AIMMS

The Language Reference

AIMMS 3.1

May 31, 2000

AIMMS

The Language Reference

Paragon Decision Technology

Johannes Bisschop
Marcel Roelofs

Paragon Decision Technology B.V.
P.O. Box 3277
2001 DG Haarlem
The Netherlands
Tel.: +31(0)23-5511512
Fax: +31(0)23-5511517
Email: info@paragon.nl
WWW: http://www.aimms.com

ISBN 90–801516–3–7

This documentation was typeset by Paragon Decision Technology B.V. using LATEX and the LUCIDA font family.

About Paragon

During the eighties it became apparent that the available algebraic modeling systems did not meet modern standards for openness and ease-of-use as required by industry and government. There was a clear need for new advanced facilities to generate model-based decision support systems. Against this background, Paragon Decision Technology B.V. was formed. Its goal is to produce general purpose and powerful tools for modelers and decision makers.

The founder, Johannes J. Bisschop, was at this time a research mathematician at Shell Research in Amsterdam, and a professor of computational optimization and modeling at the University of Twente, The Netherlands. Today he combines his part-time position at the University of Twente with managing Paragon Decision Technology B.V. and the continuing development of AIMMS.

The first commercial AIMMS system was introduced onto the market at the end of 1993, since which time AIMMS has established itself as a solid professional tool for model-based decision support systems. Several major customers with international reputations have contracted Paragon Decision Technology B.V. to implement important additions to the functionality of AIMMS which they felt were important to their particular project requirements. This has resulted in generic improvements adding to the maturity of AIMMS as it stands today. As well as customizing AIMMS Paragon Decision Technology B.V. also executes projects centered around model building and end-user GUI design.

Today, AIMMS is used on nearly every continent by companies, government agencies and universities. It is mostly used for scheduling and planning applications in such areas as production, distribution, finance, forestry, and energy. Amongst AIMMS users are to be found such household names as Amoco, Heineken and Shell.

Contents

Preface

What is AIMMS*?*

The acronym AIMMS stands for

Advanced Integrated Multidimensional Modeling Software.

It is a new type of mathematical modeling software—a combination of a modeling language, a graphical interface, and the most powerful numerical solvers. You may use it to create user-friendly applications in which the most complex issues are concealed. The software is constructed to run in different modes to support two primary user groups: modelers (application builders) and end-users (decision makers). AIMMS gives you the ability to place all of the power of the most advanced mathematical modeling directly into the hands of the people who need it to make decisions.

What is in the AIMMS documentation

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 on-line

In addition to the printed versions, these books are also available on-line in the ADOBE Portable Document Format (PDF). Additions to the system and small changes in its functionality are always directly reflected in the on-line documentation, but not necessarily in the printed material. Therefore, the on-line versions should be considered as the authoritative documentation describing the functionality of the AIMMS system.

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 a 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 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.

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.

Demo and example projects

In addition to the documentation, the AIMMS systems is accompanied by an interactive demo, as well as a set of example projects. The interactive demo provides you with a brief tour through the AIMMS system, illustrating its basic functionality. The example projects serve two different purposes:

- they give you an idea of AIMMS' capabilities to create a complete model-based end-user application, and
- they illustrate specific aspects of both the AIMMS modeling language and end-user interface building facilities.

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—*Semantics of sparse execution*—explains the possible interpretations that can be given to various types of expressions in a sparse execution environment, and provides a number of convenience operators to distinguish between them.
- Chapter 13—*Execution efficiency*—focuses on the efficiency aspects that should be taken into consideration for a successful implementation of advanced, large-scale AIMMS applications.

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 16—*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—*Matrix Manipulation*—describes a list of procedures which allow you to implement efficient algorithms for the sequential solution of linear and mixed-integer linear programming models.

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 ASCII reports.

- Chapter 18—*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 19—*The `READ` and `WRITE` statements*—describes the basic mechanism offered by AIMMS for data transfer with various data sources.
- Chapter 20—*Communicating with databases*—discusses the specific aspects of setting up a link between AIMMS and a database.
- Chapter 21—*Format of ASCII data files*—presents the various data formats offered by AIMMS for initializing a model through a number of ASCII data files.
- Chapter 22—*ASCII reports and listing*—describes the statements and formatting options available for producing standardized and customized ASCII 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 23—*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 24—*Time-Based Modeling*—describes the advanced concepts in AIMMS to deal with time-dependent data and models in a flexible and easy manner.
- Chapter 25—*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.

Acknowledgements

Acknowledgements

Many people have both knowingly and unknowingly helped in the development of this manual and the AIMMS system by their contributions, suggestions and opinions expressed over the last twenty years. We would like to mention a few by name.

First of all we would like to thank *Alexander Meeraus*, now president of GAMS Development Corporation, who has been a source of inspiration during the time that he and the first author, Johannes Bisschop, were collaborators at the World Bank in the mid seventies. Other people who have made a major contribution to our modeling software with their ideas are (in alphabetical order) *Robert Fourer* of Northwestern University, *David Gay* of AT&T, *Art Geoffrion* of UCLA, *Harvey Greenberg* of the University of Colorado, *Tony Hürlimann* of the University of Fribourg, and *Gautam Mitra* of Brunel University.

We owe special thanks to Robert Entriken for his contribution to bringing about the AIMMS 2.0 manual during his six-month stay at Paragon Decision Technology B.V. in 1993, and for his detailed comments and suggestions on some of the chapters of the current manual during the initial AIMMS 3.0 documentation efforts in 1996.

The AIMMS system would never have been realized if there had not been several companies in the Netherlands and North America, who entrusted us with their modeling-related projects. Several users of AIMMS 2.0 have made extensive comments about that system, thereby contributing to several of the new developments in AIMMS 3.0. A selected few of them are: Steve Kleinman of Amoco; Dan Streiffert and David Sun of Esca; Reinier Huizer and Jacob Shakouri of Heineken; Michel Draper of KLM; Nico van den Hijligenberg and Thorsten Gragert of SFISS Financial Technology; Rob Davis of the Ontario Ministry of Natural Resources; Koos Ballintijn, Peter Bost, Rafi Maman, Weijian Mo, and Nort Thijssen of Shell.

The authors

The authors

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. Since 1989 he combines his part-time position at the University with managing Paragon Decision Technology B.V. and the continuing development of AIMMS. His main interests are in the areas of computational optimization and modeling.

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 technical director of Paragon Decision Technology B.V. His main responsibilities have been the design and documentation of the AIMMS 3.0 language and interface.

Contributors to AIMMS

Contributors to AIMMS

Developing and documenting a modeling system such as AIMMS is hard work. This work has been carried out by the employees of Paragon Decision Technology B.V. over the last nine years. Several of them have contributed to the system by developing code and/or contributing to parts of the documentation. Without their willingness to work well as a team, both the AIMMS 3.0 system and the documentation would not have become a reality.

The authors would like to express their special gratitude to the following persons for their assistance in getting both the system and the documentation to a consistent state.

- Richard Stegeman and Koos Heerink, the main contributors on the implementation of the graphical interface of AIMMS, have also contributed considerably to the User's Guide and help files.
- Chris Kuip, the main contributor on the implementation of the modeling layer of AIMMS, has also contributed to the overall content of the Language Reference through his ideas, comments and careful reading.
- John Boers and Marcel Hunting, responsible for the linkage with solvers and other external components, have contributed to the documentation describing these links.
- Gertjan de Lange, Pim Beers, Peter Bonsma and Thorsten Gragert have contributed with various helpful comments and innovative ideas based on their extensive use and testing of preliminary versions of AIMMS 3.0.
- Pim Beers, Gertjan Kloosterman and Jacques de Swart have contributed to the online documentation of AIMMS.

- Gertjan de Lange has compiled the demo models and applications which complement the AIMMS documentation.
- Koos Heerink has made major contributions to the book on Optimization Modeling in terms of textual improvements and by implementing and verifying the models described in that book.
- Nico van den Hijligenberg has contributed substantially to the chapter "A Portfolio Selection Problem" in the book on Optimization Modeling.
- Martine Uyterlinde has compiled most of the material in the chapter "Designing End-User Interfaces" of the User's Guide.
- Michelle Chamalaun and Giles Stacey performed the editing and proofreading of this manual. We are grateful for their co-operation and flexibility in working to our tight schedule.

List of Modifications

List of modifications

The following list describes the differences between the current electronic version of the Language Reference and the printed version of September 1999.

- Section 4.1: added documentation for the `Double` property of parameters.
- Section 5.2.2: added documentation for the new optional argument *create* of the `ElementCast` function.
- Section 5.3.2: added documentation for the new `%u` conversion code for the `FormatString` function.
- Section 5.3.3: added documentation for the new optional argument *create* of the `StringToElement` function.
- Section 6.2.2: clarified the syntax diagram of the *expression-inclusion* expression.
- Section 10.1: added documentation for the property `UndoSafe` for internal procedures.
- Section 10.3: clarified paragraph on the use of global subdomains.
- Section 10.3.1: clarified the requirements and example of the `APPLY` operator.
- Section 11.2: added documentation for the property `UndoSafe` for external procedures.
- Section 14.1: clarified the interpretation of the `RANGE` attribute of variables.
- Section 18.3: clarified the syntax diagrams of the `EMPTY`, `CLEANUP` and `CLEANDEPENDENTS` statements for consistency.
- Section 19.2: clarified the paragraph on specifying a selection of a `READ` or `WRITE` statement.
- Section 20.5: made a minor modification to the `SQL query` attribute in the declaration of the example database procedure `DeleteTableVersion`.
- Section 23.5: clarified the documentation about local unit overrides in expressions.
- Section 24.3: clarified the use of lag and lead operators in conjunction with horizons.
- Section 24.4: added documentation for the new mandatory argument *length-dominates* of the function `CreateTimeTable`, and modified the corresponding examples accordingly.
- Section 24.6: modified the examples to reflect the modified syntax of the function `CreateTimeTable`.

- Section 24.9: clarified the documentation of the function `MomentToString` and `StringToMoment`.
- Section 25.1: made some minor modifications to the declaration of and call to the external procedure `PrintParameterInfo`.

Part I

Preliminaries

Chapter 1

Introduction to the AIMMS language

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.

Example makes a good starting point

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*.

Familiarity with algebraic notation

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.

What to expect in this chapter

- 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

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:

The modeling process

- 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:
   identifier :  Depots
   index      :  d
SET:
   identifier :  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 *index domain* of this quantity.

Example

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

```
PARAMETER:
   identifier   :  Distance
   index domain :  (d,c)
```

In this example the identifier Distance is referred to as an *indexed 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:
   identifier   :  MaxDeliveryDistance
PARAMETER:
   identifier   :  UnitTransportRate
```

Compound sets

The sets Depots and Customers declared above are examples of *simple* sets containing *atomic* elements only. In some modeling applications, however, it is also natural to consider *compound* sets containing *tuples* of atomic elements. A good example of a compound set in the depot location problem is the set of all permitted *routes* from depots d to customers c.

Example

In AIMMS, the compound set Routes can be declared as follows.

```
SET:
   identifier   :  Routes
   subset of    :  (Depots, Customers)
   index        :  r
   definition   :  { (d,c) | Distance(d,c) <= MaxDeliveryDistance }
```

Explanation

In the SUBSET OF attribute of the above declaration it is indicated that the set Routes is a subset of the Cartesian product of the simple sets Depots and Customers. The DEFINITION attribute globally defines the set Routes 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.

Indexing over compound sets

In the declaration above, the set Routes has been given its own *compound* index r. You can use this compound index in index domains to indicate that a particular identifier is defined only for tuples (d,c) within the set Routes. A good example in the depot location problem is the identifier representing the *unit transport cost*. Obviously, such a quantity only needs to be considered for the existing Routes.

Example

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

```
PARAMETER:
   identifier   :  UnitTransportCost
   index domain :  r
   definition   :  UnitTransportRate * Distance(r)
```

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.

Compound index versus index tuples

In the above definition, the parameter `Distance` is referenced using the compound index `r` as opposed to the index tuple (`d`,`c`) used in the index domain of its declaration. AIMMS lets you use these two interchangeably, as long as the underlying domains refer to tuples with the same components. In the case of `UnitTransportCost(r)`, any reference to `UnitTransportCost(d,c)` for tuples (`d`,`c`) outside the set `Routes` is not defined, and AIMMS will evaluate this quantity to 0. For a discussion of the implications you are referred to Chapter 12.

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:
   identifier   :  DepotRentalCost
   index domain :  d
PARAMETER:
   identifier   :  DepotCapacity
   index domain :  d
PARAMETER:
   identifier   :  CustomerDemand
   index domain :  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 route `r` representing the level of transport on it.

Example

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

```
VARIABLE:
   identifier   :  DepotSelected
   index domain :  d
   range        :  binary
VARIABLE:
   identifier   :  Transport
   index domain :  r
   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:
   identifier   :  CustomerDemandRestriction
   index domain :  c
   definition   :  Sum[ d, Transport(d,c) ] >= CustomerDemand(c)
CONSTRAINT:
   identifier   :  DepotCapacityRestriction
   index domain :  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 `r`. As a result, the transport from a depot `d` to a customer `c` not in the set `Routes` 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:
   identifier   :  TotalCost
   definition   :  Sum[ d, DepotRentalCost(d)*DepotSelected(d) ] +
                   Sum[ r, UnitTransportCost(r)*Transport(r) ]
```

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.

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.

```
MATHEMATICAL PROGRAM:
   identifier   :  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,
- ASCII 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 ASCII format

The following fictitious data set is provided in the form of an ASCII data file. It illustrates the basic constructs available for providing data in ASCII 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 ASCII 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 MainInitialization. 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 `Routes`, 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 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.

This reference guide

2.1 Managing your model

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

Models in AIMMS

- 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.

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.

Optimization included ...

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

... but not necessary

- 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 binary files with the "`.amb`" (AIMMS model base) 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.

Backups

After each editing session AIMMS will save a backup of the last version of your model file in the `Backup` subdirectory of your project directory. Along with this backup AIMMS can create additional ASCII files with the "`.aim`" extension. These files contain an ASCII representation of the model and are created for your convenience only.

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 of a particular *data type*. AIMMS supports the following data types:

- *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,
- *macro*—macro facility using expression substitution,
- *assertion*—a logical condition to verify data integrity,
- *procedure* and *function*—code segments to initiate execution,
- *external procedure* and *function*—procedures and functions in an external DLL linked to a model,
- *database table*—a table in an external data source for data retrieval and storage,
- *database procedure*—link to an SQL query or stored procedure in an external data source,
- *calendar* and *horizon*—special time sets to automate data aggregation and disaggregation,

- *file*—an external file, and
- *quantity* and *convention*—a specification for automatic unit analysis and conversion.

Declaration forms

The declarations of all sections, 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
```

Global name space

AIMMS operates with a global name space for all declared identifiers, except those that are declared within procedures and functions. You are not allowed to declare the keywords in AIMMS as identifiers. AIMMS will produce a compiler error when you try to do so.

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 Chapter 23.

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 *compound* (multidimensional). 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 compound set 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.

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 21.2 and 21.3).

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 16, 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, or
- a logical value.

Expressions are discussed in full detail in Chapters 5 and 6.

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 *`INITIAL DATA` attribute* of sets, parameters and variables,
- by reading in data from an *ASCII 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 `INITIAL DATA` attribute.
- Subsequently, AIMMS will execute the predefined procedure `MainInitialization`. You can use it to specify `READ` statements to read in data from ASCII 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 18.

Entering the `INITIAL DATA` attribute

The `INITIAL DATA` 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 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, compound sets and indexed sets.

This chapter

3.1 Sets and indices

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.

General

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.

Example

$$(\forall j \in Cities) \qquad Arrival_j = \sum_{i \in Cities} Transport_{ij}.$$

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*.

A set in AIMMS

Several types of sets

- has either *strings* or *integers* as elements,
- is either a *simple* set or a *compound* set, 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 compound

A *simple* set is a one-dimensional set, such as the set *Cities* mentioned above, while a *compound* or multidimensional set is the Cartesian product of a number of simple sets or a subset thereof. An example 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*.

Indexed sets

Both simple and compound sets may or may not 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.

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.

Compound indexing

As you shall see, you can use both simple and compound sets for indexing. If you think, for example, that it is more convenient to express your model components in terms of an index r into the compound set *Routes* rather than in terms of tuples (i, j) of cities, AIMMS allows you to do so.

Tags for component reference

If you use an index into a compound set, AIMMS still lets you access the individual components of the underlying tuple by means of user-definable *tags*. With tags you can easily construct subtuples in any order you desire.

Sorting of sets

The contents of any set 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 INDEX DOMAIN and TAGS are only relevant to compound sets and indexed sets, respectively.

Attribute	Value-type	See also page
INDEX DOMAIN	*index-domain*	37
SUBSET OF	*subset-domain*	
INDEX	*identifier-list*	
PARAMETER	*identifier-list*	
TAGS	*tags*	36
TEXT	*string*	19
COMMENT	*comment string*	19
PROPERTY	NoSave	
DEFINITION	*set-expression*	
ORDER BY	*expression-list*	

Table 3.1: SET attributes

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 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:
   identifier : 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:
   identifier : SupplyCities
   subset of  : Cities
   parameter  : LargestSupplyCity
   text       : "The subset of cities that act as supply city"
   definition : { i | Exists( j | Transport(i,j) ) }
   order by   : 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 SUBSET OF attribute

With the `SUBSET OF` 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 ORDER BY attribute

With the ORDER BY attribute you can indicate that you want the elements of a certain set to be ordered according to a single or multiple ordering criteria. Both simple and compound 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 ORDER BY attribute of a root set may influence overall execution times of your model in a negative manner. Section 13.7.1 discusses these efficiency aspects in more detail, and provides alternative solutions.

Ordering criteria

The value of the ORDER BY 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 ORDER BY 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 ORDER BY 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 ORDER BY 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:
   identifier : LexicographicSupplyCities
   subset of  : SupplyCities
   order by   : i ;
SET:
   identifier : ReverseLexicographicSupplyCities
   subset of  : SupplyCities
   order by   : - i ;
SET:
   identifier : SupplyCitiesByIncreasingTransport
   subset of  : SupplyCities
   order by   : sum( j, Transport(i,j) ) ;
SET:
   identifier : SupplyCitiesByDecreasingTransportThenLexicographic
   subset of  : SupplyCities
   order by   : - 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.

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 8.4.

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:
   identifier : SupplyCities
   subset of  : 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 `SUBSET OF` 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.

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:
    identifier : LowInt
    range      : Integer ;
PARAMETER:
    identifier : HighInt
    range      : Integer ;
SET:
   identifier  : EvenNumbers
   subset of   : Integers
   index       : i
   parameter   : LargestPolynomialValue
   order by    : - 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 `ORDER BY` 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 Compound sets

Definition

A *compound* or multidimensional set is the Cartesian product of a number of simple sets or a subset thereof. Compound sets are typically used as the domain space for multidimensional identifiers.

Tuples and index components

An element of a compound set 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.

Compound indexing

Like simple sets, the elements of a compound set can be referenced using a single index. You can also use an *index tuple*, whereby each tuple component contains an index corresponding to a simple set.

Compound set attributes

Compound sets support all the attributes of a simple set. In addition, compound sets support the `TAGS` attribute, which enables you to access and use the individual components of an element tuple in expressions and statements.

The `SUBSET OF` attribute

The `SUBSET OF` attribute for compound sets is mandatory, 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 compound set, just the name of that set.

Example

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

```
SET:
   identifier : ConnectedCities
   subset of  : (Cities, Cities)
   index      : cc
   definition : { (i,j) | Distance(i,j) > 0 } ;
SET:
```

```
   identifier : ExpensiveConnections
   subset of  : ConnectedCities
   definition : { cc | TransportCost(cc) > 100 };
```

Accessing compound set elements

The above example shows that you can specify indices and element parameters for compound sets just as you could for simple sets. They are called *compound indices* and *compound element parameters*. As long as the domains are compatible, AIMMS lets you unrestrictedly interchange compound indices with tuples of indices as illustrated in the definitions of `ConnectedCities` and `ExpensiveConnections`.

The TAGS attribute

When you use compound indices or compound element parameters, you still have access to individual index components through the use of *tags*, which can be declared in the `TAGS` attribute of a compound set. Tags provide a placeholder for every individual component of a tuple in a compound set to be used in expressions and assignments. To reference a specific index component or subtuple, one can simply use the fairly standard dot notation followed by a single tag or a tuple of tags.

Examples

Consider an extension of the declaration of the set `ConnectedCities` with a `TAGS` attribute as follows.

```
SET:
   identifier : ConnectedCities
   subset of  : (Cities, Cities)
   index      : cc
   tags       : (orig, dest)
   comment    : "The tag orig for the first index component stands
                 for originating cities, while the second index
                 component dest stands for destination cities." ;
```

The following compound set declarations demonstrate the use of these tags.

```
SET:
   identifier : LinksWithSupplyCities
   subset of  : ConnectedCities
   definition : { cc | cc.orig in SupplyCities } ;
SET:
   identifier : BidirectionalLinks
   subset of  : ConnectedCities
   definition : { cc | cc.(dest,orig) in ConnectedCities } ;
```

Note that the second example above reverses the tag order. As a result, only those connections that also have a reverse link are selected.

Uniqueness

The tag names that you specify in the `TAGS` attribute are AIMMS identifiers, and hence must be unique within your application. The length of a tag tuple (i.e. the number of components) must coincide with the dimension of the compound set. Within the tree of a compound root set and all of its subsets, you can only specify a tuple of tag names for the root set. However, you are allowed to use these tags for all subsets in the entire tree.

Ordering compound sets

By default, compound sets are ordered componentwise from left to right, and consistent with the ordering of the underlying simple root sets. By using one or more ordering criteria, you can overrule this default ordering, as illustrated by the following declaration.

Example

```
SET:
    identifier : LinksByDistanceThenDoublyLexicographic
    subset of  : ConnectedCities
    definition : ConnectedCities
    order by   : Distance(cc), cc.orig, cc.dest ;
```

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 INDEX DOMAIN attribute

A set becomes an indexed set by specifying a value for the INDEX DOMAIN 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 INDEX DOMAIN attribute is discussed on page 41.

Example

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

```
SET:
    identifier   : SupplyCitiesToDestination
    index domain : j
    subset of    : Cities
    definition   : { i | Transport(i,j) } ;
SET:
    identifier   : DestinationCitiesFromSupply
    index domain : i
    subset of    : Cities
    definition   : { j | Transport(i,j) } ;
SET:
    identifier   : IntermediateTransportCities
    index domain : (i,j)
    subset of    : 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, a compound set identifier, or another family of indexed simple or compound sets of the same or lower dimension. The subset domain of an indexed set *cannot* be an explicit Cartesian product of sets.

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 or compound 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.

Attribute	Value-type	See also page
RANGE	*set-identifier*	
TEXT	*string*	19
COMMENT	*comment string*	19

Table 3.2: INDEX attributes

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.

Example

The following declaration illustrates a direct INDEX declaration.

```
INDEX:
    identifier : 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

Parameters in AIMMS can hold data of the following four data types:

- numeric values,
- strings,
- set elements, and
- 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
INDEX DOMAIN	*index-domain*	
RANGE	*range*	
DEFAULT	*constant-expression*	
UNIT	*unit expression*	
PROPERTY	NoSave *numeric-storage-property*	
TEXT	*string*	19
COMMENT	*comment string*	19, 32
DEFINITION	*expression*	33

Table 4.1: PARAMETER attributes

Basic examples

The following declarations demonstrate some basic numeric parameter declarations

```
PARAMETER:
     identifier   : Population
     index domain : i
     range        : [0,inf)
     unit         : [ 1000 ]
     text         : "Population of city i in thousands" ;
PARAMETER:
     identifier   : Distance
     index domain : (i,j)
     range        : [0,inf)
     unit         : [ km ]
     text         : "Distance from city i to city j in km" ;
```

The INDEX DOMAIN attribute

For each multidimensional identifier you need to specify its dimensions by providing a list of index bindings at the INDEX DOMAIN attribute. Identifiers without an INDEX DOMAIN are said to be *scalar*. In the index domain you can specify default or local bindings to either simple or compound 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 INDEX DOMAIN 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 DestinationCities-FromSupply(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:
    identifier   : UnitTransportCost
    index domain : (i,j) ;
PARAMETER:
    identifier   : UnitTransportCostWithCondition
    index domain : (i,j) in ConnectedCities ;
PARAMETER:
    identifier   : UnitTransportCostWithCompoundDomain
    index domain : cc ;
PARAMETER:
    identifier   : UnitTransportCostWithIndexedDomain
    index domain : (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 $\times$ 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.
- The identifier UnitTransportCostWithCompoundDomain is defined over the two-dimensional compound set ConnectedCities by the default binding of the index cc. Although the declaration seems equivalent to that of

UnitTransportCostWithCondition, AIMMS will now produce a domain error, when you try to make assignments to tuples outside of this set.

- 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 .Domain suffix

For every indexed identifier you have access to the current contents of the INDEX DOMAIN attribute through the .Domain suffix associated with the identifier at hand. The .Domain suffix behaves like a set of the dimension of its associated identifier. You can use it, for instance, in the INDEX DOMAIN of another identifier to indicate that two identifiers should have exactly the same domain.

Example

The following declaration illustrates a parameter that has exactly the same domain as the parameter UnitTransportCostWithIndexedDomain declared above.

