.3

[Ne88] G.L. Nemhauser and L.A. Wolsey, *Integer and combinatorial optimization*, John Wiley & Sons, New York, 1988. 21.3

[Qu92] I. Quesada and I.E. Grossmann, *An lp/nlp based branch and bound algo-*

rithm for bonvex minlp optimization problems, Computers and Chemical Engineering **16** (1992), 937–947. 18, 18.5

[Ro06] F. Rossi, P. Van Beek, and editors T. Walsh, *Handbook of constraint programming*, Elsevier, 2006. 22

[Sa94] M.W.P. Savelsbergh, *Preprocessing and probing techniques for mixed integer programming problems*, ORSA Journal on Computing **6** (1994), 445–454. 17.1