```
PARAMETER:
    identifier   : UnitTransportCostWithSharedDomain
    index domain : (i,j) in UnitTransportCostWithIndexedDomain.Domain ;
```

The RANGE attribute

With the RANGE attribute you can restrict the values to certain intervals or sets. 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. $\{a \;..\; b\}$ covering all integers from a until and including b,
- a set identifier, 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:
    identifier   : UnitTransportCost
    index domain : (i,j)
    range        : [ UnitLoadingCost(i), 100 ] ;
PARAMETER:
    identifier   : DefaultUnitsShipped
    index domain : (i,j)
    range        : { MinShipment(i) .. MaxShipment(j) } ;
```

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 `DefaultUnitsShipped(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:
    identifier   : MaxTransportFrom
    index domain : 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 13.6.

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 23.

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.

- 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 8.4.

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.

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, or a string 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.

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.

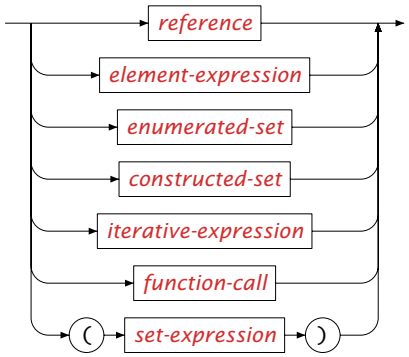
Constant set expressions

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 21.2.

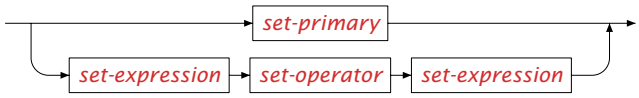
Symbolic set expressions

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*

Through the `.Domain` suffix you have access to the current contents of the index domain of any indexed identifier. You can use the `.Domain` suffix as if it were a set identifier. *The `.Domain` suffix*

Consider the indexed variable `Transport(i,j)` defined over (a subset of) the set `Cities` × `Cities`. You can use the reference *Example*

```
Transport.Domain
```

as a two-dimensional subset of `Cities` × `Cities`, containing all tuples that currently lie within the `INDEX DOMAIN` of the variable `Transport(i,j)`.

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 compound set. If you use an *integer element range*, an integer set will result. *Enumerated sets*

Constant 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.

Example

The following simple and compound set assignments illustrate constant enumerated set expressions.

```
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 )  } ;
```

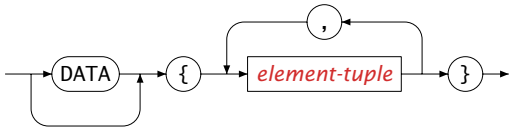
The following two set assignments contrast the semantics between constant and symbolic enumerated sets.

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

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

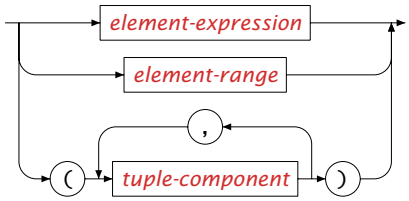
The syntax of enumerated set expressions is as follows.

Syntax

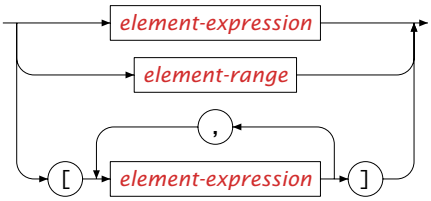
enumerated-set :



element-tuple :



tuple-component :



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

Element range

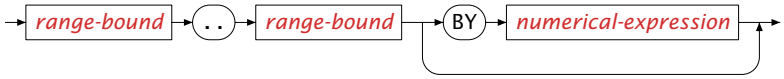
By using the `..` operator, you can specify an *element range*. An element range is a sequence of consecutively numbered elements. The following set assignments illustrate both constant and symbolic element ranges. Their difference is explained below.

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

FirstNode     := 1;
LastNode      := 100;

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

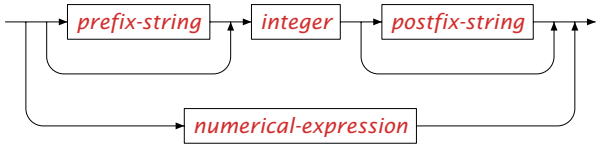
The syntax of element ranges is as follows.

Syntax

element-range :



range-bound :



Prefix and postfix strings

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.

Constant range

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 versus ElementRange

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.

```
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. An example of a dynamic integer range follows.

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

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`.

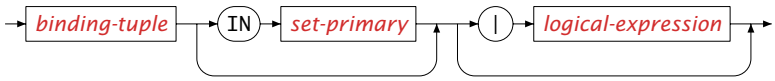
```
LargeCities := { i | Population(i) > 500000 } ;

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`.

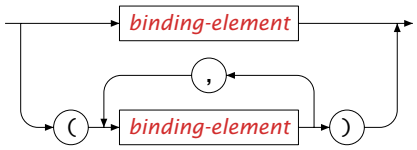
Syntax

constructed-set :

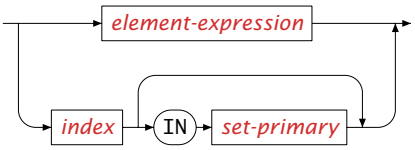
binding-domain :



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

There are four 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. *Four set operators*

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

Table 5.1: Set operators

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

```
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)}.

U := {(1,2),(1,3)} cross {4,5} ; ! The cross of a compound and integer set:
                                 ! {(1,2,4),(1,2,5),(1,3,4),(1,3,5)}.
```

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 or compound *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 ) ;
```

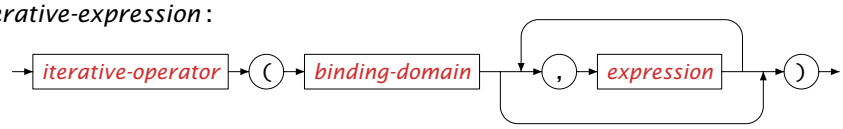
5.1.5 Iterative set operators

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 `ORDER BY` 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 `ORDER BY` 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 `ORDER BY` 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 13.7.1 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.

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)`. *Example*

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

With the `Intersection` and `Union` operators you can perform repeated set intersection or union within a particular class of indexed sets. You cannot use these operators outside of the context of indexed sets. *Repeated intersection and union*

The following assignments illustrate the use of the `Intersection` and `Union` iterative operators. *Example*

```
DestinationsForAtLeastOneCity := Union(i, DestinationCities(i));
SourcesForAllCities           := Intersection(j, SourceCities(j));
```

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

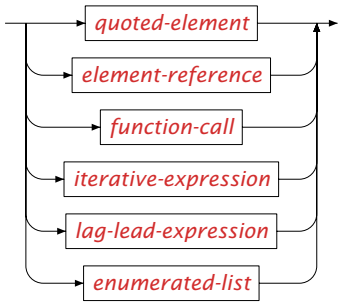
The most elementary form of a set element expression is an element parameter, which turns out to be a useful device for communicating set element information with the graphical interface. You can instruct AIMMS to locate the position in a table or other object where an end-user made changes to a numerical 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.

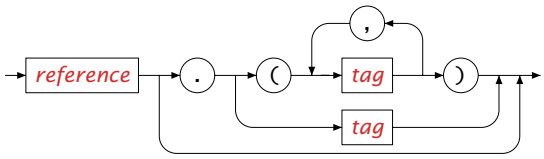
5.2.1 Element references

An element reference is any reference to either an element parameter or a (bound) index. An element reference can be followed by a tag or tuple of tags as a way to refer to a component or subtuple of components.

Element references

element-reference :

Syntax



You can use tags when you want to refer to a subtuple of an element of a compound set. The tags must be declared as attributes in the corresponding compound root set. Tags and their use are explained in detail in Section 3.2.3.

Use of tags

Assume that `orig` and `dest` (in this order!) are tags in the compound set `Routes`, and that the element parameter `MainRoute` contains a two-tuple from the compound set `Cities`×`Cities`. The following statement computes the reverse route using a tagged element reference.

Example

```
ReverseRoute := MainRoute.(dest,orig) ;
```

5.2.2 Intrinsic functions for sets and set elements

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, and the element in a non-compatible set with the identical string representation. 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` and `ElementCast` functions are given in Table 5.2.

The element-related functions . . .

The `Ord`, `Card` and `Element` functions can be applied to both simple and compound 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.

. . . for simple and compound sets

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.

Table 5.2: The `Ord`, `Card`, `Element` and `ElementCast` functions

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 Tuple function

The Tuple function has a variable number of input arguments, all of which must be element expressions. The value of this function is a literal element of a compound set. The dimension of the created tuple should match the dimension of the context in which it is used, and AIMMS must be able to determine the compound set from which the presented tuple is supposed to be an element from the context. When the result of the Tuple function is assigned to an element parameter, no assignment is made when the result is not an element of the range set of the parameter.

Example

The following assignment creates a reference to a particular route, namely between the cities represented by the element parameters HarborCity and AirportCity.

```
MainRoute := Tuple( AirportCity, HarborCity );
```

You can also assign a tuple with literal elements.

```
MainRoute := Tuple( 'Amsterdam', 'Rotterdam' );
```

In both cases the created tuple should be an element of the range set Routes of the parameter MainRoute.

5.2.3 Element-valued iterative expressions

Selecting elements

AIMMS offers special iterative operators that let you select a specific element from a simple or compound 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)
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`, `ArgMin`, and `ArgMax` operator can only consist of a single index in either a simple or compound 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.

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.

5.2.4 Lag and lead operators

Lag and lead operators

The 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.

lag-lead-expression : *Syntax*

→ *element-expression* → *lag-lead-operator* → *numerical-expression* →

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. *Definition*

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

You cannot 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 it is impossible for AIMMS to determine which set to work with. *Not for literal elements*

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. *Verify the effect of lags and leads*

Consider the set Cities initialized through the assignment *Example*

```
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.

5.3 String expressions

String expressions are useful for *String expressions*

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

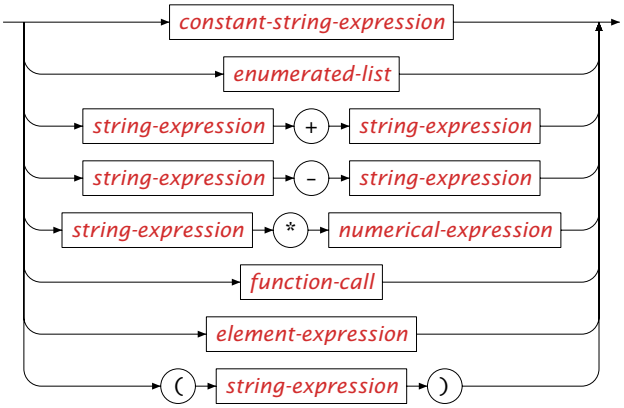
Lag/lead expression	Result
i+1	'The Hague'
i+6	''
i++6	'Amsterdam'
i++7	'Rotterdam'
i-2	''
i--2	'Madrid'
CurrentCity+2	'London'
'Rotterdam' + 1	ERROR

Table 5.5: Example of lag and lead operators

- 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 concatenation, subtraction and repetition

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 concatenation*

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.

Examples

The following examples illustrate some basic string manipulations in AIMMS.

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

5.3.2 Formatting strings

The function `FormatString`

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

- 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 format string

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.

Conversion specifiers	Argument type
`%s`	String expression
`%e`	Element expression
`%i`	Integer expression
`%n`	Numerical expression
`%u`	Unit expression
`%%`	% sign

Table 5.6: Conversion codes for the `FormatString` function

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 number %n", 4*ArcTan(1))                 ! "The number 3.14156"
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"
```

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)
- *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.

Example

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

```
FormatString("The number %>+08i", 10)              ! "The number +0000010"
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 "
```

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

Special characters

Special character	ASCII code	Meaning
\f	FF	Form feed
\t	HT	Horizontal tab
\n	LF	Newline character
\"	"	Double quote
\\	\	Backslash
\n	n	ASCII character n ($001 \leq n \leq 255$)

Table 5.7: Special characters

Examples of the use of special characters within FormatString follow.

Example

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

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.

Case conversion functions

```
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

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

Other string related functions

- 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
```

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:

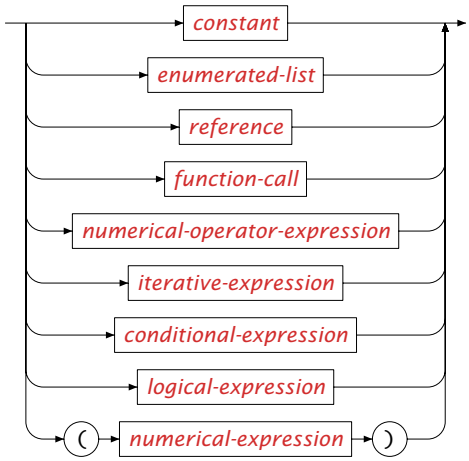
- a *scalar* value,
- a *list* expression,
- a *table* expression, or
- a *composite table*.

Table expressions and composite tables are mostly used for data initialization from *external* files. They are discussed in Chapter 21.

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 symbols NA and UNDF

The symbol NA can be used for missing data. Any operation that uses NA and does not use the symbol UNDF will also produce the result NA. 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.

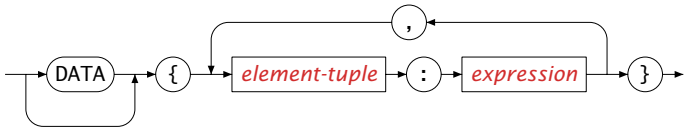
6.1.2 List expressions

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.

Syntax

enumerated-list :



Constant versus symbolic

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.

Example

The following assignments illustrate the use of list expressions.

- 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'
} ;
```

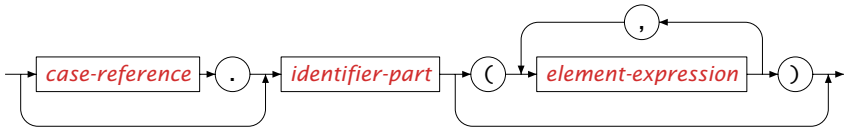
6.1.3 References

References

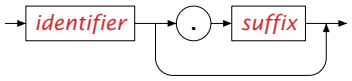
Sets, parameters and variables can be referred to by name resulting in a set-, set element-, string-valued, or numerical quantity. A reference can be scalar or multidimensional, and index positions may contain either indices or element expressions. By specifying a case reference in front, a reference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

Syntax

reference :



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 );
```

References with lag and lead

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 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

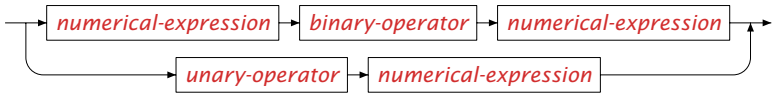
Special caution is required when one or more of the arguments in the functions are special symbols of AIMMS' extended range arithmetic. If the value of any of the arguments is `UNDF` or `NA`, then the result will also be `UNDF` or `NA`. If the value of any of the arguments is `ZERO` and the numerical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

numerical-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.

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.

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`$(x_1,\ldots,x_n)$	$\max(x_1,\ldots,x_n) \quad (n > 1)$
`Min`$(x_1,\ldots,x_n)$	$\min(x_1,\ldots,x_n) \quad (n > 1)$
`Mod`(x_1,x_2)	$x_1 \bmod x_2$
`Sign`(x)	$\mathrm{sign}(x) = +1$ 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)	$\mathrm{arccosh}(x)$
`ArcSinh`(x)	$\mathrm{arcsinh}(x)$
`ArcTanh`(x)	$\mathrm{arctanh}(x)$
`Card`(I)	cardinality 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

Exponential operator

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

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

hand side operand (called the *exponent*):

- a positive base with a real exponent,
- a negative base with an integer exponent, and
- a zero base with a positive exponent.

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.

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

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.

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

Each distribution in Table 6.5 can be used as an argument for the two functions `CumulativeDistribution` and `InverseCumulativeDistribution`. The syntax is as follows:

- `CumulativeDistribution(`*distribution*,x`)`, for $x \in (-\infty, \infty)$ computes the probability $P(X \leq x)$ where the stochastic variable X is distributed according to the given *distribution*.
- `InverseCumulativeDistribution(`*distribution*,α`)`, for $\alpha \in [0, 1]$ computes the largest $x \in (-\infty, \infty)$ such that the probability $P(X \leq x) \leq \alpha$ where the stochastic variable X is distributed according to the given *distribution*.

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);
```

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`(a,b)	Uniform distribution with lower and upper bound (a,b)
`Normal`(μ,σ)	Normal distribution with mean μ and standard deviation σ
`LogNormal`(μ,σ)	Lognormal distribution with mean μ and standard deviation σ
`Triangular`(a,b,c)	Triangular distribution with lower bound a, likeliest b and upper bound c
`Exponential`(λ)	Exponential distribution with rate λ
`Weibull`(l,β,s)	Weibull distribution with location l, shape β and scale s
`Beta`(α,β,s)	Beta distribution(α,β) with scale s
`Gamma`(α,β)	Gamma distribution with location α, and shape β
`Logistic`(μ,s)	Logistic distribution with mean μ and scale s
`Pareto`(l,β)	Pareto distribution with location l and shape β
`ExtremeValue`(m,s)	Extreme Value distribution with mode m and scale s

Table 6.5: Distributions available in AIMMS

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

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

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

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

Combinatoric functions

In addition to the distributions listed in Table 6.5, AIMMS also offers support for the most common combinatoric calculations. Table 6.6 contains the list of combinatoric functions that are available in AIMMS.

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

Table 6.6: Combinatoric functions

Statistical iterative 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 6.7. The second column in this table refers to the required number of expression arguments following the binding domain argument. You can find a more elaborate description of these statistical operators in Appendix A.3.

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

Table 6.7: Statistical iterative operators

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 iterative 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));
```

Histogram support

Besides the standard statistical iterative operators discussed above, 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 the AIMMS User's Guide.

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.
- 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 in the AIMMS Function Reference, which is only available as an online document. The Function Reference provides a description as well as the prototype of every function present in AIMMS.

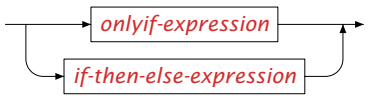
6.1.9 Conditional expressions

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

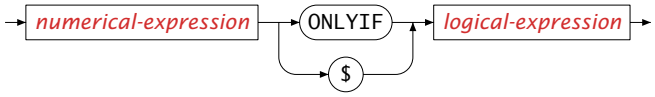
conditional-expression :



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 argument 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.

Example

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

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

or equivalently, using the $ operator,

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

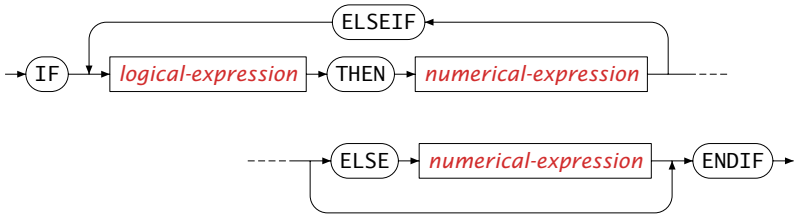
Both expressions evaluate to Distance / TravelTime if TravelTime assumes a nonzero value, or to zero otherwise. In Section 12.3 you will see that this particular expression can be written even more concisely using the sparsity modifier "$".

IF-THEN-ELSE expressions

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.

Syntax

if-then-else-expression :



Explanation

The IF-THEN-ELSE expression works like a *switch statement*—a series of ELSEIFs 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.

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

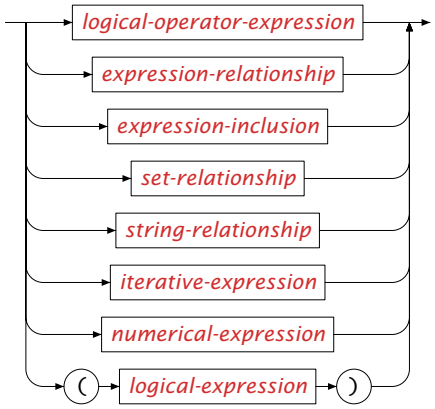
```
WeightedDistance(i) :=
    IF     Distance(i) <= 100 THEN Distance(i)
    ELSEIF Distance(i) <= 200 THEN (100 + Distance(i)) / 2
    ELSEIF Distance(i) <= 300 THEN (250 + Distance(i)) / 3
    ELSE   550 / 3
    ENDIF ;
```

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. *Numerical expressions as logical*

Example

Table 6.8 illustrates the different interpretation of a number of numerical and logical expressions as either a numerical or a logical expression. See also Table 6.9 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

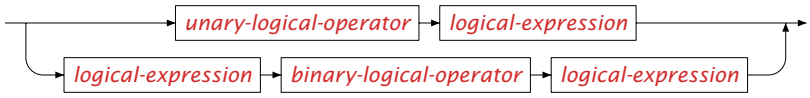
Table 6.8: Numerical and logical values

6.2.1 Logical operator expressions

Syntax

AIMMS lets you use the familiar unary and binary logical operators to construct complex logical expressions from simpler ones. The syntax is straightforward.

logical-operator-expression :



Unary and binary logical operators

AIMMS supports the unary logical operator `NOT` and the binary logical operators `AND`, `OR`, and `XOR`. Table 6.9 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 6.9: Logical operators

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.

Precedence order

The expression

Example

```
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).
```

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.

Execution order

6.2.2 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.

Numerical comparison

expression-relationship :

Syntax

expression-inclusion :

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

Numerical 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

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

AIMMS expression	Evaluates as
$x=y$	$\lvert x-y\rvert \le \varepsilon_{x,y}$
$x<>y$	$\lvert x-y\rvert > \varepsilon_{x,y}$
$x<=y$	$x-y \le \varepsilon_{x,y}$
$x<y$	$x-y < -\varepsilon_{x,y}$

Table 6.10: Interpretation of numerical tolerances

then the relational operators evaluate as shown in the Table 6.10.

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.

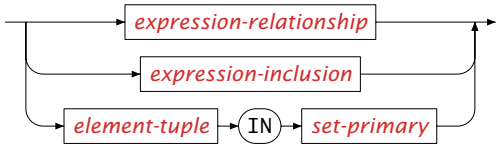
6.2.3 Set and element comparison

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`, 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.

$$\mathtt{i} \pm m \gtrless \mathtt{j} \pm n \quad \Leftrightarrow \quad \mathtt{Ord}(\mathtt{i},\mathtt{S}) \pm m \gtrless \mathtt{Ord}(\mathtt{j},\mathtt{S}) \pm n$$

This type of relational expression evaluates to "false" if one or both of the operands do not refer to existing set elements, with the exception that two nonexisting elements are considered to be equal.

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 ⊊, ⊆, ⊋ and ⊇, which are not part of the ASCII character set.

Example

The following statement illustrates a logical set comparison operator.

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

6.2.4 String comparison

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 insensitive, i.e. strings that only differ in case are considered to be equal.

Examples

All the following string comparisons evaluate to true.

```
"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

Logical iterative operators verify whether some or all elements in a domain satisfy a certain logical condition. Table 6.11 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.

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 6.11: Logical iterative operators

The following statements illustrate the use of some of the logical iterative operators listed in Table 6.11. *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

In the previous sections we have introduced unary and binary operators for several types of expressions, together with their relative precedence order. Table 6.12 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. *Combined precedence order*

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 6.12: Operator precedence (highest to lowest)

6.4 MACRO declaration and attributes

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. *MACRO facility*

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 6.13.

Attribute	Value-type	See also page
TEXT	*string*	19
ARGUMENTS	*argument-list*	
COMMENT	*comment string*	19
DEFINITION	*expression*	33

Table 6.13: MACRO attributes

The DEFINITION attribute

The DEFINITION 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.

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:
    name       : 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:
    name       : 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

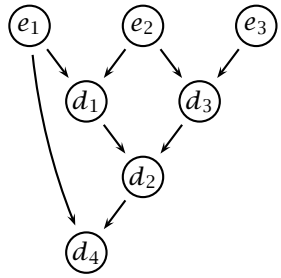
The definitions inside the declarations of global sets and parameters together form a system of interrelated functional relationships. AIMMS automatically determines the dependency between the defined identifiers and the inputs that are used inside these relationships. Such dependencies can be depicted 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

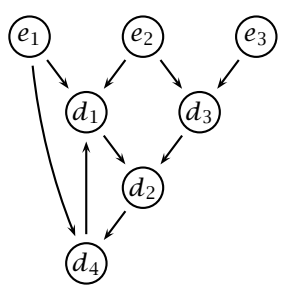
The dependency graph associated with the set and parameter definitions must be a-cyclic, i.e. must not contain circular references. In this case, every change to one or more input parameters of defined sets or parameters will result in a *finite* sequence of assignments to update the system. If the dependency 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 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.

AIMMS will check

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.

Variables for simultaneous systems

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.

Feedback loops

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 global only

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.

Dependency is symbolic

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 13.6.

Inefficiency may occur

7.2 Expressions and statements allowed in 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-

Complicated definitions

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:
    identifier   : Stock
    index domain : 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:
      identifier : 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:
      identifier : 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:
    identifier   : FixedCost
    index domain : (m,p)
    definition   : ComputeCosts ;
PARAMETER:
    identifier   : VariableCost
    index domain : (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:

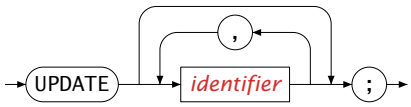
- you can call the `UPDATE` statement from within the language, or
- you can attach update requests of particular identifiers as actions to buttons and pages in the end-user interface.

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 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.

This chapter

8.1 Procedural and nonprocedural execution

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 definitions

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.

Execution based on 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.

Relating definitions and procedures

Definitions and algorithms

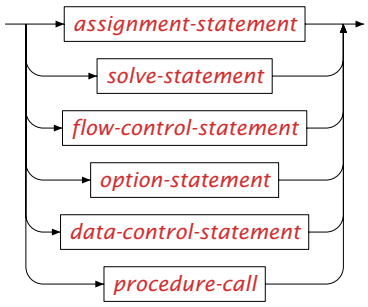
Procedural and nonprocedural execution both have their own natural role in an AIMMS application. Identifier definitions are the most convenient way to define unique functional relationships between various identifiers in your model—and keep them up-to-date at all times. Procedures provide a powerful 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

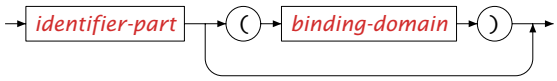
Assignment

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
```

form a shorthand notation for the assignments

```
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.

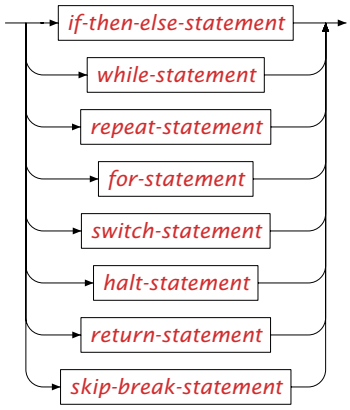
8.3 Flow control statements

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, and
- the `SKIP` and `BREAK` statements for terminating the current repetitive execution.

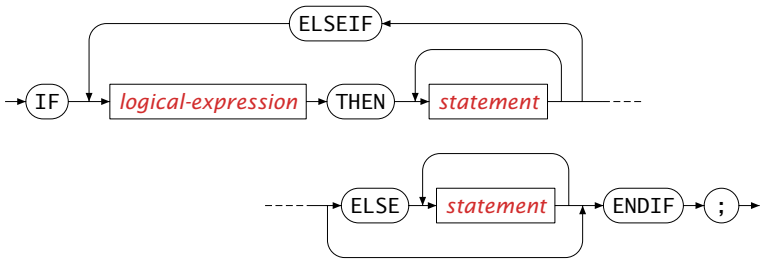
flow-control-statement : *Syntax*



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*

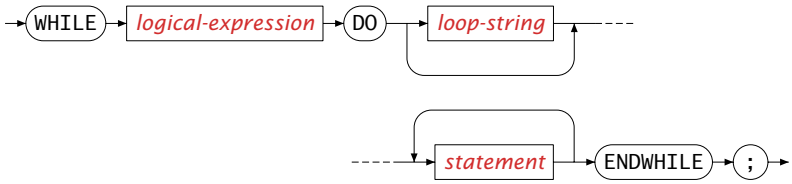
```
if ( not SupplyDepot ) then
    DialogMessage( "Select a supply depot before solving the model" );
elseif ( Exists[ p, Supply(SupplyDepot,p) < Sum( i, Demand(i,p) ) ] ) then
    DialogMessage( "The selected supply depot has insufficient capacity" );
else
    solve TransportModel ;
endif ;
```

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.

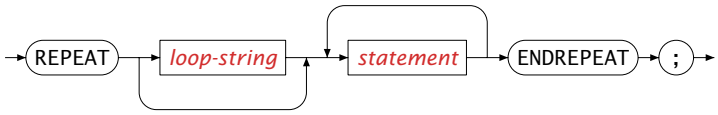
8.3.2 The WHILE and REPEAT statements

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

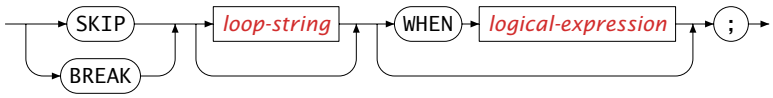
An alternative way to terminate a WHILE or REPEAT statement is the use of a BREAK statement inside the loop. BREAK statements make it possible to abort the execution at any position inside the loop. This freedom allows you to formulate more natural termination conditions than would otherwise be possible 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.

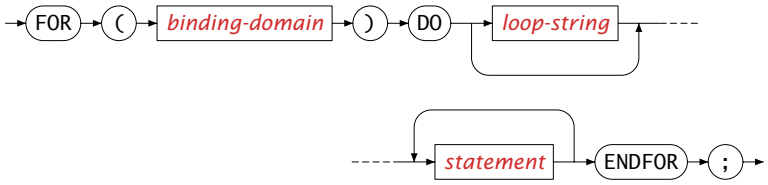
```
repeat "OuterLoop"
     ... ! Determine initial settings for sequential solution process

    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. 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 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.

AIMMS issues a warning

The following example shows an unnecessary use of the FOR statement.

Example

```
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 );
```

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.

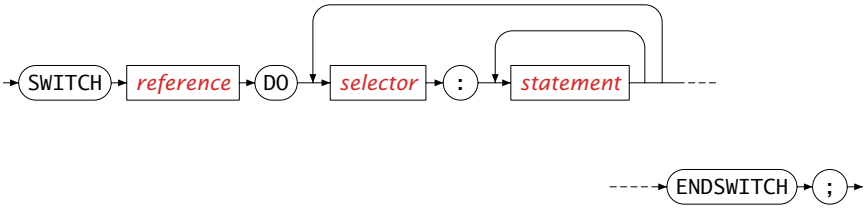
Use of SKIP and BREAK

8.3.5 The SWITCH statement

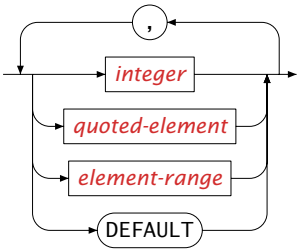
The SWITCH statement is used to choose between the execution of different groups of statements depending on the value of a scalar parameter reference. The syntax of the SWITCH statement is given in the following two diagrams.

The SWITCH statement

switch-statement : *Syntax*



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.

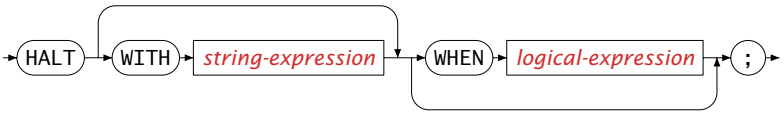
Compare to RETURN

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.

Syntax

The syntax of the HALT statement follows.

halt-statement :



Printing a message

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.

The WHEN clause

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.

Example

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.

```
solve LinearOptimizationModel;

halt with "Execution aborted: model not solved to optimality"
     when OptimizationModel.ProgramStatus <> 'Optimal' ;
```

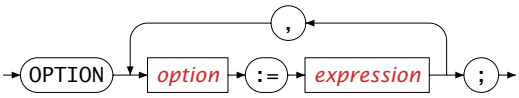
8.4 The OPTION and PROPERTY statements

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.

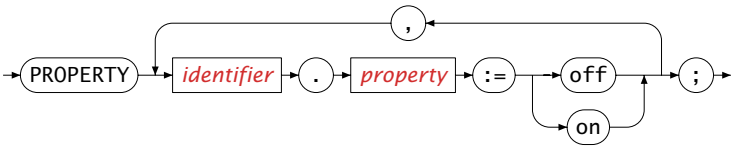
```
option  Bound_Tolerance   := 1.0e-6,
        Iteration_Limit   := UserSettings('IterationLimit');
```

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 `IntermeditateTransport` 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 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.

This chapter

9.1 Binding rules

During execution, indices are used to traverse a set to repeatedly apply a specific operation on all elements of a set. These operations concern

Repetitive operations

- indexed assignment statements,
- FOR statements,
- iterative operations like summation over a domain,
- constraint generation,
- arc generation, and
- constructed set expression.

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.

Index binding

There are three ways in which index binding takes place:

Different types of binding

- *local* binding,
- *default* binding, and
- *context* binding.

Local binding takes place through the use of an IN modifier at the index binding position as illustrated in the following example.

Local binding

```
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

```
ELEMENT PARAMETER:
   identifier   : NextCity
   index domain : i
   range        : Cities  ;
ELEMENT PARAMETER:
   identifier   : PreviousCity
   index domain : 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 5.2.2), 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 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.

Functions inside constraints

An AIMMS constraint term cannot consist of multiple flow-control statements like `FOR` and `WHILE`. However, when you have a constraint which requires a complex algorithm to compute, then it can be implemented as a function, and the function can be referenced inside the AIMMS constraint.

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*	99
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 and compound sets, 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 INDEX DOMAIN 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
    identifier : 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.

```
ELEMENT PARAMETER:
    identifier   : City
    range        : Cities
    property     : Input ;
PARAMETER:
    identifier   : DistanceMatrix
    index domain : (i,j)
    property     : Input ;
PARAMETER:
```

```
identifier   : Distance
index domain : 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.

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

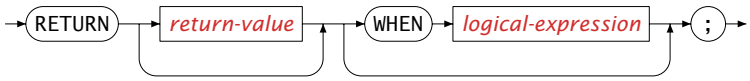
By partitioning the body of a long procedure into several execution subnodes, you can effectively implement the procedure in a self-documenting top-down approach. While the body can just contain the outermost structure of the procedure's execution, the implementation details can be hidden behind subnode 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, be-

cause 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.
- 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:

- In addition to sets, set elements and parameters, arguments of functions can also be variables.
- Functions return a result that can be used in numerical expressions. The result can be either scalar-valued or indexed.
- Functions cannot have side effects either on global identifiers or on their arguments, i.e. every function argument is of type `Input` by definition.
- Functions can be used in constraints.
- For use in constraints, you can specify the partial derivatives for all variable arguments, or AIMMS can estimate them by a simple differencing scheme.

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
    identifier : 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:
   identifier   : InputFactors
   index        : f ;
PARAMETER:
   identifier   : a
   index domain : f ;
VARIABLE:
   identifier   : c
   index domain : f ;
```

Function attributes

The attributes of functions are listed in Table 10.2. Most of them are the same as those of procedures.

Attribute	Value-type	See also page
ARGUMENTS	*argument-list*	
INDEX DOMAIN	*index-domain*	41
RANGE	*range*	42
PROPERTY	RetainsValue	
BODY	*statements*	99
DERIVATIVE	*statements*	
COMMENT	*comment string*	

Table 10.2: FUNCTION attributes

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. Note that the RETURN statement cannot have a return value in the context of a function body.

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
    identifier   : ShortestDistance
    arguments    : (City, DistanceMatrix)
    index domain : 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;
```

Variable arguments

AIMMS allows you to use 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 subsection.

10.2.1 Derivative computation

Functions in constraints

Whenever you use 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 computes the derivatives based on the contents of the body, or
- AIMMS estimates the derivatives using a simple differencing scheme.

Automatic derivative computation

When the body of a function consists of a *single* AIMMS assignment to compute the function value, AIMMS is able to generate the instructions for the computation of all partial derivatives automatically from this assignment as well. Thus, for such simple functions there is no need for you to provide any further information.

Example: the Cobb-Douglas function

Consider the Cobb-Douglas function discussed in the previous section. Because its body consists of a single assignment, AIMMS is able to compute its partial derivatives for the Cobb-Douglas function automatically. This will result in the following automatic computations:

$$\frac{\partial q}{\partial c_i} = a_i c_i^{a_i - 1} \prod_{f \neq i} c_f^{a_f}$$

The DERIVATIVE attribute

When AIMMS is not able to perform the single-assignment automatic derivative computation, and you decide to provide the partial derivatives yourself, their computation must be specified in the special DERIVATIVE attribute that is unique to functions. The following rules apply.

- If the solver only needs the function value itself, AIMMS will only execute the statements in the BODY attribute.
- If the solver needs the function value as well as its partial derivatives, AIMMS will only execute the statements in the DERIVATIVE attribute.

As a consequence, the DERIVATIVE attribute should compute the function value as well. The reason for this combined computation is that the function value and derivative can often be computed quicker in parallel than by calculating the derivative separate from calculating the function value. If there is no penalty using serial computation, then the function value computation can be shared by the BODY and DERIVATIVE attributes by creating an execution subnode containing the computation, and referencing this node in both attributes.

The .IsVariable suffix

When a function with formal variable arguments is called inside a constraint, the actual arguments need not necessarily be variables at all. Consequently, there is no need to compute the partial derivatives of these arguments in such a case. During the evaluation of the DERIVATIVE attribute of a function you can use the .IsVariable suffix of a formal variable argument to verify whether an actual argument in this call to the function is really a variable.

The .Derivative suffix

For every function argument which is a variable and for which the .IsVariable suffix is set, 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

$$\frac{\partial \mathsf{f}(i_1, \ldots, i_m)}{\partial \mathsf{x}(j_1, \ldots, j_n)}$$

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:

$$\frac{\partial q}{\partial c_i} = \frac{a_i}{c_i} CD(c_1, \ldots, c_k)$$

Thus, you could consider providing the computation of the partial derivatives for the Cobb-Douglas function (although in this particular example this will probably not lead to increased efficiency). This can be implemented in AIMMS as follows.

```
FUNCTION
    identifier : CobbDouglas
    arguments  : (a,c)
    body       : CobbDouglas := prod[f, c(f)^a(f)];
    derivative : CobbDouglas := prod[f, c(f)^a(f)];
                 if ( c.IsVariable ) then
                    c.Derivative(f) := (a(f)/c(f)) * CobbDouglas ;
                 endif;
```

Because the function value is scalar, the derivative suffix for the variable argument c has the same dimension as c itself.

Numerical differencing

When AIMMS cannot compute the derivative values automatically, and you have not specified the DERIVATIVE attribute to compute the derivatives yourself, AIMMS employs a simple differencing scheme to estimate the derivatives. For example, assume that 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 a small number.

The DIFFERENCING attribute

In the declaration of the formal arguments of a function, you can specify the value for ε that AIMMS will use for each individual variable argument. For this purpose all VARIABLE declarations inside a function have an additional DIFFERENCING attribute that is not present for variable declarations outside the context of a function. If you do not specify a value for the DIFFERENCING attribute of a particular variable, AIMMS will use the current value of the global option Differencing_Delta instead.

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.

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.

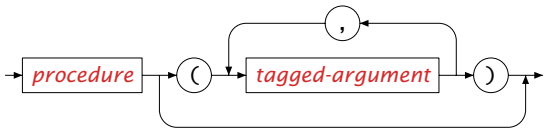
```
ShortestDistanceMatrix(i,j) := ShortestDistance(i, DistanceMatrix)(j) ;
```

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)`.

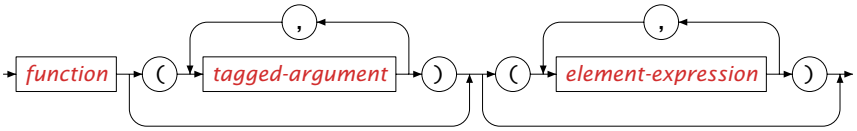
Call syntax

The general forms of procedure and function calls are identical, except that a function reference can have additional indexing.

procedure-call :



function-call :



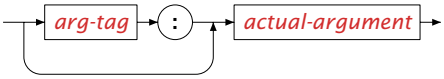
Actual arguments

Each actual argument can be

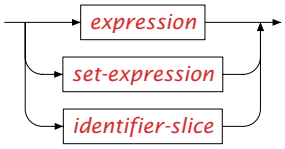
- any type of scalar expression for *scalar* arguments, and
- a reference to an identifier slice of the proper dimensions for *non-scalar* arguments.

Actual arguments can be tagged with their formal argument name used inside the declaration of the function or procedure. The syntax follows.

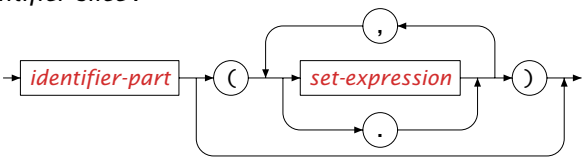
tagged-argument :



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.

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 domain 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.

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 variable `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 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.

```
ELEMENT PARAMETER:
    identifier     : EventType
    index domain   : e
    range          : EventTypes ;
ELEMENT PARAMETER:
    identifier     : EventHandler
    index domain   : et in EventTypes
    range          : AllIdentifiers
    default        : NoEventHandlerSelected
    initial data   :
          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 INITIAL DATA 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.

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

```
ELEMENT PARAMETER:
    identifier     : BlendRule
    index domain   : q
    range          : AllIdentifiers
    default        : BlendLinear
    initial data   :
          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:
    identifier     : ComputeBlendProperty
    index domain   : 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 25.1 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 `EXTERNAL PROCEDURE` and `EXTERNAL FUNCTION` 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 EXTERNAL FUNCTION CobbDouglasExternal, which just relays its execution to the C implementation of the Cobb-Douglas function discussed above. The declaration of CobbDouglasExternal looks as follows.

```
EXTERNAL FUNCTION
    identifier    : CobbDouglasExternal
    arguments     : (a,c)
    range         : nonnegative
    DLL name      : "Userfunc.dll"
    return type   : double
    body call     : 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 126.

```
SET:
   identifier   : InputFactors
   index        : f ;
PARAMETER:
   identifier   : a
   index domain : f
VARIABLE:
   identifier   : c
   index domain : 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

Like with internal functions, the declaration can be extended with a `DERIVATIVE CALL` 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 `DERIVATIVE CALL` 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
DLL NAME	*string*, *file-identifier*	
RETURN TYPE	integer, double	
PROPERTY	FortranConventions, UndoSafe	124
BODY CALL	*external-call*	
DERIVATIVE CALL	*external-call*	

Table 11.1: Additional attributes of external procedures and functions

The DLL NAME attribute

With the mandatory DLL NAME 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. If you do not specify an absolute path, AIMMS will search for the DLL in all directories in the AIMMSFNC and PATH environment variables, respectively.

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 23.6). 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. Without any such convention, AIMMS will use the default convention (i.e. as specified for every identifier in the model).

The RETURN TYPE attribute

The RETURN TYPE 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 EXTERNAL PROCEDURE, or as the (numerical) function value of the EXTERNAL FUNCTION, whichever is applicable. If you do not specify the RETURN TYPE 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 EXTERNAL FUNCTION 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 EXTERNAL FUNCTION declaration provides a link with Cobb_Douglas_Arg and obtains its function value via the argument list.

```
EXTERNAL FUNCTION
    identifier    : CobbDouglasArgument
    arguments     : (a,c)
    range         : nonnegative
    DLL name      : "Userfunc.dll"
    body call     : 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 and compound SETS,
- scalar and indexed PARAMETERS,
- scalar and indexed VARIABLES (external functions only), and
- HANDLES (external procedures only).

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 25).

The BODY CALL attribute

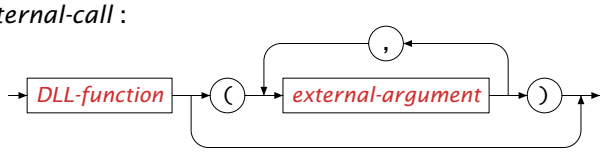
In the mandatory BODY CALL attribute you must specify the call to the DLL procedure or function, to which the execution of the EXTERNAL PROCEDURE or FUNCTION must be relayed. Such an external call specifies:

- 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.

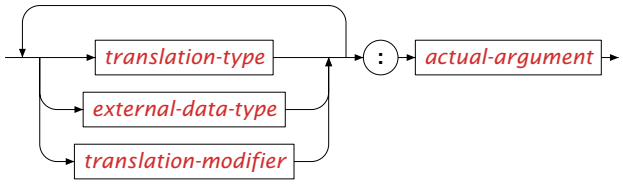
Any external call must be specified according to the syntax below. In the Model Explorer, you can specify all components of the BODY CALL attribute using a wizard which will guide you through most of the necessary detail.

Syntax

external-call :



external-argument :



Mandatory translation type

The mandatory translation type indicates the type of the external argument into which the actual AIMMS argument must be translated before being passed to the external procedure. The following translation types are supported.

- **scalar**: the scalar AIMMS argument is passed on as a scalar of the indicated external data type.
- **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.
- **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 25) 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.

Input-output type

For every argument of an `EXTERNAL PROCEDURE`, you can specify its associated *input-output* type through the `Input`, `InOut` (default) or `Output` properties in the `PROPERTY` 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 `EXTERNAL FUNCTION` arguments are `Input` by definition. The return value of an `EXTERNAL PROCEDURE` and the function value of an `EXTERNAL FUNCTION` 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
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.

Output string arguments

When you pass a scalar or multidimensional output string argument, AIMMS will pass a single char buffer of length 256, or an array of such buffers. 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 25.4 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 25.4 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 18.3). 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 25.4.

... 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 compound), and multidimensional 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, compound 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 *compound subset* (i.e. a multidimensional relation) is the dimension of the Cartesian product of which the set is a subset,
- the dimension for a *subset of a compound set* is 1,
- 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 25.2) 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

Derivative computation

Whenever an internal function is called with variable arguments in a constraint of a mathematical program, the function declaration should also indicate how the partial derivatives with respect to these variable arguments are computed. As discussed in Section 10.2.1 these partial derivatives can result from either a simple and automatic numerical differencing scheme based on multiple evaluations of the function, or alternatively, from your own algorithm.

... for external functions

When the entire function body consists of a single call to a DLL function written by you, it is not unlikely that you can also provide an external procedure to compute the partial derivatives (or estimates thereof). Such a procedure can be made more efficient and accurate than the automatic differencing scheme offered by AIMMS.

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 25).

Example: the Cobb-Douglas function

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 (see also Section 10.2.1).

... 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. Instead, for FORTRAN procedures, you can pass the `.IsVariable` suffices (see also Section 10.2.1) of all variable arguments as additional external arguments, and use those to decide on the computation of particular partial derivatives. AIMMS will implicitly consider such arguments as `input` arguments.

The DERIVATIVE CALL attribute

In the `DERIVATIVE CALL` 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 `DERIVATIVE CALL` attribute is the same as that of the `BODY CALL`, and is most conveniently completed using the wizard in the Model Explorer.

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.

```
EXTERNAL FUNCTION
    identifier      : CobbDouglasPlusDerivative
    arguments       : (a,c)
    range           : nonnegative
    DLL name        : "Userfunc.dll"
    return value    : double
    body call       : Cobb_Douglas_No_Der( card : InputFactors, array: a, array: c )
    derivative call : Cobb_Douglas_Der( card : InputFactors, array: a,
                                        array: c, array: c.Derivative )
```

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.

```
EXTERNAL PROCEDURE
    identifier      : ExternalAverage
    arguments       : (x,res)
    DLL name        : "Userfunc.dll"
    body call       : ComputeAverage(double array: x, card: i, card: j, double scalar: res)
```

where the argument `x` and `res` are declared as

```
PARAMETER:
    identifier      : x
    index domain    : (i,j)
    property        : Input ;
PARAMETER:
    identifier      : 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

Semantics of Sparse Execution

This chapter

The aim of this chapter is to make you aware of the semantics associated with sparse execution. Careful analysis of index domains plays an important role in the correct specification of statements and expressions. AIMMS provides two sparsity modifiers `$` and `&` which you can use to restrict index domains in a compact manner, and helps to increase the overall readability of your model.

12.1 Introduction

Sparse data storage

The multidimensional data tables in most applications contain data for just a subset of all possible tuples. For instance, a parameter `Production(p,f)` has many empty values when each factory `f` produces only a small subset of products `p`. For large data structures the ratio of empty values over all possible values may very well be more than 99%. For this reason, AIMMS only stores the nonempty data values, allowing it to handle large-scale applications.

Sparse data computation

In modeling applications, computations with sparse multidimensional data tables can often be performed using only the nonempty data values. For instance, assume that the parameter price is available for every product `p`, and that the expression `Price(p) * Production(p,f)` needs to be computed for every product `p` and factory `f`. Then the computation can be implemented such that the multiplication of a nonzero `Price` with a zero `Production` never takes place. Such efficiency considerations are the foundation of the sparse execution engine of AIMMS.

Non-sparse notation

Even though AIMMS has a sparse storage scheme and a sparse execution engine, its modeling language offers a multidimensional notation which references all possible tuples. This allows you to express functional relationships between sparse identifiers without explicit reference to sparse storage considerations. This is an important AIMMS characteristic which allows you to express large-scale models in a compact and transparent notation.

Be aware of semantics

A direct reference to a multidimensional structure does not explicitly reflect its sparsity considerations, and consequently you need to understand the semantics of the underlying sparsity properties to make sure that the result is as intended. Consider, for instance, an assignment statement which determines for each product p the least amount of combined stock and production available in any factory f. Assume that you had written it as follows.

```
LeastSupply(p) := Min[ f, Stock(p,f) + Production(p,f)];
```

This evaluation of the statement is clear when Stock and Production have both nonempty positive data values for all tuples (p,f), but what happens when this is not the case? What does it mean to take a minimum over empty values? In AIMMS there is a basic rule which states that all references to empty values evaluate to zero. This implies that the value of LeastSupply(p) in this example will typically be zero, because not every factory f produces product p. But is this what you as a modeler intended to compute? It is more likely that you are interested in the LeastSupply of only those factories that actually manufactured product p. If so, then you need to rewrite the above assignment statement.

No standard semantics

Different modeling languages have different semantic rules to govern the execution of statements involving parameters with references outside their index domain. There are advantages and disadvantages associated with each choice, and consequently no single approach is preferred. Independent of the choice, you as a modeler must always take care when specifying index domain restrictions in expressions and statements. The design approach implemented in AIMMS is to keep the rules to a minimum, and to offer special sparsity modifiers which can (but need not) be used to obtain compact and transparent model descriptions.

12.2 Sparse execution and index domains

One basic rule

During the execution of expressions and assignment statements in AIMMS, all identifier references are subject to the following important rule.

> *A numerical indexed parameter always evaluates to the number 0 when referenced outside its index domain.*

It has the advantage that whenever AIMMS does not find a compile-time error, the resulting execution can essentially be performed without run-time errors. Most mathematical operations are well-defined for the number zero, except for a few isolated instances such as dividing by zero.

Effect of zero on iterative operators

The effect of the number zero within the scope of iterative operators other than the `Sum` operator may lead to unexpected results if the associated index domain of the operator is not sufficiently restricted. The `Min` operator used in the previous section represents a typical example. Carefully constructing domain restrictions will in general be sufficient to obtain the desired meaning.

Example

Consider the example from the introduction above, together with the following three assignment statements. Each statement determines the parameter `LeastSupply`, but with distinctly different semantics.

```
LeastSupply(p) := Min[f, Stock(p,f) + Production(p,f)];
```

In this statement the `Min` operator considers all factories `f`, and evaluates the corresponding expression `Stock(p,f) + Production(p,f)`. The expression of `LeastSupply(p)` evaluates to zero for every product `p` that is *not* manufactured in *every* factory `f`. The domain restriction in the next statement avoids this situation.

```
LeastSupply(p) := Min[f | Stock(p,f) + Production (p,f), Stock(p,f) + Production(p,f)];
```

In this statement the `Min` operator considers only those factories `f` where the combined stock and production is nonzero. This implies that the value of `LeastSupply(p)` will be positive, unless it is not produced in any factory. In the latter case the value of `LeastSupply(p)` is infinity (the initial value of the `Min` operator). The domain of the following statement allows a zero value of `LeastSupply(p)`.

```
LeastSupply(p) := Min[f | (p,f) in Production.Domain, Stock(p,f) + Production(p,f)];
```

In this statement the `Min` operator considers only those factories `f` for which `Production(p,f)` is allowed (thus within the index domain of the parameter `Production(p,f)`). This implies that the value of `LeastSupply(p)` could be zero when there happens to be no combined stock and production available at a particular factory `f` while there could have been.

Effect of zero on relational operators

The effect of the number zero within the scope of relational operators other than <> can also lead to unexpected results, because they return the value true when both operands reference values outside their index domain. Again, careful specification of the corresponding index domains will lead to the desired outcome.

Example

Consider the following assignment statement.

```
ShortLink((i,j) | Distance(i,j) <= ShortLinkThreshold(i,j)) := 1 ;
```

This assignment is executed for those tuples `(i,j)` that are in the intersection of the index domain of the identifier `ShortLink` and the set of tuples for which the above domain restriction is true. What happens if both `Distance` and `ShortLinkThreshold` are referenced outside their domain for a particular tuple `(i,j)`?

The condition evaluates then to `0 <= 0` which is true. Thus, the assignment is made. But is this what you intended? If the answer is no, then the following modified assignment may amend the situation by further restricting the tuples to be considered.

```
ShortLink((i,j) | Distance(i,j) and ShortLinkThreshold(i,j)
                  and (Distance(i,j) <= ShortLinkThreshold(i,j)) ) := 1 ;
```

In this statement a tuple `(i,j)` is only considered when both parameters `Distance` and `ShortLinkThreshold` have nonzero values. As a result, the logical condition involving the distance parameters will never evaluate to `0 <= 0`, and the contents of `ShortLink` is adjusted accordingly.

12.3 Sparsity modifiers of binary operators

Need for compact notation

As discussed in the previous section, sparsity considerations are important in most large-scale applications. As a result, in many applications you will find yourself specifying extra index domain restrictions in order to obtain the correct computation of identifiers. These restrictions take up space, and have a negative effect on both the readability and understandability of the underlying model statements.

Two sparsity modifiers

To provide a compact notation for the most common domain restrictions, AIMMS offers two special operators, namely the `$` and `&` symbol, to use inside expressions. With these symbols you essentially modify the way in which a particular operator handles references to sparse identifiers. For this reason, these two symbols are referred to as *sparsity modifiers*.

The $ operator

The `$` operator can be placed on either or both sides of a binary operator, and is an instruction to AIMMS to consider only the nonzero values (i.e. unequal to 0.0) of the corresponding operand(s).

Examples

The following examples illustrate the use of the sparsity modifier `$` by providing equivalent statements with and without the modifier.

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

AssignedTransport(i,j) $:= GainFactor(i,j) * Transport(i,j);
AssignedTransport((i,j) | AssignedTransport(i,j))
                                  := GainFactor(i,j) * Transport(i,j);

AssignedTransport(i,j) $:=$ GainFactor(i,j) * Transport(i,j);
AssignedTransport((i,j) | AssignedTransport(i,j) and
                          (GainFactor(i,j) * Transport(i,j)))
                                  := GainFactor(i,j) * Transport(i,j);

ShortLink((i,j) | Distance(i,j) $<=$ ShortLinkThreshold(i,j)) := 1 ;
ShortLink((i,j) | Distance(i,j) and ShortLinkThreshold(i,j)
                  and (Distance(i,j) <= ShortLinkThreshold(i,j)) ) := 1 ;
```

The & operator

The & operator can be placed on either or both sides of a binary operator. It is an instruction to AIMMS to consider only the intersection of all index domains of the identifiers used in the corresponding operand(s).

Examples

The following examples illustrate the use of the sparsity modifier & by providing equivalent statements with and without the modifier.

```
AssignedTransport(i,j) :=& GainFactor(i,j) * Transport(i,j);
AssignedTransport((i,j) in (GainFactor.Domain * Transport.Domain))
                                  := GainFactor(i,j) * Transport(i,j);

ShortLink((i,j) | Distance(i,j) &<=& ShortLinkThreshold(i,j)) := 1 ;
ShortLink((i,j) in (Distance.Domain * ShortLinkThreshold.Domain) |
                     Distance(i,j) <= ShortLinkThreshold(i,j)) := 1 ;
```

Applicability

Adding sparsity modifiers on either side of a binary operator does not always make sense. For instance, adding a sparsity modifier to a multiplication is unnecessary, because the result is nonzero only when both operands are nonzero in the first place. AIMMS recognizes such situations, and only allows sparsity modifiers when appropriate. Table 12.1 summarizes the applicability of sparsity modifiers for all binary operators.

Operator	Sparsity modifier allowed			
	$ left	$ right	& left	& right
^	yes	yes	–	–
*	–	–	–	–
/	–	yes	–	–
+, -	–	–	–	–
=, <>, <, <=, >, >=	yes	yes	yes	yes
:=	yes	yes	yes	yes
+=, -=	yes	–	yes	yes
*=, /=	yes	yes	yes	yes
$, ONLYIF AND, OR, XOR	–	–	–	–

Table 12.1: Sparsity modifiers of binary operators

Sparsity and special values

The $ modifier considers only the nonzero values of the operands under consideration. If any of these operands contain one or more of the special values ZERO, NA, UNDF, INF and -INF, then these values are also considered as nonzero values, and will have an effect on the computational results. Similarly, when special values are part of index domains of identifiers, they have an effect on the computational results whenever you use the & modifier. Awareness and careful specification on your part will suffice in creating semantically correct statements and expressions.

12.4 Sparsity modifiers of iterative operators

Modifiers of iterative operators

The sparsity modifiers introduced in the previous section for binary operators, can also be applied to the iterative operators in AIMMS. The associated semantics are explained in this section.

The $ operator

The $ operator can be added to an iterative operator. In this case, AIMMS will only consider those elements of the index domain for which the argument of the iterative operator is nonzero.

The & operator

The & operator can also be added to an iterative operator. In this case, AIMMS will only consider those elements of the index domain that are in the intersection of all index domains of the identifiers inside the second argument of the iterative operator.

Example

The following examples illustrate the use of the sparsity modifiers on iterative operators by providing equivalent statements with and without the modifier.

```
SmallestNonZeroDistance := Min$[ (i,j), Distance(i,j)] ;
SmallestNonZeroDistance := Min [ (i,j) | Distance(i,j), Distance(i,j)] ;

SmallestDomainDistance := Min&[ (i,j), Distance(i,j)] ;
SmallestDomainDistance := Min [ (i,j) in Distance.Domain, Distance(i,j)] ;
```

Applicability

Like with binary operators, not all combinations of iterative operators and sparsity modifiers make sense. Table 12.2 summarizes the applicability of both sparsity modifiers for all iterative operators.

Iterative operator	**Sparsity modifier allowed**	
	$ added	& added
Sort, NBest	yes	yes
Intersection, Union	yes	yes
First, Last, Nth	–	yes
ArgMin, ArgMax	yes	yes
Sum	–	–
Prod	yes	yes
Min, Max	yes	yes
Statistical operators (see also page 79)	yes	yes
ForAll	–	yes
Other logical operators (see also page 87)	–	–

Table 12.2: Sparsity modifiers of iterative operators

Chapter 13

Execution Efficiency

This chapter

There is often more than one way to express a calculation, and due to the nature of the sparse execution engine underlying AIMMS, these variations can affect the performance of your model. In this chapter, the basics of the execution engine is explained in more detail, together with some of the more common alternative expressions and flow control statements that may help you reduce the execution times of your model.

Intended audience

For most models that do not work with very large or high-dimensional data sets, the considerations in this chapter may have but a marginal effect on the execution times of your application. Thus, this chapter is mainly aimed at those of you who build large and highly complex models and find that your model is pushing AIMMS to its limits. In such cases, a thorough understanding of the issues discussed in this chapter can be helpful to reformulate your model in such a manner that it cooperates better with AIMMS' sparse execution engine.

13.1 Introduction

Two aspects

There are two aspects to consider when writing efficient models. The first aspect is knowing the quantity of work involved in executing a statement. The second aspect is understanding how statements are executed in AIMMS, and knowing the range of possible alternatives for expressing a calculation.

Measuring execution times

With the AIMMS profiler you can directly measure the amount of CPU time that a particular procedure or individual statements take to execute. In this way you can focus on those parts of your model that take the longest to execute, and thus have the greatest opportunity for improvement.

Alternative expressions

A more demanding task is to develop your knowledge of why it takes AIMMS a long time to execute particular statements, and how to formulate alternative but equivalent expressions that execute faster. To help you with this task, it is explained how the AIMMS execution engine works in basic terms, why some operations are more efficient than others, and how to write some of the more common alternative expressions.

While you are trying to improve execution performance, do not hesitate to test various alternatives, and measure their relative performance.

Test and measure

13.2 Programming basics

A basic tool for efficient execution is the use of temporary storage for data values that are used often within a calculation. Take, for instance, the calculation of costs and revenues for a customer and a merchant.

Storage versus execution

```
CustomerTotalCost    := Sum(p, Price(p) * Quantity(p)) + OtherCosts;
MerchantTotalRevenue := Sum(p, Price(p) * Quantity(p)) + OtherRevenue;
```

If we slightly restructure the calculation, we can cut the amount of work in half by storing the value of the sum.

```
TransactionCost      := Sum(p, Price(p) * Quantity(p));
CustomerTotalCost    := TransactionCost + OtherCost;
MerchantTotalRevenue := TransactionCost + OtherRevenue;
```

This is a very simple and effective principle, which is not typical to modeling but is true for any programming language as well. A typical place to find this type of opportunity for efficiency is in loops.

Not modeling specific

13.3 Efficiency of implied loops

Sets and their use in indexing multidimensional data structures allow AIMMS to offer very powerful assignments in a single statement. AIMMS will try to make use of the sparsity of the identifiers involved in the statement to reduce its computational complexity. Most of the time, the *implied* loops thus constructed by AIMMS execute much more efficiently than the same assignment contained in an *explicit* user-defined loop. You are advised to use implied loops as much as possible.

Implied versus explicit loops

Consider the following declarations

Example

- a set `S` containing n elements with indices `i`, `j` and `k`,
- a parameter `Sparse(i,j)` containing $m \ll n^2$ nonzero values, and
- a parameter `Dense(i,j)` containing n^2 nonzero values.

Next, consider the assignment statements

```
Result(i,j) := Sparse(i,j) * Dense(i,j);     ! implied loop O(m)

for (i,j) do
    Result(i,j) := Sparse(i,j) * Dense(i,j); ! explicit loop O(n^2)
endfor;
```

The first assignment statement is called a *simultaneous assignment* using *implied loops*. By making the loops explicit, as in the second assignment statements, AIMMS is forced to execute the assignment sequentially for every tuple (i,j). The first assignment can be made with $O(m)$ operations. Clearly, the serial one requires $O(n^2)$ operations and should be avoided.

Constants over implied loops

Next to the issue of implied versus explicit loops discussed in the previous paragraphs, one also has to be wary about the use of iterative operators are constant over one or more of the indices of the (implied) loop induced by a simultaneous assignment. Such iterative operators may force AIMMS to fall back to a much less efficient computation scheme. The following example illustrates the point.

Example

Consider the following assignments.

```
Result(i,j) := Sparse(i,j) * Sum(k, Sparse(i,k));  ! O(m*n)
Result(i,j) := Sparse(i,j) + Sum(k, Sparse(i,k));  ! O(n^3)
```

The implied loops are nested over i and j, while the summation over k is constant when looping over j. This causes $O(mn)$ and $O(n^3)$ operations respectively. By pulling the summation out of the assignment and reserving temporary storage for it, a more efficient way to structure this calculation is as follows.

```
Temp(i)     := Sum(k, Sparse(i,k));               ! O(m).
Result(i,j) := Sparse(i,j) * Temp(i);             ! O(m).
Result(i,j) := Sparse(i,j) + Temp(i);             ! O(n^2).
```

By computing Temp(i) first and using it in the assignment, the calculation is sped up by at least a factor of n. A little bit of extra storage has given a great benefit in execution time. Note that sparse multiplication is more efficient than addition.

13.4 Operator efficiency

Numerical and logical operators

In this section you will find an a priori analysis of operator efficiency for various types of operators. Table 13.1 lists the efficiency of the common numerical and logical unary and binary operators as employed by AIMMS.

Efficiency indicators

The efficiency indicators used in Table 13.1 for binary operators are explained next.

- **Intersection:** only operations for which both operands are nonzero need to be executed. All others can be skipped as the result is zero anyway.
- **Union:** only operations for which one of the operands is nonzero need to be executed. If both operands are zero, the operations can be skipped as the result is zero anyway.

Operator	Efficiency
Unary	
+, -	sparse
NOT	dense
Binary	
$, *, /$, AND	intersection
+, -, OR, XOR	union
/, ^	dense

Table 13.1: Numerical and logical operator efficiency

- **Dense:** the operations are always executed, even if both operands are zero.

Example

We present a simple assignment using the *division operator* where the numerator is sparse and of higher dimension than the denominator. By carefully considering the efficiency of both multiplication (intersection) and division (dense), we can reformulate this assignment in such a way that the same computation is executed more efficiently at the cost of some extra (temporary) storage. The following statements make the point.

```
Result(i,j)  := Sparse(i,j)/Dense(i);         ! O(n^2)

DenseInv(i)  := 1/Dense(i);                   ! O(n)
Result(i,j)  := Sparse(i,j) * DenseInv(i);    ! O(m)
```

Assignment operators

In addition to the efficiency of the unary and binary numerical and logical operators discussed above, the assignment operator itself can also influence the efficiency of implied loops. The efficiency indicators of assignment operators in AIMMS are listed in Table 13.2.

Operator	Efficiency
:=, +=, -=, *=	union
/=	dense

Table 13.2: Assignment efficiency

Explanation

The efficiency of the assignment operator := can be explained as follows. An assignment should only be made for those tuples for which

- the right-hand side is nonzero, or
- the left-hand side identifier holds a nonzero value, which should be replaced by the zero outcome of the right-hand side.

No assignment is necessary when both left- and right-hand sides are zero. The efficiency of the other assignment operators can be explained in a similar fashion.

Use of sparsity modifiers

With the sparsity modifiers `$` and `&` (see also Section 12.3) you do not only modify the semantics of an assignment statement, but you also enable AIMMS to execute such comparisons much more efficiently. Namely, adding such a modifier to either side of an assignment will cause the left- and/or right-hand domain to be *intersected* with the set of tuples with nonzero values or within the domain, respectively.

Example

Assume that the identifier `Dense(i,j)` holds nonzero values for all tuples `(i,j)`, and consider the following assignments.

```
Dense(i,j) := Sparse(i,j);          ! O(n^2)

Dense(i,j) :=$ Sparse(i,j);         ! O(m), but assignment only made when
                                    ! Sparse(i,j) <> 0

empty Dense;
Dense(i,j) := Sparse(i,j);          ! O(m), assignment made for all tuples
```

Thus, by merely removing all old values before executing the assignment the execution times may be reduced significantly!

Comparison operators

A third class of operators that can influence the efficiency of implied and explicit loops, are the equality and inequality operators. The efficiency of these numerical comparison operators in AIMMS are listed in Table 13.3.

Operator	Efficiency
<, >, <>	union
<=, >=, =	dense

Table 13.3: Comparison operator efficiency

Explanation

The difference in efficiency of inclusive versus exclusive comparison operators lies in the fact that the inclusive comparison operators <=, >= and = are still true when both operands are zero, while this is not the case for the exclusive comparison operators <, > and <>. Stated differently, for an exclusive comparison to be true, at least one of the operands must be nonzero (i.e. union behavior). For inclusive comparison operators, AIMMS has to consider all tuples (i.e. dense behavior).

Avoid dense efficiency ...

You should be aware that adding an inclusive comparison to the left-hand side of an assignment or in a constraint of a mathematical program, can have a devastating effect in terms of efficiency. This is especially true when the operation must be performed over a high-dimensional domain. As stated above, an inclusive comparison is an inherently dense operation, and AIMMS is forced to execute or generate over the full domain. Without the comparison, and if the assignment or constraint does not contain any other dense operations, the operation will be performed at most over the unions of the respective nonzero domains.

... by using sparsity modifiers

When you need an inclusive numerical comparison to restrict a (high-dimensional) domain, you are advised to make use of the sparsity modifiers `$` and `&` as much as possible. They will effectively modify the efficiency indicator from dense to union or intersection. Of course, you should make sure that the semantics of the modified comparison is still as you intended.

Example

Consider the following two semantically equivalent assignments, using an inclusive comparison operator to restrict the left-hand side domain.

```
SlackDistance((i,j) | Distance(i,j)  <=  MaxDistance(j)) :=
    MaxDistance(j) - Distance(i,j);

SlackDistance((i,j) | Distance(i,j) $<=$ MaxDistance(j)) :=
    MaxDistance(j) - Distance(i,j);
```

The use of the `$` sparsity modifier in the second assignment will reduce the complexity of the assignment from $O(n^2)$ to linear in the number of nonzeros of `Distance` and `MaxDistance`.

13.5 Domain-related efficiency

Structural efficiency

In the previous sections the computational behavior of many numerical and logical operators in AIMMS has been explained. As you have seen, some operations must—without further restrictions being specified—be executed over the full domain because of semantical reasons. Therefore, the art of efficient modeling foremost consists of identifying and specifying the common domains of definition and/or computation, and using such domains wherever possible.

Specifying domain conditions

The first and most important opportunity to specify domain conditions is in the declaration of multidimensional identifiers. When such a domain condition does not consist of a simple identifier lookup, AIMMS will silently compute and store the condition as the definition of an AIMMS-generated parameter, and refer to this defined parameter during the further execution of the model.

Effect of replacement

The effect of this replacement of such compound conditions by AIMMS-generated defined parameters is that the evaluation of the condition during execution of the model boils down to a simple identifier lookup, which will always result in intersection efficiency. In addition, the condition only needs to be recomputed when the identifiers involved in the condition have been changed.

Use of domain conditions

AIMMS will use the domain of definition of a multidimensional identifier as specified by the domain condition in its declaration as follows.

- Any assignment to the identifier will be restricted to the current contents of its domain of definition.
- Any reference to the identifier will be skipped (and thus implicitly evaluate to zero) outside of its domain of definition.
- If the identifier is a variable, AIMMS will only generate columns in the solver matrix for those tuples that lie within its domain of definition.

Each of these reductions will lead to increased efficiency of your model.

Example

Consider the declaration of the parameter `RestrictedPar(i,j)`.

```
PARAMETER:
    identifier    : RestrictedPar
    index domain  : (i,j) | a(i,j) <= b(i,j) ;
```

As the domain condition does not consist of a single identifier lookup, but of a numerical comparison with dense efficiency, the AIMMS compiler will internally replace this declaration as if declared as follows.

```
PARAMETER:
    identifier    : DomainRestriction
    index domain  : (i,j)
    definition    : 1 $ (a(i,j) <= b(i,j)) ;
PARAMETER:
    identifier    : RestrictedPar
    index domain  : (i,j) | DomainRestriction(i,j) ;
```

Subsequently, AIMMS will execute the assignment

```
RestrictedPar(i,j) := 1;
```

as if written

```
RestrictedPar((i,j) | DomainRestriction(i,j)) := 1;
```

Thus, the computational complexity is completely determined by the number of nonzeros in `DomainRestriction(i,j)`. Without the replacement the complexity would be $O(n^2)$, as the operator `<=` has dense efficiency.

Storing common domains

A second opportunity for gaining domain-related efficiency can be reached by carefully examining your model and identifying those complex compound conditions that are used in the specification of the index domains of multiple identifiers and/or as a domain restriction in multiple assignments or FOR loops. It then certainly makes sense to store such a common domain of computation in a defined parameter (as above), and replace the explicit conditions by a reference to this domain parameter.

Execution and storage efficiency

Not only do you gain execution efficiency by replacing common explicit conditions, you also prevent AIMMS from generating multiple *identical* domain parameters itself. Thus, you potentially gain storage efficiency as well.

Breaking down domains

In many applications it is possible to break down complicated high-dimensional domain conditions into multiple simpler lower-dimensional conditions on subselections of the indices in the domain. As illustrated in the example below, such a break down usually results in a much faster computation than the original condition.

Secondary effect

However, in practice the addition of simpler domain parameters may lead to secondary efficiency improvements as well, although this may require some additional effort on your side. Frequently, the conditions expressed in these newly introduced domain parameters provide additional structural information to your application that you had not thought of before. You can quite often use this information in other parts of your model as well, either to speed up the execution of your model by replacing densely by sparsely executed statements, or by adding them as conditions to statements which you unnoticedly executed over a too large domain before.

Example

The logical domain of computation introduced in the assignment statement below is a good example of a domain that may appear at multiple places in the model. As illustrated there, the efficiency benefits of explicitly constructing this domain can be considerable. In case the full 4-dimensional domain of computation is used more than once, one might consider storing this in a separate identifier as well for optimal efficiency.

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 into a set of Regions and st and dt (source terminal and destination terminal) are indices into a set of Terminals.

```
Transport( (sr,st,dt,dr) | sr <> dr and st <> dt and
                             TRDistance(sr,st) <= MaxTRDistance(st) and
                             TRDistance(dr,dt) <= MaxTRDistance(dt) and
           MinTransDistance <= RRDistance(sr,dr) <= MaxTransDistance  and
           MinTransDistance <= TTDistance(st,dt) <= MaxTransDistance  and
         ) := Demand(sr,dr);
```

It states that region-terminal-terminal-region transports should only be assigned if various distances between regions and/or terminals satisfy the given bounds. Because of the unfavorable efficiency of the <= operators contained in it, the statement demonstrates $O(m^2n^2)$ computational behavior (where m is the cardinality of the set `Regions` and n that of `Terminals`).

... and its reformulation

A solution to reformulating this assignment more efficiently can be found in formulating the following logical conditions.

```
RegionalTerminal( (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;
```

These assignments have $O(mn)$, $O(n^2)$ and $O(m^2)$ dense computational behavior, respectively. With them, the original assignment can be reformulated as follows.

```
Transport( (sr,st,dt,dr) |
           RegionalTerminal(sr,st) and RegionalTerminal(dr,dt) and
           ConnectableRegions(sr,dr) and ConnectableTerminals(st,dt)
         ) := Demand(sr,dr);
```

Although the precise computational complexity is not easily expressed anymore, the initial dense efficiency has been replaced by the much more efficient intersection behavior, at the cost of some additional lower order computations and some additional storage. As the names of the additional domain parameters already indicate, it is not unlikely that they can be used at other places in the model as well.

13.6 Definitions versus assignments

Definitions versus assignments

The use of defined parameters in your model is a very appealing concept, providing a single localized entry in the model tree for both the parameter declaration and the expression necessary to compute its intended contents. This appealing nature of definitions, however, may lure you into also using parameter definitions when the use of an equivalent assignment would lead to much more efficient execution.

Dependency is symbolic

As explained in Section 7.1, the dependency structure between set and parameter definitions is purely based on symbol references. Consequently, AIMMS' automatic evaluation scheme can only recompute a defined indexed (output) parameter depending on an indexed (input) parameter *in its entirety*, even when only a single input entry of the input parameter has changed. As you will see in this section through an illustrative example, this may lead to a severe penalty in the execution times of your model.

When to avoid definitions

You should avoid the use of defined parameters when your model contains a loop in which only data of a slice of one or more of the input parameters of a definition is modified, and, consequently, only the corresponding slice of that parameter needs to be re-evaluated during each iteration. When you use a definition in such a case, AIMMS' automatic evaluation scheme has no way to detect that only a particular slice has to be updated and will re-evaluate the definition for the entire domain.

A simulation example

The most notable example where such behavior might play an important role in reducing the execution times of your model is in a simulation over time. In simulations, computations are usually performed period by period referring to data of the previous period. You should avoid definitions for time-dependent parameters when these are modified and referenced during the computation for every period, as illustrated in the example below.

Example

The relation for the computation of the stock of a particular product `p` in period `t` can easily expressed by, for instance, the following definition.

```
PARAMETER:
    identifier   : ProductStock
    index domain : (p,t)
    definition   : ProductStock(p,t-1) + Production(p,t) - Sales(p,t)
```

Although this relation is true for every period, you should not use it when your model a simulation loop that references `ProductStock`. The following loop provides an overly simplified example.

```
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` for *all* periods before executing the assignment for the next period. This results in a complexity of $O(mn^2)$ is there are m products and n simulation periods.

Improved solution

In this case, execution times can be greatly reduced by omitting the definition of `ProductStock`, and adding the computation of `ProductStock` for only the current period explicitly to the time loop.

```
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;
```

Thus, the resulting complexity of the second solution is $O(mn)$.

Use of macros

If efficiency considerations prohibit the use of definitions for particular identifiers in your model, you might still consider the use of a `MACRO` (see also Section 6.4) to localize the defining expression at a single node in the model tree. This is especially useful if you need to add assignments for the previously defined parameter at several places in your model.

13.7 Efficiency of set-related operations

Set-related efficiency

Another resort for gaining execution efficiency is a thorough understanding of the ways in which AIMMS uses roots sets for both storage and execution. In this section, you will find the basic principles and some common pitfalls which may lead to inefficient coding of your procedures and functions.

13.7.1 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 e.g. 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

All storage of multidimensional data defined over a root set is 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 tour through all the nonzero 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 when 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 `ORDER BY` attribute of the set declaration or through an explicit `Sort` operation. Consequently, there is no direct match anymore between the execution order of the loop and the storage order of nonzero identifier values. Depending on the precise type of statement, this may result in no, slight or serious deterioration 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 expensive 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 into a set with a nondefault ordering whenever possible. 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 case, you should carefully evaluate whether ordered execution is really imperative. If not, it is advisable to leave the original set unordered, and create an ordered *subset* (containing all elements of the original set) for use when the nondefault element ordering is required.

Effect on assignments

In most cases, the efficiency of indexed assignments is not affected by the use of indices into sets with a nondefault ordering. AIMMS only has 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

When a nondefault ordering of some sets in your model causes a serious deterioration in execution times, you may want to apply the `CLEANDEPENDENTS` statement (see also Section 18.3) to those roots sets that are the cause of the deterioration of execution times. The `CLEANDEPENDENTS` statement will completely 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 rebuilt completely, the `CLEANDEPENDENTS` is an inherently expensive operation. You should, therefore, only use it when really necessary.

13.7.2 Lag and lead operators

Lag and lead operators

Assignments using lag and lead operators are less efficient than normal assignments, because AIMMS has to perform explicit lookups in its sparse data structures to compute the lag and lead expressions, similar as with ordered sets (see also Section 13.7.1). For this reason, if you use lag and lead operators on big sets or if a parameter or variable appears several times in your model

with the same lag or lead operation on its index, you can save execution time by first assigning it to another parameter and using that parameter in your calculations.

Example

Consider the following assignments.

```
Result(i,j)  := Sparse(i-1) + Sparse(j);

SparseLag(i) := Sparse(i-1);
Result(i,j)  := SparseLag(i) + Sparse(j);
```

In the first assignment AIMMS will have to check for every `i` whether `Sparse(i-1)` holds a nonzero value. In the second assignment to `Result(i,j)` AIMMS will directly use the sparsity of `SparseLag(i)`.

13.7.3 Element parameters versus single entry subsets

Elements versus single entry subsets

Before the availability of element parameters in AIMMS 3.0, it was common practice to use single entry subsets to reference single elements. Compared to the use of element parameters, single entry subsets are much less efficient, however, and it is advisable to convert all such references in your previous AIMMS models. Moreover, the use of element parameters is more intuitive, and thus makes your model easier to understand.

Example

Consider the following assignments, where `SelectedCity` is an element parameter into the set `Cities`, and `SingleCitySet` a single entry subset of `Cities` with index `sc`.

```
SelectedCitySupply := Sum(sc, Supply(sc));

SelectedCitySupply := Supply(SelectedCity);
```

Not only is the second assignment more intuitive, it will also execute faster because AIMMS does not have to loop over the set `SingleCitySet`.

13.7.4 Subset indexing

Subset indexing

Subset indexing is faster than indexing over the full domain with a conditional expression. If you have the same expression appearing multiple times in your code, then you should consider forming the subset and using its index instead of the conditional expression.

The use of subsets for indexing is not limited to simple sets only. When you index over a sparse compound subset multiple times, your model may benefit from using a compound index in that set.

Simple and compound

Consider the following three groups of statements.

Example

```
Sparse((i,j) | a(i) and b(j)) := Dense(i,j);

Subset1 := {i | a(i)};                    ! with index i1
Subset2 := {j | b(j)};                    ! with index j1
Sparse(i1,j1) := Dense(i1,j1);            ! use simple subsets

Compound := {(i,j) | a(i) and b(j)};      ! with index c
Sparse(c) := Dense(c);                    ! use compound subset
```

The statements in each group perform the same assignment to `Sparse(i,j)`, but require a different number of operations.

13.7.5 Reducing the indexing dimension

When an assignment to a multidimensional identifier must only be performed when some condition between its indices is fulfilled, one might be tempted to index over the full domain subject to that condition. When the condition maps one index uniquely to another index, it is often possible to reformulate the assignment such that the indexing dimension is reduced by one.

Reducing indexing dimension

The following statements illustrate two assignments that can be reformulated to reduce the indexing dimension by one.

Examples

```
Sparse1( (i,j) | j = i + 1  ) := 1;
Sparse2( (i,j) | j = Map(i) ) := 1;

Sparse1(i,i+1   ) := 1;
Sparse2(i,Map(i)) := 1;
```

Without the index reduction, such statements may often give rise to dense execution behavior, while this is not necessarily the case for the reduced statements.

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

Declarations 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 INDEX DOMAIN 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
INDEX DOMAIN	*index-domain*	41
RANGE	*range*	42
DEFAULT	*constant-expression*	43
UNIT	*unit-valued expression*	44, 269
PRIORITY	*expression*	
NONVAR STATUS	*expression*	
RELAX STATUS	*expression*	
PROPERTY	NoSave, *numeric-storage-property*, Inline, SemiContinuous, ReducedCost, ValueRange, CoefficientRange, *constraint-related-sensitivity-property*	33, 44
TEXT	*string*	19
COMMENT	*comment string*	19
DEFINITION	*expression*	33, 43

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:
    identifier   : Transport
    index domain : (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:
    identifier   : Shipment
    index domain : (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 23.

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.

Example

The following example defines the total cost of transport, based on unit transport cost and actual transport taking place.

```
VARIABLE:
   identifier : TransportCost
   definition : sum( (i,j), UnitTransportCost(i,j)*Transport(i,j) );
```

14.1.1 The PRIORITY, NONVAR and RELAX STATUS attributes

The PRIORITY attribute

With the PRIORITY attribute you can assign priorities to integer variables. 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. 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 smallest 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 NONVAR STATUS attribute

You can use the NONVAR STATUS attribute to tell AIMMS which variables should be considered as parameters during the execution of a SOLVE statement. The value of the NONVAR STATUS 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 NONVAR STATUS 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 NONVAR 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 RELAX STATUS attribute

With the RELAX STATUS 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 through assignments to the .Relax suffix. This is only allowed if you have not specified the RELAX 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, and ValueRange. 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 replace 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.

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.

Basic, superbasic, and nonbasic

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. You can recognize the difference, because basic variables have a reduced cost value of 0.0, whereas the superbasic variables have a reduced cost of `ZERO`.

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. 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. 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.

Storage and computational costs

Setting any of the properties `ReducedCost`, `CoefficientRange` or `ValueRange` may result increase the use of memory considerably. 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.

14.2 `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.

Attribute	Value-type	See also page
`INDEX DOMAIN`	*index-domain*	41
`UNIT`	*unit-valued expression*	44, 269
`TEXT`	*string*	19
`COMMENT`	*comment string*	19
`DEFINITION`	*expression*	43, 179
`PROPERTY`	`NoSave`, `Sos1`, `Sos2`, `Level`, `Bound`, `ShadowPrice`, `RightHandSideRange`, `ShadowPriceRange`	33, 44, 181

Table 14.2: `CONSTRAINT` attributes

Domain restriction for constraints

Restricting the domain of constraints through the `INDEX DOMAIN` attribute influences the matrix generation process. Constraints are generated only for those tuples in the index domain that satisfy the domain restriction.

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.

```
CONSTRAINTS:
    identifier   : SupplyConstraint
    index domain : i
    unit         : kton
    definition   : sum( j, Transport(i,j) ) <= Supply(i) ;

    identifier   : DemandConstraint
    index domain : 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, 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.

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.

Sos1 constraints

A type Sos1 constraint specifies to the solver that at most one of the variables inside the constraint is allowed to be nonzero, while all other variables must be zero. Inside a Sos1 constraint all variables 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 inside the constraint are allowed to be nonzero, while all other variables must be zero. Only one symbolic variable is permitted inside a Sos2 constraint, and all individual members must have a lower bound of zero and an upper bound greater than zero. The order of the individual members of the symbolic variable is determined by its index order. The right-most index is the least significant.

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 and ShadowPrice properties

With the Level, Bound, 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 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.

- 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.

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. The result of the sensitivity analysis are three parameters defined over the domain of the constraint. The values assigned to the parameters will be the smallest, nominal, and largest values for the *right- or left-hand side* of the constraint so that the basis remains constant. There are two cases:

- if the constraint is binding, the smallest, nominal,and largest value for the binding side of the constraint are reported
- if the constraint is not binding, 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. 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`.

Chapter 15

Solving Mathematical Programs

A *mathematical program* consists of

Mathematical program components

- 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.

Depending on the characteristics of the variables and constraints, a mathematical program in AIMMS can be classified as one of the following.

Different types

- 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 first discusses the declaration of a mathematical program, together with auxiliary functions that you can use to specify its set of variables and constraints. At the end, the `SOLVE` execution statement needed to solve any type of mathematical program is presented.

This chapter

15.1 `MATHEMATICAL PROGRAM` 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*	
TEXT	*string*	19
COMMENT	*comment string*	19
CONVENTION	*convention*	279

Table 15.1: MATHEMATICAL PROGRAM attributes

Example

The following example illustrates a typical mathematical program.

```
MATHEMATICAL PROGRAM
    name        : 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.

The OBJECTIVE attribute

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.

Omitting the objective

If you do not specify an objective, your mathematical program will be solved to find a feasible solution and it will then terminate.

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 NONVAR STATUS

The VARIABLES attribute performs a similar function as the NONVAR STATUS 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 NONVAR STATUS (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.

```
MATHEMATICAL PROGRAM
    name        : 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.

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
`network`	pure network program

Table 15.2: Available model types with AIMMS

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 23.6.

15.2 Suffices and callbacks

Suffices

A mathematical program has a number of suffices which can be used for various purposes. Typical examples are:

- to obtain information about the solution process,
- to determine when and how to activate a callback procedure, and
- to get general information about the generated mathematical program.

The complete list of suffices of a mathematical program and their meaning are given in Table 15.3.

Suffix	Meaning	Set by
`Objective`	Current objective value	Solver
`LinearObjective`	Current linear objective value	
`Incumbent`	Current incumbent value	
`ProgramStatus`	Current program status	
`SolverStatus`	Current solver status	
`Iterations`	Current number of iterations	
`SolutionTime`	Current solution time in seconds	
`CallbackProcedure`	Name of callback procedure	User
`CallbackIterations`	Return to callback after this number of iterations	
`CallbackStatusChange`	Name of callback procedure to be called after a status change	
`CallbackNewIncumbent`	Name of callback procedure to be called for every new incumbent value	
`CallbackReturnStatus`	Return status of callback	
`SolverCalls`	Total number of applied `SOLVE`'s	AIMMS
`NumberOfConstraints`	Number of individual constraints	
`NumberOfVariables`	Number of individual variables	
`NumberOfNonzeros`	Number of nonzeros	
`NumberOfInfeasibilities`	Final number of infeasibilities	Solver
`SumOfInfeasibilities`	Final sum of the infeasibilities	

Table 15.3: Suffices of a mathematical program

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 is 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.

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 change of status of the solution process, or
- at every new incumbent value 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.

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 `CallbackNewIncumbent` you specify the name of the callback procedure to be performed when the solver finds a new incumbent value during the solution process of mixed integer program. When not specified (the default), no such callbacks are made.

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.

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 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 15.4.

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
UnknownError	InternalSolverError
NoSolution	PostProcessorError
	SystemFailure

Table 15.4: Mathematical program and solver status

15.3 The SOLVE statement

The SOLVE statement

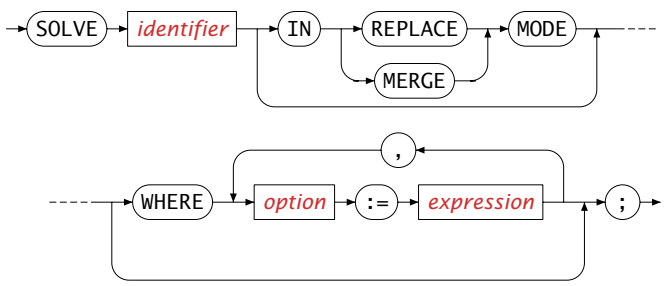
With the SOLVE statement you can instruct AIMMS to compute the solution of a MATHEMATICAL PROGRAM, resulting in the following actions.

- AIMMS determines which solution method(s) are appropriate, and checks whether the specified type is also appropriate.
- 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 MATHEMATICAL PROGRAM, 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 NONVAR STATUS attribute or the .NonVar suffix of the variable,
- because they are outside the planning interval of a HORIZON (see Section 24.3), 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-crossover'`.

```
solve TransportModel in replace mode
    where solver    := 'cplex',
          type      := 'rmip',
          LpMethod  := 'Barrier-crossover' ;
```

Chapter 16

Node and Arc Declaration

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.

This chapter

16.1 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.

Networks

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.

Mixed formulations

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:

Flow keywords

- `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.

16.2 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 16.1.

Attribute	Value-type	See also page
INDEX DOMAIN	*index-domain*	41, 176, 183
UNIT	*unit-valued expression*	44, 178
TEXT	*string*	19, 44
COMMENT	*comment string*	19
DEFINITION	*expression*	184
PROPERTY	NoSave, Sos1, Sos2, Level, Bound, ShadowPrice, RightHandSideRange, ShadowPriceRange	44, 181, 185

Table 16.1: 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:
     identifier   : CustomerDemandNode
     index domain : (j in Customers, p in Products)
     definition   :
        NetInflow >= ProductDemanded(j,p) ;

NODE:
     identifier   : DepotStockSupplyNode
     index domain : (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)`.

16.3 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 16.2.

Attribute	Value-type	See also page
INDEX DOMAIN	*index-domain*	41
RANGE	*range*	176
DEFAULT	*constant-expression*	43, 178
FROM	*node-reference*	
FROM MULTIPLIER	*expression*	
TO	*node-reference*	
TO MULTIPLIER	*expression*	
COST	*expression*	
UNIT	*unit-valued expression*	178
PRIORITY	*expression*	179
NONVAR STATUS	*expression*	180
RELAX STATUS	*expression*	180
PROPERTY	NoSave, *numeric-storage-property*, Inline, SemiContinuous, ReducedCost, ValueRange, CoefficientRange	33, 44, 181
TEXT	*string*	19, 44
COMMENT	*comment string*	19

Table 16.2: 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, FROM MULTIPLIER, TO MULTIPLIER, 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 FROM MULTIPLIER and TO MULTIPLIER 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*.

FROM and TO MULTIPLIER

The FROM MULTIPLIER is the conversion factor of the flow at the source node, while the TO MULTIPLIER 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 FROM and TO MULTIPLIERS, the drawing in Figure 16.1 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).

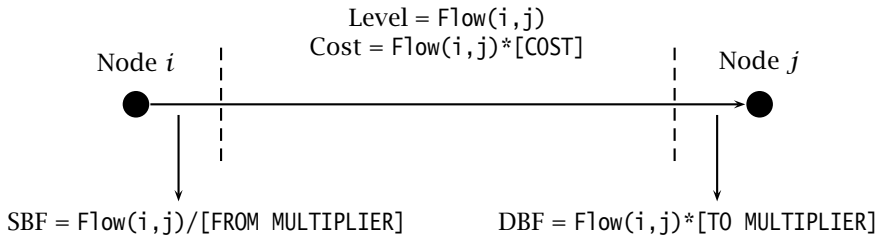


Figure 16.1: Flow levels and cost from node i to node j

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:
     identifier   :  Transport
     index domain :  (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`.

16.4 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.

```
MATHEMATICAL PROGRAM:
    identifier  : 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.

Chapter 17

Matrix Manipulation

This chapter introduces a collection of matrix manipulation procedures that allow you to implement efficient algorithms for the sequential solution of linear and mixed-integer linear programming models. These procedures operate directly on the matrix underlying the mathematical program, and thus avoid the constraint-generation process normally required to solve a mathematical program after input data has been modified. The matrix manipulation procedures described in this chapter are only applicable when the underlying algorithm requires no end-user input. In addition to the basic principles and a brief description of the particular matrix manipulation procedures, you will find four examples illustrating the use of these procedures.

This chapter

17.1 Introduction

This section clarifies the need for a collection of matrix manipulation procedures for the efficient updating of solver input associated with linear mathematical programs. The distinction between manual and automatic input data changes inside an AIMMS application is emphasized, thereby providing the justification for the special matrix manipulation procedures referred to in this chapter.

This section

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.

Manual data input ...

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.

... requires structure recognition

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.

For linear programs only

The above effects of data input changes on the input to the solver are straightforward to implement with linear mathematical programs, because the underlying data structure is a matrix with rows, columns and nonzero elements. The input data structures for nonlinear mathematical programs are essentially nonlinear expressions, and modifications of the type discussed in the previous paragraph are not easily passed on to these nonlinear data structures. For this reason, the efficient updating of solver input has been confined to linear mathematical programs only.

17.2 Matrix manipulation procedures

Matrix manipulation procedures

The complete collection of matrix manipulation procedures is listed in Table 17.1. They all start with the prefix "Matrix" indicating their applicability. The table is divided into several blocks, and each block covers a coherent group of procedures. There are modification procedures at the level of matrix coefficients, rows, columns, and mathematical programs as a whole. Each group is discussed in a separate paragraph.

Scalar arguments only

All matrix procedures listed in Table 17.1 have scalar-valued arguments. The *row* argument should always be a scalar reference to an existing constraint name in your model. The *column* argument should always be a scalar reference to an existing variable name in your model. Thus, any indices used in the

`MatrixModifyCoefficient`(*MP*, *row*, *column*, *value*)
`MatrixModifyRightHandSide`(*MP*, *row*, *value*) `MatrixModifyLeftHandSide`(*MP*, *row*, *value*) `MatrixModifyRowType`(*MP*, *row*, *type*) `MatrixAddRow`(*MP*, *row*) `MatrixRegenerateRow`(*MP*, *row*) `MatrixDeactivateRow`(*MP*, *row*) `MatrixActivateRow`(*MP*, *row*)
`MatrixModifyLowerBound`(*MP*, *column*, *value*) `MatrixModifyUpperBound`(*MP*, *column*, *value*) `MatrixModifyColumnType`(*MP*, *column*, *type*) `MatrixAddColumn`(*MP*, *column*) `MatrixFreezeColumn`(*MP*, *column*, *value*) `MatrixUnfreezeColumn`(*MP*, *column*)
`MatrixModifyType`(*MP*, *type*) `MatrixModifyDirection`(*MP*, *direction*) `MatrixReSolve`(*MP*)
`MatrixSaveState`(*MP*, *state*) `MatrixRestoreState`(*MP*, *state*)

Table 17.1: Matrix manipulation procedures

expressions for the *row*, *column* and *value* arguments should be bound prior to calling any of these procedures.

Initial SOLVE required

Before you can apply any of the procedures of Table 17.1 to a mathematical program, you must first solve it using the regular AIMMS `SOLVE` statement, *even if this is not necessary for your algorithm*. This call to the `SOLVE` statement will set up the initial row-column matrix required by the matrix manipulation procedures. Also, any row or column referenced in the procedures must either be generated by the initial `SOLVE` statement, or by a call to `MatrixAddRow` or `MatrixAddColumn`, respectively.

Coefficient procedure

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 `MatrixModifyCoefficient`. This procedure can also be used when a value for the coefficient does not exist prior to calling the procedure. The argument *MP* refers to the mathematical program in hand, allowing you to modify several mathematical programs inside a single overall procedure.

Tags are optional

The argument names listed in Table 17.1 can also be used as tags in any call to one of the procedures listed. For instance, the following two calls are valid

and identical.

```
MatrixModifyCoefficient ( LinearizedProgram, ObjectiveRow, x(j), ObjCoeff(j));

MatrixModifyCoefficient ( MP     : LinearizedProgram,
                          row    : ObjectiveRow,
                          column : x(j),
                          value  : ObjCoeff(j) );
```

Because the arguments to all matrix manipulation procedures should be scalar, both calls to `MatrixModifyCoefficient` should be surrounded by a `FOR` statement that binds the index `j`, when called within a procedure of your model.

Row procedures

The second block of matrix manipulation procedures refers to rows. You can use these procedures to change any aspect of an existing row, or even include a new row. 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, the newly added row will initially not have any nonzero coefficients, regardless of whether the corresponding AIMMS constraint had a definition or not. Through the procedure `MatrixRegenerateRow` 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.

Column procedures

The third block of matrix manipulation procedures refers to columns. As with rows, you can use these procedures to change any aspect of an existing column, or even include a new column. 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.

Mathematical program procedures

The last two blocks of matrix manipulation procedures refer to mathematical programs as a whole. The type of a mathematical program can be (see also section 15.1)

- 'LP',
- 'LS',
- 'MIP', or
- 'RMIP'.

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.

Saving state

Through the procedures MatrixSaveState and MatrixRestoreState you can save and subsequently restore the entire data structure underlying a mathematical program. These facilities are ideal when you are implementing a custom branch-and-bound approach inside an AIMMS application.

17.3 Examples of use

This section

In this section there are four examples to illustrate the use of matrix manipulations. 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 matrix manipulation 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.

17.3.1 Sensitivity analysis

Parametric changes

Sensitivity analysis considers how the optimal solution, and the corresponding objective function value, change as a result of changes in input data. Using the matrix manipulation functions, it is straightforward to write a procedure to determine these sensitivities for a discrete set of input values.

Procedure in AIMMS

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.

```
solve MathProgramOfInterest;

for ( n ) do
```

```
    MatrixModifyCoefficient( MathProgramOfInterest,
                             ResourceConstraint(SelectedResource),
                             ActivityVariable(SelectedActivity),
                             OriginalCoefficient + Delta(n) );
    MatrixReSolve( MathProgramOfInterest );

    ObjectiveValue(n) := MathProgramOfInterest.Objective;
  endfor;
```

17.3.2 Finding a feasible solution for a binary program

Fixing one variable at a time

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

- 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 there is no solution, or all variables have been fixed.

```
solve RelaxedBinaryProgram;

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 MatrixFreezeColumn( RelaxedBinaryProgram, x(SmallestGreaterThanZero), 0 );
    else MatrixFreezeColumn( RelaxedBinaryProgram, x(LargestLessThanOne), 1 );
    endif;

    MatrixReSolve( RelaxedBinaryProgram );
endrepeat;
```

17.3.3 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.

```
solve CuttingStock;
solve FindPattern;

while ( PatternContribution > 1 ) do
    MaxPattern += 1;
    MatrixAddColumn( MP: CuttingStock, column: RollsUsed(MaxPattern) );

    for ( width ) do
        MatrixModifyCoefficient( MP    : CuttingStock,
                                 row   : MeetCutDemand(width),
                                 column: RollsUsed(MaxPattern),
                                 value : CutsInPattern(width)  );
    endfor;
    MatrixReSolve( CuttingStock );

    for ( width ) do
        MatrixModifyCoefficient( MP    : FindPattern,
                                 row   : PatternContribution,
                                 column: CutsInPattern(width),
                                 value : MeetCutDemand(width).ShadowPrice );
    endfor;
    MatrixReSolve( FindPattern );
endwhile;
```

17.3.4 Sequential linear programming

Sequential linear programming

Linear constraints and a nonlinear objective function together form a special class of nonlinear program. 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 availability of a function to compute the required first partial derivatives with respect to the variables in the objective function.

```
solve LinearizedProgram;

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
      MatrixModifyLowerBound ( LinearizedProgram, x(j),
                               max(x.lower(j), x(j) - 0.5*BoxWidth(j)) );
      MatrixModifyUpperBound ( LinearizedProgram, x(j),
                               min(x.upper(j), x(j) + 0.5*BoxWidth(j)) );
      MatrixModifyCoefficient( LinearizedProgram, ObjectiveRow,
                               x(j), ObjCoeff(j)                      );
   endfor;
   MatrixResolve( LinearizedProgram );

   BoxWidth(j) *= ShrinkFactor;
endwhile;
```

Part VI

Data Communication Components

Chapter 18

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 ASCII files, cases and external databases are discussed in subsequent chapters.

18.1 Data initialization

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 ASCII 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.

Data initialization sequence

To initialize the data in your model, AIMMS performs the following actions directly after compiling the model:

- first AIMMS fills the contents of any global set or parameter with the contents of its `INITIAL DATA` attribute, and
- then AIMMS execute the predefined procedure `MainInitialization`.

The `MainInitialization` procedure

AIMMS will add the procedure `MainInitialization` to a new project automatically. Initially it is empty, leaving the (optional) specification of its body to you. You can use this procedure 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.

The attribute `INITIAL DATA`

Both sets and parameters can have an `INITIAL DATA` 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 `INITIAL DATA` 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 `INITIAL DATA` attribute must be a *constant expression* (i.e. a constant, a constant enumerated set or a constant list expression) or a `DATA TABLE`. The table format is explained in Section 21.2.

18.1.1 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 ASCII files containing constant lists and tables,
- AIMMS-generated binary case files, and
- external ODBC compliant databases.

Reading from ASCII data files

With the `READ` statement you can initialize selected model input data from ASCII files containing explicit data assignments. Only `DATA TABLES` 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 ASCII 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.

18.2 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 18.1.

Attribute	Value-type	See also page
INDEX DOMAIN	*index-domain*	41, 176
TEXT	*string*	19, 44
PROPERTY	WarnOnly	
ASSERT LIMIT	*integer*	
DEFINITION	*logical-expression*	33, 43
ACTION	*statements*	
COMMENT	*comment string*	19

Table 18.1: 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. By default, the execution is halted. Examples follow.

Examples

```
ASSERTION:
    identifier   :  SupplyExceedsDemand
    text         :  "Error: Total demand exceeds total supply"
    definition   :  Sum( i in Cities, Supply(i) ) >=
                    Sum( i in Cities, Demand(i) )       ;
ASSERTION:
    identifier   :  CheckTransportData
    index domain :  (i,j) | Distance(i,j)
    text         :  "Please supply proper transport data for transport (i,j)"
    assert limit :  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 non-zero 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 shown when the assertion fails for an element in its domain. The text is displayed in the message window, as well as in a message dialog box to the user. 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 will not be displayed.

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 displayed. By default, AIMMS will halt the current execution.

The ASSERT LIMIT attribute

By default, AIMMS will verify an assertion for every element in its index domain, and display a message dialog box for every element for which the assertion fails. With the ASSERT LIMIT attribute you can limit the number of messages displayed. When the number of failed assertions reaches the ASSERT LIMIT, AIMMS will terminate the verification of the assertion. By default, the ASSERT LIMIT is set to 1.

The ACTION attribute

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.

The FailCount operator

If you 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 ASSERT LIMIT. 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

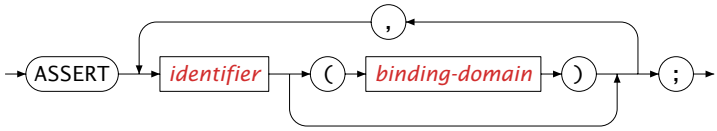
Assertions can be verified in two ways:

- 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);
```

18.3 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

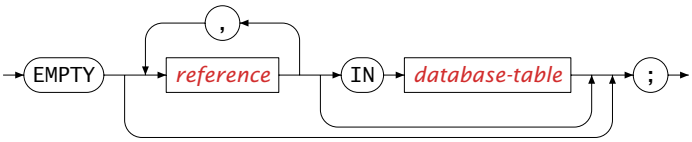
AIMMS offers the following data control tools:

- 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 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 20.4. 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.

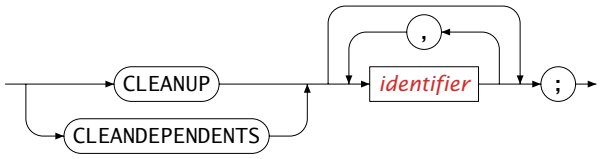
When useful

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. 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 or CLEANUP 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 13.7.1. This section also explains the benefits of resorting a root set.

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;
```

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;
```

Chapter 19

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

- *ASCII data files*,
- AIMMS *case 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.

19.1 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 ASCII files, AIMMS cases and database tables. It is based on the familiar transport problem with the following input data:

- the set Cities,
- the compound set 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 ASCII files as in Table 19.1 (format explained in Section 21.3). Alternatively, the data can be contained in binary AIMMS case files, or 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 19.1: Example data set for the transport model

19.1.1 Simple data transfer

Simple data initialization

The simplest use of the READ statement is to initialize data from a fixed name ASCII data file, a binary case file, or a database table. To read all the data from each source, the following groups of statements will suffice, where SelectedCaseFile is an element parameter into the set of AllDataFiles.

```
read from file  "transport.inp" ;

read from case  SelectedCaseFile ;

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 either a binary case file or 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 ASCII 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 ASCII files only).

19.1.2 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

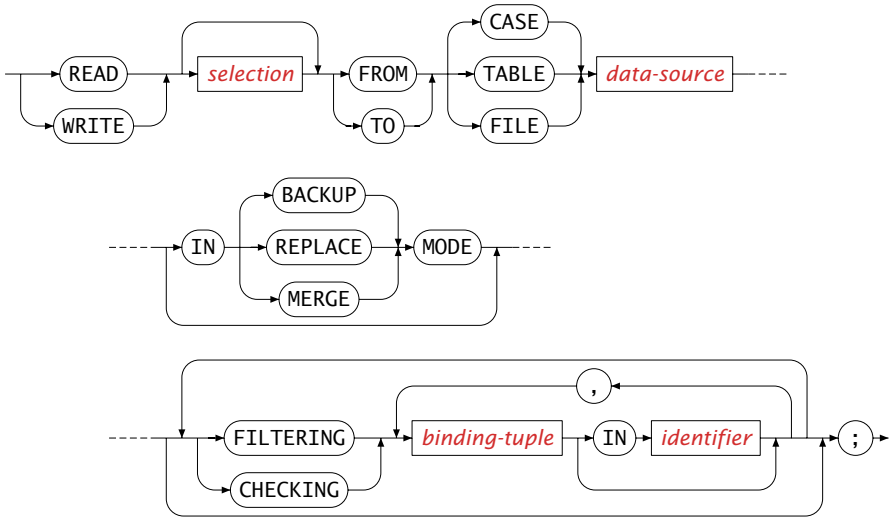
19.2 Syntax of the READ and WRITE statements

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.

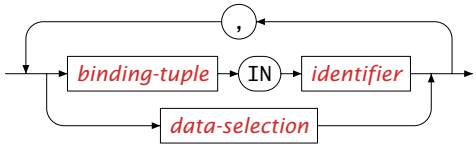
READ and WRITE statements

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 CASE is represented by an element into the set AllDataFiles,
- a TABLE represented by a DATABASE TABLE identifier.

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. The elements of the predefined set AllDataFiles refer to cases and datasets created in the AIMMS data manager tool. AIMMS provides a number of functions that you use within your model to associate a case or dataset name with an element of AllDataFiles, or to create new cases or datasets from within your model. A description of these functions can be found in the AIMMS User's Guide.

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 an ASCII file, an AIMMS case 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 an ASCII or case 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 ASCII 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 either *merge* mode or *replace* 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. When you are writing data to an ASCII data file, AIMMS also supports a *backup* mode.

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 an ASCII file, AIMMS will *append* the newly written data to the end of the file.
- If the data source is an AIMMS case file or 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 an ASCII 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 an AIMMS case file, AIMMS will overwrite the data of all identifiers in the identifier selection.
- If the data source is a database table, AIMMS will empty all columns in the table that are mapped onto identifiers in the identifier selection.

Writing in backup mode

When you are transferring data to an ASCII 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 an ASCII 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 or case file.

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 × 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 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 an ASCII file or AIMMS case 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 an ASCII data file or an AIMMS case file.

Chapter 20

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.

Communicating with databases

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.

This chapter

20.1 The DATABASE TABLE declaration

You can make a database table known to AIMMS by means of a DATABASE TABLE 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 DATABASE TABLE is given in Table 20.1.

Database tables

Attribute	Value-type	See also page
INDEX DOMAIN	*index-domain*	41
DATA SOURCE	*string-expression*	
TABLE NAME	*string-expression*	
OWNER	*string-expression*	
PROPERTY	ReadOnly	
MAPPING	*mapping-list*	
TEXT	*string*	19
COMMENT	*comment string*	19
CONVENTION	*convention*	279

Table 20.1: DATABASE TABLE attributes

The DATA SOURCE attribute

The mandatory DATA SOURCE 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 DATA SOURCE wizard. 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 TABLE NAME attribute

With the TABLE NAME attribute you must specify the name of the table or view within the data source to which the DATABASE TABLE is mapped. Once you have provided the DATA SOURCE attribute, the TABLE NAME wizard will let you select any table or view available in the specified data source.

Example

The following declaration illustrates the simplest possible DATABASE TABLE declaration.

```
DATABASE TABLE:
    identifier  :  RouteData
    data source :  "Topological Data"
    table name  :  "Route Definition" ;
```

It will connect to an ODBC 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 DATABASE TABLE 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. This 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 column names from the database table used in a mapping list must be quoted.

Example

The following declarations demonstrate the use of mappings in a DATABASE TABLE declaration. This example assumes the set and parameter declarations of Section 19.1 plus the existence of the compound set Routes given by

```
SET:
    identifier  : Routes
    subset of   : (Cities, Cities) ;
```

The following mapped database declaration will take care of the necessary column to identifier mapping.

```
DATABASE TABLE:
    identifier   :  RouteData
    data source  :  "Topological Data"
    table name   :  "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 set
```

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

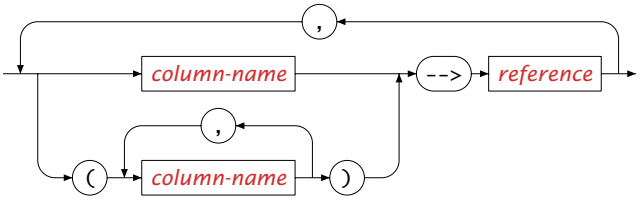
If the number of attributes in the primary key of a database table is lower than the dimension of the intended AIMMS identifier, you can also map a column name to a *slice* of an AIMMS identifier of the proper dimension, as shown in the fcost and vcost mapping. You can do this by replacing one or more of the indices in the identifier's index space with a reference to a fixed element.

Mapping primary key to set

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 or compound) set 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 compound set.

The syntax of the MAPPING attribute is given by the following diagram. *Syntax*

mapping-list :



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 23.6. *The CONVENTION attribute*

20.2 Indexed database tables

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. *Exogenous columns in primary key*

- 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.

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 INDEX DOMAIN 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. *Indexed DATABASE TABLES*

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:
    identifier : TableVersions
    index      : v
    parameter  : LatestVersion ;
```

The following declaration will provide a number of virtual tables indexed by v.

```
DATABASE TABLE:
    identifier   :  RouteData
    index domain :  v
    data source  :  "Topological Data"
    table name   :  "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 DATABASE TABLE in AIMMS

  ```
  DATABASE TABLE:
      identifier   :  VersionTable
      data source  :  "Topological Data"
      table name   :  "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) ;
  ```

20.3 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* ASCII 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 20.5.

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 setup 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 ASCII 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 or compound) set 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

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 `DATABASE TABLE` 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 DATABASE PROCEDURE 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 20.5).

20.4 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 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 18.3.

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 `LatestVersion` from the indexed table `RouteTable(v)`.

```
empty Transport in RouteTable(LatestVersion) ;
```

20.5 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 `DATABASE TABLE` 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.

The `DATABASE PROCEDURE` declaration

Every stored procedure or SQL query that you want to call from within AIMMS must be declared as a `DATABASE PROCEDURE` within your application. The attributes of a `DATABASE PROCEDURE` are listed in Table 20.2.

SQL query or stored procedure

A `DATABASE PROCEDURE` 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 `STORED PROCEDURE` and `SQL QUERY` attributes. If the `STORED PROCEDURE` attribute is nonempty, AIMMS assumes that the `DATABASE PROCEDURE` represents a stored procedure and expects the `SQL QUERY` attribute to be empty, and vice versa.

Attribute	Value type	See also page
DATA SOURCE	*string*	232
ARGUMENTS	*argument-list*	130
STORED PROCEDURE	*string-expression*	
SQL QUERY	*string-expression*	
OWNER	*string-expression*	232
PROPERTY	UseResultSet	
MAPPING	*mapping-list*	232
COMMENT	*comment string*	19
CONVENTION	*convention*	234, 279

Table 20.2: DATABASE PROCEDURE attributes

The STORED PROCEDURE attribute

With the STORED PROCEDURE attribute you can specify the name of the stored procedure within the ODBC data source that you want to be called. The STORED PROCEDURE 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 SQL QUERY attribute

You can use the SQL QUERY attribute to specify the SQL query that you want to be executed when the DATABASE PROCEDURE 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 DATABASE PROCEDURE.

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 DATABASE PROCEDURE. If the DATABASE PROCEDURE represents a stored procedure, the argument list is interpreted as the argument list of the stored procedure. When you use the STORED PROCEDURE wizard, AIMMS will automatically enter the argument list, including their AIMMS prototype, for you. For a DATABASE PROCEDURE 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 STORED PROCEDURE 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 DATABASE PROCEDURE you can indicate the intended use of the procedure.

- When you do not specify the property UseResultSet, AIMMS lets you call the DATABASE PROCEDURE as if it were an AIMMS procedure.
- When you do specify the property UseResultSet, AIMMS lets you use the DATABASE PROCEDURE 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.

```
DATABASE PROCEDURE:
    identifier       : StoreSingleTransport
    data source      : "Topological Data"
    stored procedure : "SP_STORE_SINGLE_TRANSPORT"
    arguments        : (from, to, transport) ;
DATABASE PROCEDURE:
    identifier       : SelectTransportNetwork
    data source      : "Topological Data"
    stored procedure : "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 DATABASE PROCEDURE 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.

```
DATABASE PROCEDURE:
    identifier  : DeleteTableVersion
    data source : "Topological Data"
    arguments   : (DeleteTable, DeleteVersion)
    SQL query   :
        FormatString( "DELETE FROM %s WHERE VersionCol = '%s'",
                      DeleteTable, DeleteVersion ) ;
```

Executing SQL statements directly

In addition to executing SQL queries through DATABASE PROCEDURES, 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 DATABASE PROCEDURE 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.

20.6 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 TestDatabase-Table

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.

20.7 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. The ODBC technology uses a fixed format for each of these data types. Most likely, this format will not coincide with the format that you use to store date 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.

The ODBCDate-TimeFormat parameter

If a date-time column in a database table does not map onto a CALENDAR in your model, you can still convert in 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 format discussed in Section 24.7.

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 '980320', whilst the arrival times should be formatted as '12:30 PM' or '9:01 AM'. Using the time specifiers of Section 24.7, you can accomplish this through the following assignments to the predefined parameter ODBCDateTimeFormat:

```
ODBCDateTimeFormat( 'OrderDates' )  := "%y%m%d";
ODBCDateTimeFormat( 'ArrivalTime' ) := "%h:%M %p";
```

Chapter 21

Format of ASCII Data Files

This chapter

Data provided in ASCII 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 ASCII data files with special emphasis on the two possible tabular formats. Data provided in ASCII files can only be read through the use of the READ statement which is discussed in Chapter 19.2.

21.1 ASCII data files

Allowed ASCII formats

ASCII 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 an ASCII 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 22.2 and 22.3).

Easily generated

The ASCII 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, ASCII 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 ASCII 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 ASCII 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 ASCII 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  } ;
```

21.2 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 ASCII data files and in the `INITIAL DATA` 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 :

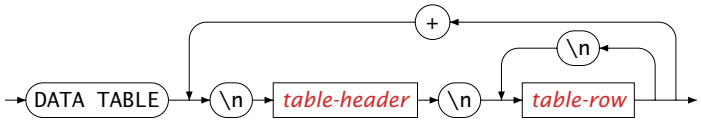
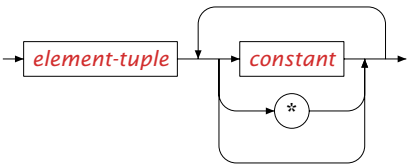


table-header :

table-row :



21.3 Composite tables

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, compound sets, parameters, and variables in a single statement. Composite tables always form a single block, and can only be used in ASCII data files. *Multiple identifiers*

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

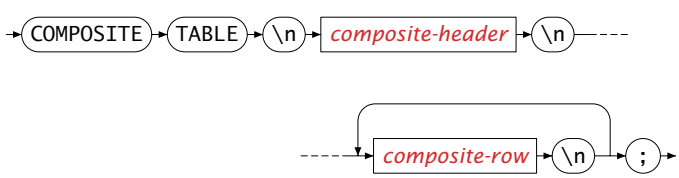
The following statement illustrates a valid example of a composite table. It initializes the compound set `Routes`, as well as the parameters `Distance` and `TransportCost`, all of which are defined over the index space (`i`, `j`).

```
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
;
```

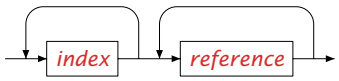
Syntax

The detailed syntax of the composite table is given by the following diagram, where the symbol "\n" stands for the newline character.

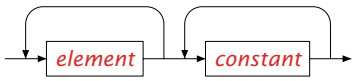
composite-table :



composite-header :



composite-row :



Chapter 22

ASCII 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 *ASCII 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. ASCII 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 ASCII reports. More specifically, it discusses the `FILE` declaration, as well as the `PUT` and `DISPLAY` statements. It also explains how you can optionally create an ASCII 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.

22.1 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 22.1.

Attribute	Value-type	See also page
NAME	*string-expression*	
DEVICE	disk, window, void	
MODE	replace, merge	
TEXT	*string*	19
COMMENT	*comment string*	19
CONVENTION	*convention*	234, 279

Table 22.1: 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", 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:
   identifier : ResultFile
   name       : "result.dat"
   device     : disk
   mode       : replace ;

FILE:
   identifier : ResultWindow
   name       : "Model results"
   device     : window
   mode       : merge ;
```

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 23.6.

22.2 The PUT statement

Customized ASCII reports

AIMMS provides two statements to create a customized ASCII 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 ASCII 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 ASCII 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 22.4.

22.2.1 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*. Until you change the current file again, all output of subsequent `PUT` and `DISPLAY` statements is also sent to it.

Example

The following statements illustrates how to send output to a particular file.

```
PUT ResultFile ;
PUT "The model results are:" ;
DISPLAY Transport ;
```

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.

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 22.4).

Undirected output

When you have not yet selected a current file, AIMMS will send the output of any `PUT` or `DISPLAY` statement to the standard listing file associated with your model.

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.

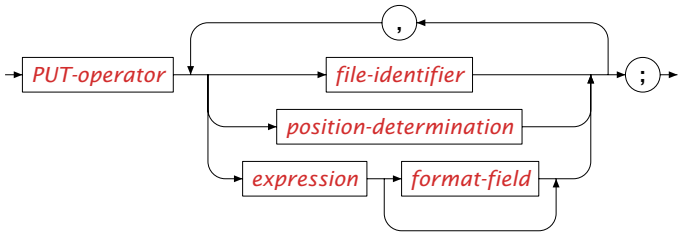
22.2.2 Formatting and positioning PUT items

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 ASCII 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 22.2. 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 22.4.

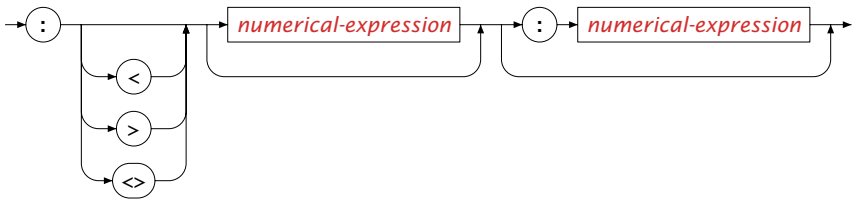
Put items are always scalar

All PUT operators only accept scalar expressions. For each scalar item to be printed you can optionally specify a format field, with syntax:

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 22.2: PUT keywords

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 22.3 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 22.3: Format specification of PUT arguments

Interpretation of precision

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.

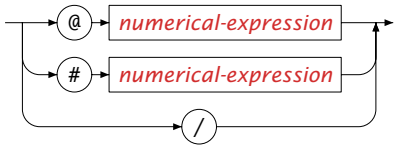
Using the `FormatString` function

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 an ASCII report or window.

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 22.4, 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 22.4: 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.

22.2.3 Extended example

Example

This example builds upon the transport model used throughout the manual. The following group of statements will produce an ASCII 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
```

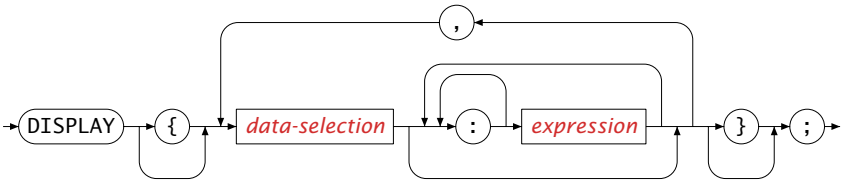
22.3 The DISPLAY statement

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.

Syntax

display-statement :



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 up to four colon-separated numbers. These numbers in sequence represent:

- the number of decimals to be printed for each entry,
- the dimension of the row space,
- the dimension of the column space, and
- the desired numbers of columns per line.

Each of the components can be a numerical expression. You can leave components of the display format empty, or omit the last components if you do not need them. When a component is not specified, AIMMS will use the system default.

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 return with a compiler error, when you try to combine 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: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):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:2:0:2:2;
```

  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:2:0:1:3;
```

  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.

```
   display { Supply:2, Demand:2 };
```

  Execution of this statement results in the creation of the following one-dimensional composite table.

```
Composite table:
   i             Supply  Demand
!  ----------    ------  ------
   Amsterdam      10.00    5.00
   Rotterdam      12.50   10.00
   'Den Haag'      7.50   15.00
   ;
```

22.4 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 22.1 gives an overview of a page in a page-based report.

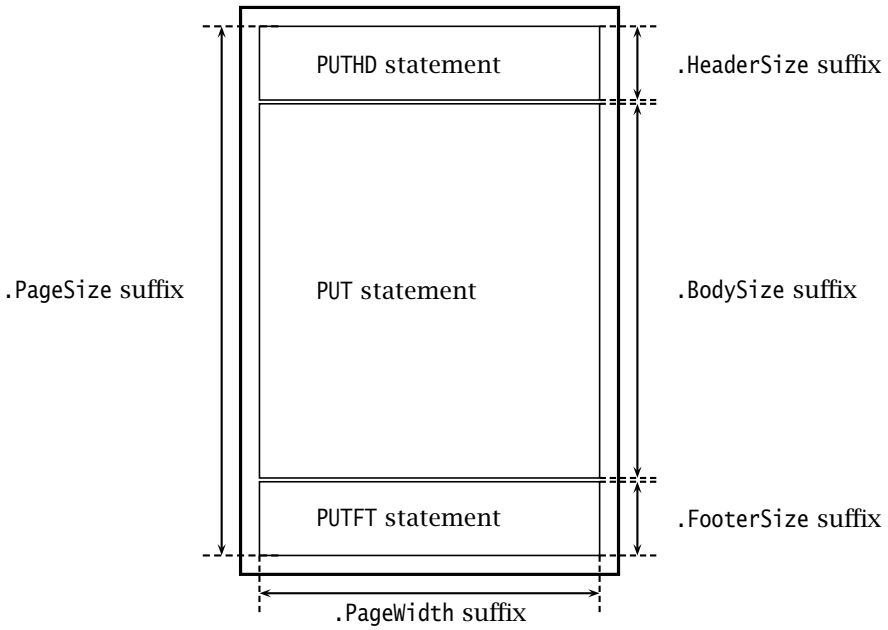


Figure 22.1: Overview of a page in a page based report

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 22.5 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 22.5: Page structure attributes

Positioning in page mode

The positioning operators @, #, and / explained in Section 22.2 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.

22.5 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 ASCII output* produced from `PUT` or `DISPLAY` statements within your model.

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 23

Units of Measurement

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.

This chapter

23.1 Introduction

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.

Units are common

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.

Why units in models

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 base units associated with the following nine basic quantities.

Standard units

Quantity	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 23.1: Basic SI quantities and their base units

Standard derived units

All physical quantities which are not one of the nine basic SI quantities are called *derived* quantities. Each such derived quantity has a base unit which can be expressed in terms of the base units of the basic SI quantities. Optionally, a symbol can be associated with the base unit of a derived quantity, 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 base units. However, many others are available in AIMMS.

Quantity	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 23.2: 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 `1 km = 1000 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 derived from base units using a prefix notation to provide scaling. Table 23.3 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 23.3: 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.

Important overview of terminology

From now on we distinguish *basic* quantities from *derived* quantities. Each quantity must have 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* units, i.e. defined through an expression in terms of other 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 a quantity, you are also allowed to specify one or more *related* unit symbols.

23.2 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 23.4.

Attribute	Value-type	Mandatory
BASE UNIT	[*unit-symbol*] [=] [*unit-expression*]	yes
TEXT	*string*	
CONVERSION	*unit-conversion-list*	
COMMENT	*comment string*	

Table 23.4: Quantity attributes

The BASE UNIT 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 set equal to a unit expression.

The number 1 is a unit symbol which allows you to express units in terms of, for instance, percentages.

Example

The following example illustrates the three types of base units.

```
QUANTITY:
   identifier : Length
   base unit  : m ;                    ! atomic unit
QUANTITY:
   identifier : Time
   base unit  : s ;                    ! atomic unit
QUANTITY:
   identifier : Velocity
   base unit  : m/s ;                  ! unit expression
QUANTITY:
   identifier : Frequency
   base unit  : Hz = 1/s ;             ! compound unit
```

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 basic

The previous example strictly adheres to the SI standards, and, for example, defines the base unit of 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.

Unit expressions

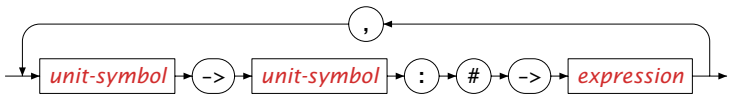
Unit expressions can only consist of unit symbols (of both base and related units), constant factors, the two operators * and /, parentheses, and the power operator ˆ with integer exponent. The order of evaluation is from left to right. For instance, let a, b and c be unit symbols, then the unit definition a/b/c is equal to (a/b)/c, and can also be written as a/(b*c).

The CONVERSION attribute

With the CONVERSION attribute you can declare and define one or more related units by specifying the (linear) transformation to the associated base unit. You may also specify the inverse transformation. The corresponding inverse transformation of the transformation that you specified, which is needed to present output data, is automatically determined by AIMMS. The conversion syntax follows.

Syntax

unit-conversion-list :



Conversion expression

A single conversion must be defined using a linear expression of the form $(a \cdot \# + b)$ where # is a special token. 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:
   identifier  : Length
   base unit   : m
   conversions : km   -> m    :  # -> # * 1000 ,
                 mile -> m    :  # -> # * 1609 ;
QUANTITY:
   identifier  : Temperature
   base unit   : degC
   conversions : degC -> degF :  # -> # * 1.8 + 32 ;
QUANTITY:
   identifier  : Energy
   base unit   : J = kg * mˆ2 / sˆ2
   conversions : kJ  -> J     :  # -> # * 1000  ,
                 MJ  -> J     :  # -> # * 1.0e6 ,
                 kWh -> J     :  # -> # * 3.6e6 ;
QUANTITY:
   identifier  : Currency
   base unit   : US$
   conversion  : DM  -> US$   : # -> # * ExchangeRate("DM")   ,
                 DFl -> US$   : # -> # * ExchangeRate("DFl") ;
QUANTITY
   identifier  : Unitless
   base unit   : 1
   conversions : % -> 1      : # -> # / 100 ;
```

23.3 Unit analysis and automatic scaling

Use of units

You can use quantities and their associated units declared in your model for various purposes:

- automatic checking of the statements in your model for unit consistency,
- automatic scaling of identifiers in assignments and display statements, and
- automatic scaling of the variables and constraints in a mathematical program.

The UNIT attribute

To assign units in your model, you can attach a unit definition to identifiers throughout your model text using their UNIT attribute. In its simplest form, a unit definition 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 unit definition may be enclosed in square braces, and can contain an additional numeric scale factor. 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.

Unit consistency

A unique unit expression in terms of atomic units can be associated with every derived unit or compound unit symbol. *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.* AIMMS will verify that the atomic unit expression for every term in either an assignment statement or a constraint is identical except for numeric scaling factors. 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 atomic unit. 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 solely, or consists of any of AIMMS' other nonscalar constant data expression (all preceded by the keyword DATA), AIMMS will assume by default that such a numerical constant or collection of numerical constants are expressed in the unit specified for 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, as explained below.

Constant terms in expressions

On the other hand, if a nonconstant expression contains a constant term, then AIMMS will make no assumption about the intended unit of the constant term. If a unit inconsistency occurs for that reason, you should explicitly add a unit to the constant term to re-solve the inconsistency.

Automatic unit checking on or off

There are facilities in AIMMS to perform automatic unit analysis of all assignment statements, parameter definitions, constraint definitions and variable definitions. 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 latter is the default). You can find the full details on all unit-related options in the help file that comes with your AIMMS system.

Scale consistency

The assignment in the previous example 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, the previous assignment is 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 specified for these identifiers.

Example of units . . .

This example illustrates the definition of units for identifiers. It is based on the SI units for weight, velocity and energy, and uses the derived units `ton`, `km`, `h` and `MJ`.

```
VARIABLE:
   identifier   : WeightOfItem
   index domain : i
   unit         : ton ;
VARIABLE:
   identifier   : VelocityOfItem
   index domain : i
   unit         : Velocity: km/h ;
VARIABLE:
   identifier   : KineticEnergyOfItem
   index domain : 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 atomic units associated with weight, velocity and energy.

... and unit consistency

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.

Specifying units in expressions

The specification of units is not restricted to the `UNIT` attribute alone. You can also specify units inside expressions. This is especially useful when there are numeric constants inside your expressions, or when you want to indicate that entire terms should be associated with a specific unit. When the specified unit does not equal the atomic unit associated with the expression, AIMMS will automatically add the appropriate scale equation to make the term commensurate with the atomic unit.

Example

Consider the following two examples in which units are placed inside an expression in order to obtain unit consistency.

- Let `a := b + 1` be an assignment statement where both `a` and `b` are measured in terms of length. As explained above, AIMMS will make no assumption about the unit associated with the numerical constant `1` in the expression on the right-hand side of the assignment. In order to satisfy the unit consistency check inside AIMMS, you need to explicitly add a unit description to the constant term inside the assignment statement, e.g. `a := b + 1 [km]`. If the associated base unit is `[m]`, AIMMS will evaluate the assignment numerically to `a := b + 1*1000`.
- Let `a := b * c` where `a` is measured in terms of length, but both `b` and `c` do not have an associated unit. If the interpretation of the value of `b * c` is also in terms of length, say `[km]`, then you must write `a := (b * c) [km]` to obtain unit consistency. If the base unit is `[m]`, AIMMS will evaluate the assignment numerically to `a := (b * c)*1000`.

Units and intrinsic functions

When you use a function inside an expression in your model, the unit relationship between the arguments and the result of the function falls into one of the following categories.

- *Unitless* functions, for which the arguments and 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 nonconstant 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 exponent.

23.4 Units and data exchange

Scaled versus unscaled values

With each identifier for which you have specified a `UNIT` attribute, AIMMS can associate two values: the *scaled* value and the *unscaled* value (i.e. expressed in base units). The transformation between scaled and nonscaled values is completely determined by the product of explicit and implicit scale factors associated with a particular unit expression. Whereas explicit scale factors are explicitly part of the unit expression, implicit scale factors are the result of expressing the `UNIT` attribute in nonbase units.

When used

As mentioned before, AIMMS uses unscaled values for all internal storage and during all internal arithmetic computations. This guarantees automatic scale consistency. However, the use of scaled values is more natural when exchanging data with components that are external to 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,
- when passing arguments to external procedures and functions, and
- when 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. That is, if you have specified `kton` as the unit attribute of an identifier, while the underlying base 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 (all 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 unscaled values for internal storage.

Automatic scaling for solvers

During communications with a solver, AIMMS will scale all variables in accordance with the scale factor associated with the UNIT attribute in their declaration. In addition, AIMMS will scale each constraint (including variable definitions) in accordance with its specified unit. This 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 therefore neither be very large nor very small. In general, nonlinear solvers benefit from such scaling.

Main example revisited

In the main example of the previous section, 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:

```
ScaledKineticEnergyOfItem(i) =
    1/2 * (1/10^3) * ScaledWeightofItem(i)
        * ((1/3.6) * ScaledVelocityOfItem(i))^2 ;
```

Notice that each variable shown in this expression has been prefixed with the string "Scaled" to indicate that the values will be passed to the solver in scaled units instead of the values in base units as they are stored internally by AIMMS. These scaled values are the same values that you would see if AIMMS displayed the identifier in a report or the graphical user interface without a unit convention.

23.5 Locally overriding units

Locally overriding units

While exchanging identifier data between AIMMS and external components such as external data sources and the various solvers provided through AIMMS, there are two ways to override the default unit locally.

- You can override the default unit by associating a different unit to *entire* identifiers on the left side of a data assignment, in the header of a composite table, or in the READ, WRITE and DISPLAY statements.
- In a PUT statement you can override the default at the element level by associating a different unit to particular scalar PUT items.

Example

Given the declarations of the examples in the previous sections, the following statement locally override the default unit [km/h] with the 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];
```

Note that parentheses are always required when you want to override the default unit in expressions and statements.

Override for consistency

In addition to overriding units during a data exchange, which results in scaling of the transferred data, you can also override the unit of a (sub)expression in an assignment for the purpose of enforcing unit consistency of all terms in the assignment. The following example illustrates such a consistency override.

```
SoundIntensity := (10 * log10( SoundLevel / ReferenceLevel )) [dB]
```

When a local override is used in an assignment statement with the intention of enforcing unit consistency, AIMMS will interpret the value of the expression as a number in the specified unit. As a result, AIMMS will scale it to corresponding atomic unit expression in which all data is stored internally. *Note that, when the overridden expression is not unitless, the resulting value may be unexpected, because* AIMMS *performs all internal computation in atomic units.*

23.6 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 unit convention offers a global medium to specify alternative (scaled) units for multiple quantities, units, and identifiers.

Effect of unit conventions

Once you have selected a unit 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 compound quantities not present in a convention, AIMMS will use the unit as specified in the declaration of the identifier after application of the selected convention on a unit's atomic components.

Convention attributes

Unit conventions must be declared before their use. The list of attributes of a CONVENTION declaration are described in Table 23.5.

Convention list

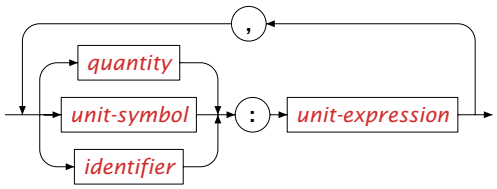
A convention list is a simple list associating single quantities, units and identifiers with a particular (scaled) unit expression. The specified unit expressions must be consistent with the base unit of the quantity, the specified unit, or the identifier unit, respectively.

Attribute	Value-type
TEXT	*string*
COMMENT	*comment string*
PER IDENTIFIER	*convention-list*
PER QUANTITY	*convention-list*
PER UNIT	*convention-list*

Table 23.5: Simple CONVENTION attributes

Syntax

convention-list :



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:
    identifier     : AngloAmericanUnits
    per identifier : GasolinePurchase : gallon,
                     PersonalHeight   : feet
    per quantity   : Velocity    : mph,
                     Temperature : degF,
                     Length      : mile
    per unit       : cm  : inch,
                     m   : yard,
                     km  : mile
```

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 use the unit that is the result from applying any applicable unit conversions to the atomic components of the UNIT attribute of the identifier.

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:

- the main model,
- a mathematical program,

- a file, 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, 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 illustrates the use of the CONVENTION attribute. All the output to the file ResultFile will be displayed in Anglo-American units.

```
FILE:
    identifier : ResultFile
    name       : "result.dat"
    convention : AngloAmericanUnits ;
```

23.7 Unit-valued parameters

Unequal units

In some cases not all entries of an indexed identifier have the same 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.

Unit analysis

In the unit analysis of a model with indexed, parametrized units, AIMMS will perform its (compile-time) unit analysis only on the basis of the symbolic unit-valued parameter names. This will automatically ensure that there is also unit consistency at the individual level, regardless of the specific individual unit values specified as data.

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.

```
UNIT PARAMETER:
   identifier   : NutrientUnit
   index domain : n ;
UNIT PARAMETER:
   identifier   : FoodUnit
   index domain : f ;
```

With these unit-valued parameters you can specify meaningful indexed unit expressions for the UNIT attribute of the following parameters.

```
PARAMETER:
   identifier   : NutritiveValue
   index domain : (f,n)
   unit         : NutrientUnit(n)/FoodUnit(f)  ;
PARAMETER:
   identifier   : NutrientMinimum
   index domain : n
   unit         : NutrientUnit(n) ;
VARIABLE:
   identifier   : Serving
   index domain : 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:
   identifer    : NutrientRequirement
   index domain : n
   unit         : NutrientUnit(n)
   definition   :
         sum[ f, Servings(f)*NutritiveValue(f,n) ] >=  NutrientMinimum(n) ;
```

The appearance of the parameter NutrientUnit(n) in the unit definition of the constraint will cause AIMMS to scale the individual constraints according to the scaling factors of the individual units specified in this parameter.

Indexed scaling

In addition to unit analysis, AIMMS can also use parametrized units for unit scaling, with and without the use of CONVENTIONS. Contrary to unit analysis, AIMMS considers the specified units at the individual (indexed) level during any data exchange with an external component (see Section 23.4), and will determine the proper scaling and unit convention for every individual index position. As usual, AIMMS will internally store the data for every index value in the corresponding atomic unit.

Conventions and indexed units

When displaying or reading data defined over an indexed unit, AIMMS will scale all individual entries according to the active unit convention as applied to the corresponding entries in the indexed unit.

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. The value of the individual entries must be valid unit expressions. For compound units 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]  };
```

Fixed versus parametrized units

Since all data is internally stored by AIMMS with respect atomic units only, you can easily make assignments from identifiers with a fixed unit to identifier slices of identifiers with a parametrized unit and vice versa, *as long as you have made sure that the corresponding units are commensurable*. However, as AIMMS considers the name of a unit parameter as a (pseudo-)atomic unit for the purpose of unit analysis, you may want to add a unit consistency override to avoid a unit consistency warning or error during compilation of the statement. The following example illustrates this.

```
NutritiveValue(f,'Energy') := (EnergyContent(f)) [NutrientUnit];
```

In this assignment, it is assumed that the unit of `EnergyContent(f)` is `kJ` or any other commensurable unit.

Chapter 24

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.

24.1 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 24.1 illustrates an example of a timetable relating a horizon and a calendar.

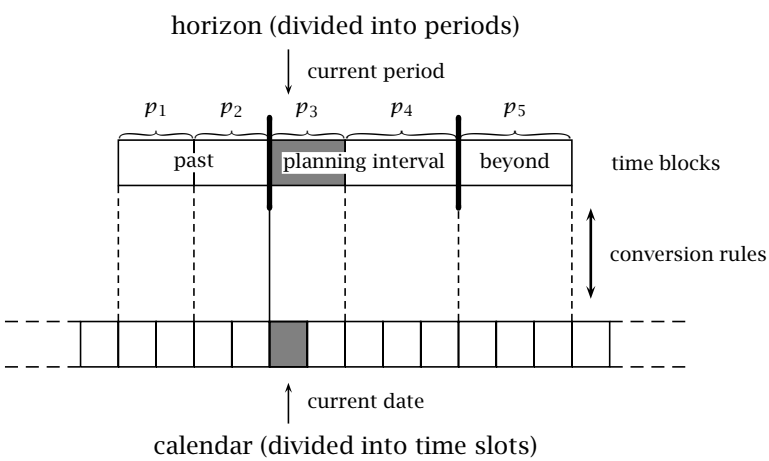


Figure 24.1: Timetable relating calendar and horizon

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.

24.2 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 24.1. 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
BEGIN DATE	*string*		yes
END DATE	*string*		yes
UNIT	*unit*		yes
TIMESLOT FORMAT	*string*		yes
INDEX	*identifier-list*	31	yes
PARAMETER	*identifier-list*	31	
TEXT	*string*	19	
COMMENT	*comment string*	19	

Table 24.1: 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.

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 23). 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.

Begin and end date

The mandatory BEGIN DATE and END DATE attributes of a calendar specify its range. AIMMS will generate all time slots of the specified length, whose *begin time* lies between the specified BEGIN and END DATE. As a consequence, the *end time* of the last time slot may be after the specified END DATE. An example of this behavior occurs, for instance, when the requested length of all time slots is 3 days and the END DATE does not lie on a 3-day boundary from the BEGIN DATE. 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 BEGIN DATE or the END DATE 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. For instance, a calendar expressed in hours may have a BEGIN DATE 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.

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 TIMESLOT FORMAT is presented in Section 24.7.

Example

The following example is a declaration of a daily calendar and a monthly calendar

```
CALENDAR:
   identifier       : DailyCalendar
   index            : d
   element          : CurrentDay
   text             : "A work-week calendar for production planning"
   begin date       : "1996-01-01"
   end date         : "1997-06-30"
   unit             : day
   timeslot format  : "%d/%m/%y" ;      ! format explained later

CALENDAR:
   identifier       : MonthlyCalendar
   index            : m
   begin date       : CalendarBeginMonth
   end date         : CalendarEndMonth
   unit             : month
   timeslot format  : "%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 BEGIN and END DATE 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.

24.3 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 24.2.

Attribute	Value-type	See also page	Mandatory
SUBSET OF	*subset-domain*	31	
INDEX	*identifier-list*	31	yes
PARAMETER	*identifier-list*	31	
TEXT	*string*	19	
COMMENT	*comment string*	19	
DEFINITION	*set-expression*	33	yes
CURRENT PERIOD	*element*		yes
INTERVAL LENGTH	*integer-reference*		

Table 24.2: Horizon attributes

Horizon attributes explained

The CURRENT PERIOD 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 INTERVAL LENGTH 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.

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:
   identifier     : ModelPeriods
   index          : h
   parameter      : IntervalStart
   current period : 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:
   identifier     : Stock
   index domain   : 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 5.2.4) 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.

24.4 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 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.

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 non-zero 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 24.2 and the horizon of Section 24.3, 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 periods `p-01` ... `p-05`, and zero otherwise,
- a parameter `LengthDominates(h)` assuming the value 1 for 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 );
```

If all weekend days are inactive, and CurrentDay equals '24/01/96' (a Wednesday), then TimeTable describes the following mapping.

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)

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*)

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 24.3.

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	

Table 24.3: 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 24.2, 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.

24.5 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 24.1, 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 24.4 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 24.4: 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 24.5 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	0	1	2	3	5	7	9	9	9
1	0	0	0	1	2	3	5	7	9
0.5	0	0	1	2	3	5	7	9	9

Table 24.5: Conversion rules for interpolated data

Example

Consider the calendar `DailyCalendar`, the horizon `ModelPeriods` and the timetable `TimeTable` declared in Sections 24.2, 24.3 and 24.4, 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)) );
```

24.6 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 24.2, 24.3 and 24.4, 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`.

The outline of the simple strategy can be implemented as follows. *Code outline*

```
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
```

The simple strategy will not work *Generic strategy*

- 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.

The generic strategy for implementing a rolling horizon is outlined as follows. *Algorithm outline*

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.

The outline of the generic strategy can be implemented as follows. *Code outline*

```
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;
```

24.7 Format of time slots and periods

While the BEGIN and END DATE attributes have to be specified using the fixed *reference date* format (see Section 24.2), AIMMS provides much more flexible formatting capabilities to describe *Flexible time slot and period formats*

- 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.

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. *Wizard support*

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 `TIMESLOT FORMAT` 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 `%%`.

24.7.1 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 24.6.

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.

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 24.6: Conversion specifiers for date components

Week year and century

The %Y and %C specifiers refer to the weekyear and weekcentury values of a specific date, as explained on page 295. 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 END DATE attributes of a calendar can be reproduced using the format "%c%y-%m-%d".

24.7.2 Time-specific components

Time-specific components

The conversion specifiers for time components are listed in Table 24.7. 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 24.7: 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`".

24.7.3 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 24.8), 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 24.8.

Conversion specifier	Meaning
`%B`	begin of unit period
`%I`	end of period (inclusive)
`%i`	end of period (inclusive), but omitted when equal to begin of period
`%E`	end of period (exclusive)

Table 24.8: 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 24.8), 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.

24.8 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. The function `CurrentToTimeSlot` returns the time slot in *calendar* containing the current moment.

24.9 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 24.7.

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 24.2. 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*. The result of `StringToMoment` is the elapsed time in *unit* between *reference-date* and *moment-string*.

Example

```
moment := MomentToString("%c%y-%Am|months|-%d (%sAw|weekdays|) %H:%M",
                         [hour], "1996-01-01 14:00", 2.2 );
          ! result : "1996-Jan-01 (Monday) 16:12"

elapsed := StringToMoment("%c%y-%Am|months|-%d (%sAw|weekdays|) %H:%M",
                          [hour], "1996-01-01 14:00", "1996-Jan-01 (Monday) 16:12" );
          ! result : 2.2
```

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*).

Chapter 25

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.

25.1 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 `EXTERNAL PROCEDURE` or `FUNCTION`.
- 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 25.1 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 procedure and function calls within your model, while all right-to-left arrows are provided for by the AIMMS API.

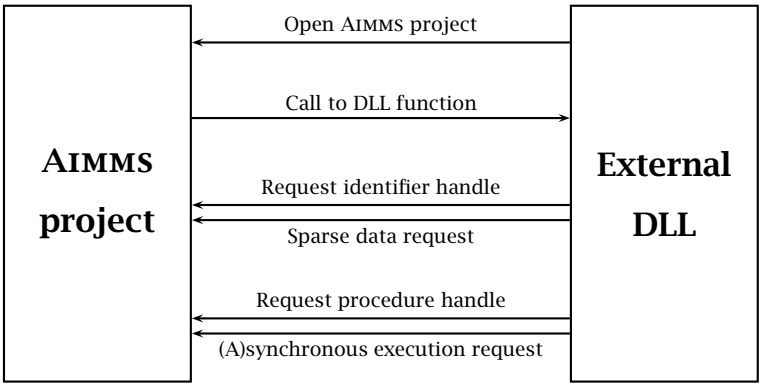


Figure 25.1: Interaction between AIMMS and an external DLL

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 EXTERNAL PROCEDURE 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 EXTERNAL PROCEDURE declaration.

```
EXTERNAL PROCEDURE:
    identifier : PrintParameterInfo
    arguments  : (param)
    DLL name   : "Userfunc.dll"
    body call  : 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.

```
ELEMENT PARAMETER:
   identifier  : 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.

aimmsapi.lib import library

The AIMMS API functions are provided in the form of a Visual C/C++ import library aimmsapi.lib to the libaimms.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 how to import the functions in libaimms.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 25.7).

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.

25.2 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 25.1 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 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 AimmsAttributeFlags(int handle, int *flags)`
`int AimmsAttributeElementRange(int handle, int *sethandle)`
`int AimmsAttributeCompoundDimension(int handle, int *dim)`
`int AimmsAttributeCompoundDomain(int handle, int *domain)`

Table 25.1: 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 (i.e. a compound set not used for indexing),
- compound root set,
- compound subset,
- indexed set,
- numeric parameter,
- element parameter,
- string parameter,
- unit parameter,
- 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 the 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.

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 25.4, 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 (simple or compound) 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 a *compound set* is 1,
- 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 25.4.

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 25.3).

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 25.3). 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 25.4) to verify whether a particular element satisfies its domain restriction.

Example

Consider the following set and parameter declarations.

```
SET:
    identifier   : S_0
    index        : i_0 ;
SET:
    identifier   : S_1
    subset of    : S_0
    index        : i_1, j_1 ;
SET:
    identifier   : S_2
    subset of    : S_1
    index        : i_2 ;
PARAMETER:
    identifier   : p
    index domain : i_0 ;
PARAMETER:
    identifier   : q
    index domain : (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.

Ordered, special, raw and read-only flags

By specifying the input-output type and the ordered, retainspecials 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 AimmsAttributeFlags you obtain the active set of flags indicating whether the data associated with a handle is passed ordered, whether special values are passed unchanged, whether inactive data is passed, and whether you can make assignments to the handle. The result is the *bitwise*

or function of the individual flag values as defined in the `aimmsapi.h` header file.

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 25.4.

Compound dimension and domain

As discussed above, AIMMS considers a compound set as a 1-dimensional domain for the communication with external DLLs. You can use the functions `AimmsAttributeCompoundDimension` and `AimmsAttributeCompoundDomain` to retrieve the dimension of the compound set as a relation, as well as handles to the associated global domain sets. You can use these to translate the compound element numbers into tuples of element names or ordinal numbers in the respective domain sets using the AIMMS API functions discussed in Section 25.5.

25.3 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 18.3) can be called from within external DLLs. Table 25.2 lists all available AIMMS API functions for creating handles and performing data control operations.

```
int AimmsIdentifierHandleCreate(char *name, int *domain, int *slicing,
                                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 25.2: AIMMS API functions for handle management

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.

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 25.2 and 25.4).

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.

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.

25.4 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 25.3 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 25.3: 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 315. 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.

Converting element numbers

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 local domain, you must be aware that the latter representation is not invariant under changes in the contents of the local domain. If you are interested in the ordinal numbers with respect to a particular set at a particular moment or if you need the string representation of a particular element, you can convert the returned element numbers into these formats using the AIMMS API functions discussed in Section 25.5.

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 25.5 to obtain the associated ordinal numbers or string representations.

Set values

For sets (either simple, relation, compound 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 compound sets the `tuple` argument contains the compound element number which can be converted into its corresponding tuple representation using the function `AimmsSetCompoundToTuple` (see also Section 25.5). For indexed sets, AIMMS returns tuples for which the last component is the (simple or compound) 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 25.2). 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 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.

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. Similarly, if you want to add nonexisting compound elements to a compound subset, you must first add such elements to the corresponding compound root set using the function AimmsSetAddTupleToCompound.

25.5 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 and compound sets, and let you convert element numbers into ordinal numbers and element names, or vice versa. Table 25.4 presents all set related API functions.

`int AimmsSetAddElement(int handle, char *name, int *element)`
`int AimmsSetAddElementRecursive(int handle, char *name, int *element)`
`int AimmsSetRenameElement(int handle, int element, char *name)`
`int AimmsSetDeleteElement(int handle, int element)`
`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)`
`int AimmsSetCompoundToTuple(int handle, int compound, int *tuple)`
`int AimmsSetTupleToCompound(int handle, int *tuple, int *compound)`
`int AimmsSetAddTupleToCompound(int handle, int *tuple, int *compound)`
`int AimmsSetAddTupleToCompoundRecursive(int handle, int *tuple, int *compound)`

Table 25.4: AIMMS API functions for passing set data

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. To add a new element tuple to a compound root set, you should use the function `AimmsSetAddTupleToCompound` discussed below.

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 either a simple or compound set. If the set is a simple or compound *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 25.3).

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.

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 25.3).

Compound elements

AIMMS represents the elements of a compound set by element numbers in a compound *root* set. With the functions

- `AimmsSetCompoundToTuple`, and
- `AimmsSetTupleToCompound`

you can obtain the tuple of element numbers in the corresponding simple root sets for a compound element number, and vice versa.

Adding and deleting compound elements

If a tuple does not correspond to an existing compound element, the function `AimmsSetTupleToCompound` will fail. In such a case, you can create a new compound element by calling the function `AimmsSetAddTupleToCompound` or `AimmsSetAddTupleToCompoundRecursive` instead, depending on whether you want to add the tuple to a compound root or subset, respectively. You can delete compound elements from a compound set using the function `AimmsSetDeleteElement`.

25.6 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 25.5 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 AimmsProcedureAsyncRunCreate(int handle, int *argtype,
                                 AimmsValue *arglist, int *request)
int AimmsProcedureAsyncRunDelete(int request)
int AimmsProcedureAsyncRunStatus(int request, int *status, int *result)
```

Table 25.5: 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 25.2) 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 124 and 134) 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 25.9 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.

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.

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.

25.7 Passing errors and messages

Passing errors and messages

The AIMMS API functions in Table 25.6 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 25.6: 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.

25.8 Opening and closing a project

Opening and closing a project

The AIMMS API functions in Table 25.7 allow you to open and close an AIMMS project from within your own application.

`int AimmsProjectOpen(char *commandline, int *handle)`
`int AimmsProjectClose(int handle, int interactive)`
`int AimmsProjectWindow(HWND *window)`

Table 25.7: 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.

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 `AimmsProjectOpen` 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.

25.9 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 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 25.8 to obtain exclusive control over the AIMMS execution engine.

```
int AimmsControlGet(int timeout)
int AimmsControlRelease(void)
```

Table 25.8: 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.

Appendices

Appendix A

Description of available distributions and statistical operators

This chapter

This chapter provides a more elaborate description of the distributions and statistical operators listed in Tables 6.5 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,
- its shape, and
- its typical use in applications.

Such information may be useful in the selection of a distribution to describe the particular statistical behavior of input data of experiments that you want to perform on top of your model.

Description of statistical operators

For each of the available statistical operators we provide

- the interpretation of its result, and
- the formula for the computation of the operator.

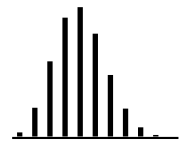
Such information may be useful when you want to perform an analysis of the results of your experiments.

A.1 Discrete distributions

We start our discussion with the discrete distributions available in AIMMS. They are

- the `Binomial` distribution,
- the `HyperGeometric` distribution,
- the `Geometric` distribution,
- the `Negative Binomial` distribution, and
- the `Poisson` distribution.

`Binomial` distribution

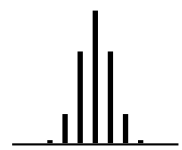


The `Binomial` distribution describes the number of times that a particular outcome (referred to as "success") occurs in a fixed number of tries. The assumption is that only two outcomes are possible, and that the likelihood of a particular outcome remains the same from try to try.

- **Input parameters** : Number of tries n and probability of success p
- **Input check** : $n > 0$ and $0 < p < 1$
- **Permitted values** : $\{i \mid i = 0, 1, \ldots, n\}$
- **Formula** : $P(X = i) = \binom{n}{i} p^i (1-p)^{n-i}$

Whenever there is a situation of two outcomes and independent tries you should consider the `Binomial` distribution. A typical example is the number of defectives in a batch of manufactured products where a fixed percentage were found to be defective in previously produced batches. Another example is the number of persons in a group voting yes instead of no, when the probability of yes has been determined on the basis of a sample.

`HyperGeometric` distribution

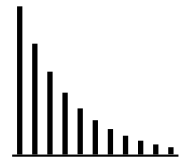


The `Hypergeometric` distribution is like the `Binomial` distribution in that it also describes the number of times that a particular outcome (a "success") occurs in a fixed number of tries. Again, only two outcomes are possible, but the likelihood of a particular outcome changes with each "successful" outcome. The reason for this is that the same successful outcome cannot occur again. This is also referred to as "sampling without replacement" indicating that something is removed from the sample. By changing the sample that you draw from, you in effect influence the probability of success for the remaining portion of the sample.

- **Input parameters** : Number of tries n, known initial probability of success p, and population size N
- **Input check** : $N > 0, 0 < n \leq N$, and $p = \frac{1}{N}, \frac{2}{N}, \ldots, \frac{N-1}{N}$
- **Permitted values** : $\{i \mid i = 0, 1, \ldots, n\}$
- **Formula** : $P(X = i) = \frac{\binom{Np}{i}\binom{N(1-p)}{n-i}}{\binom{N}{n}}$

In the `Hypergeometric` distribution there is always a finite number of elements (referred to as a population) and a sample size representing a portion of the population. In addition, there is an initial probability of success. Consider, for instance, a total production of 1000 books and the reassurance of the printer that usually 5% of the books may be faulty. Assume that you select a few boxes containing 50 books all together. Whenever you find a faulty book, you remove it from the boxes. The distribution shows the probability of observing i ($i = 0, 1, \ldots, n$) number of faulty books.

`Geometric` distribution

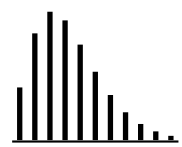


The `Geometric` distribution describes the number of tries until a first success, where the probability of success is the same for each try. The assumption is that there is no a priori limit on the number of tries.

- **Input parameters** : Probability of success p
- **Input check** : $0 < p < 1$
- **Permitted values** : $\{i \mid i = 0, 1, \dots\}$
- **Formula** : $P(X = i) = (1 - p)^i p$

Whenever there is a repetition of the same activity and you are interested in observing a particular outcome, then the `Geometric` distribution might be applicable. A typical situation is going from door-to-door until you make a sale, where the probability of making a sale has been determined on the basis of previous experience. 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.

`Negative Binomial` distribution

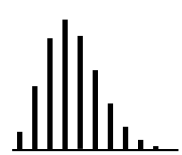


The `Negative Binomial` distribution is very similar to the `Geometric` distribution. Instead of describing the number of tries until a first success occurs, it describes the number of tries until the r-th success occurs. The probability of success is the same for each try. The assumption is that there is no a priori limit on the number of trials.

- **Input parameters** : Success probability p and number of successes r
- **Input check** : $0 < p < 1$ and $r = 1, 2, \dots$
- **Permitted values** : $\{i \mid i = 0, 1, \dots\}$
- **Formula** : $P(X = i) = \binom{r + i - 1}{i} p^r (1 - p)^i$

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. Typical examples are similar to the ones described under the `Geometric` distribution. For instance, in the door-to-door example you are interested in making the r-th sale instead of the first sale.

`Poisson` distribution



The `Poisson` distribution describes the number of times that an event occurs in a given unit of measurement (an interval, a location, etc.). The number of occurrences is not a priori limited to a fixed number, and their mean value is constant for each unit of measurement. The occurrences themselves are independent.

- **Input parameters** : Average number of occurrences λ
- **Input check** : $\lambda > 0$
- **Permitted values** : $\{i \mid i = 0, 1, \dots\}$
- **Formula** : $P(X = i) = \frac{\lambda^i}{i!} e^{-\lambda}$

Recognizing the unit of measurement is a first step in deciding whether the `Poisson` distribution is applicable. Typical examples are the number of visitors in a day over several days, the number of errors per page in a document, the number of defects per batch in a large number of batches, the number of telephone calls per minute during a particular time interval, etc.

A.2 Continuous distributions

In this section we discuss the set of continuous distributions available in AIMMS. They are

- the `Uniform` distribution,
- the `Triangular` distribution,
- the `Normal` distribution,
- the `LogNormal` distribution,
- the `Exponential` distribution,
- the `Gamma` distribution,
- the `Weibull` distribution,
- the `Beta` distribution,
- the `Logistic` distribution,
- the `Pareto` distribution, and
- the `Extreme Value` distribution.

`Uniform` distribution

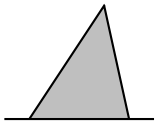
In the `Uniform` distribution all values of the random variable occur between a fixed minimum and a fixed maximum with equal likelihood.

- **Input parameters** : *Min, Max*
- **Input check** : $Min < Max$
- **Permitted values** : $\{x \mid Min \leq x \leq Max\}$
- **Formula** : $f(x) = \dfrac{1}{Max - Min}$

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

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.



- **Input parameters** : *Min, Likeliest, Max* *(Mi, Li, Ma)*
- **Input check** : $Min < Likeliest < Max$
- **Permitted values** : $\{x \mid Min \leq x \leq Max\}$

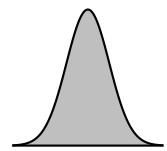
- **Formula** : $h\left(\frac{x - Mi}{Li - Mi}\right) \qquad Mi \le x \le Li$

  $h\left(1 + \frac{x - Li}{Li - Ma}\right) \qquad Li < x \le Ma$

  where the height $h = \frac{2}{Max - Min}$.

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.

`Normal` distribution



In the `Normal` distribution the mean value of the random variable is the most likely. In addition, the random variable is more likely to be near the mean value than far away, and its value could as likely be above the mean as below.

- **Input parameters** : Mean μ and standard deviation σ
- **Input check** : $\sigma > 0$
- **Permitted values** : $\{x \mid -\infty < x < \infty\}$
- **Formula** : $f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$

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 the values in these examples are naturally bounded, while any normally distributed random variable can take on any value between $-\infty$ and ∞. This is not a reason to avoid using the normal distribution, because the likelihood of extreme values away from the mean is essentially zero. A close fit between data values of limited range and normally distributed values is quite common in practice.

`LogNormal` distribution

Whenever an uncertain variable must be nonnegative and most of its values lie near the minimum value, you should consider the `Lognormal` distribution as its distribution. The assumption is that you know both a most likely value and the variation around this most likely value. This distribution gets its name from the fact that any lognormally distributed variable X is such that its logarithm is normally distributed.

- **Input parameters** : Mean μ and standard deviation σ
- **Input check** : $\mu > 0$ and $\sigma > 0$
- **Permitted values** : $\{x \mid 0 < x < \infty\}$
- **Formula** : $f(x) = \frac{1}{\sqrt{2\pi}x\sigma} e^{\frac{-(\ln(x)-\mu)^2}{2\sigma^2}}$

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 you can always estimate a standard deviation term on the basis of historical data.

`Exponential` distribution

Assume that you are observing a sequence of events occurring in time in accordance with the `Poisson` distribution with λ being the number of occurrences per time unit. The `Exponential` distribution gives answer to the question: how long a time do you need to wait until you observe the first occurrence of an event. As stated explicitly for the `Poisson` distribution, the events are independent.

- **Input parameters** : Average number of occurrences λ
- **Input check** : $\lambda > 0$
- **Permitted values** : $\{x \mid 0 \leq x < \infty\}$
- **Formula** : $f(x) = \lambda e^{-\lambda x}$

Whenever you are interested in time between successes or time between failures, you should consider the `Exponential` distribution. 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` distribution is in some sense a generalization of the `Exponential` distribution, just as the `Negative Binomial` distribution is a generalization of the `Geometric` distribution. The `Gamma` distribution gives answer to the question: how long a time do you need to wait until you observe the r-th occurrence of an event (instead of the first occurrence as in the `Exponential` distribution).

- **Input parameters** : Location α and shape β
- **Input check** : $\alpha > 0$ and $\beta > 0$
- **Permitted values** : $\{x \mid 0 < x < \infty\}$
- **Formula** : $f(x) = \dfrac{\alpha^\beta}{\Gamma(\beta)} x^{\beta-1} e^{-\alpha x}$

  where $\Gamma(\beta)$ is the Gamma function

Whenever you are interested in time until the r-th success or time until the r-th failure, you should consider the `Gamma` distribution. Typical examples are the ones you also find for the `Geometric` distribution: time until the r-th failure of equipment, or time until the r-th arrival of customers at a service desk (bank, hospital, etc.).

`Weibull` distribution

The `Weibull` distribution is not easy to characterize. It is a flexible distribution in that particular settings of its input parameters can lead to very good approximations of other distributions. For instance, when the shape input value is 1.0, the `Weibull` distribution is exactly the same as the `Exponential` distribution.

- **Input parameters** : Location l, shape β and scale s

- **Input check** : $\beta > 0$ and $s > 0$
- **Permitted values** : $\{x \mid l \leq x < \infty\}$
- **Formula** : $f(x) = \left(\frac{\beta}{s}\right)\left(\frac{x-l}{s}\right)^{\beta-1} e^{-\left(\frac{x-l}{s}\right)^{\beta}}$

The `Weibull` distribution has been successfully used to describe failure time in reliability studies, and the breaking strengths of items in quality control testing. For instance, by using a value of the shape parameter that is less than 1.0, the `Weibull` distribution becomes steeply declining and could be of interest to a manufacturer testing failures of items during their initial period of use.

`Beta` distribution

The `Beta` distribution, just as the `Weibull` distribution, is not easy to characterize. Whenever an uncertain variable can vary between 0 and a positive value, and you can set two parameters to determine the desired shape of the distribution, the `Beta` distribution may be worthwhile to consider.

- **Input parameters** : Alpha α, Beta β and scale s
- **Input check** : $\alpha > 0, \beta > 0,$ and $s > 0$
- **Permitted values** : $\{x \mid 0 < x < s\}$
- **Formula** : $f(x) = \frac{1}{B(\alpha,\beta)}\left(\frac{x}{s}\right)^{\alpha-1}\left(1-\frac{x}{s}\right)^{\beta-1}$

  where $B(\alpha,\beta)$ is the Beta function

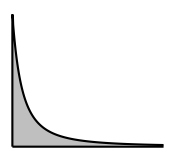
The `Beta` distribution is a very flexible distribution with a wide variety of shapes. These shapes can be made symmetrical, positively skewed, negatively skewed, etc. The distribution has been used to describe empirical data and predict the random behavior of percentages and fractions.

`Logistic` distribution

The `Logistic` distribution, just like the previous distributions, is also not easy to characterize. It has been used to describe growth of a population over time, chemical reactions, and similar processes.

- **Input parameters** : Mean μ and scale s
- **Input check** : $s > 0$
- **Permitted values** : $\{x \mid -\infty < x < \infty\}$
- **Formula** : $f(x) = \frac{z}{s(1+z)^2}, \quad z = e^{-\frac{x-\mu}{s}}$

`Pareto` distribution

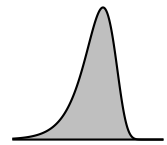


The `Pareto` distribution, just like the previous few distributions, is also not easy to characterize. It has been used to describe the sizes of such phenomena as human population, companies, incomes, stock fluctuations, etc.

- **Input parameters** : Location l and shape β
- **Input check** : $l > 0$ and $\beta > 0$
- **Permitted values** : $\{x \mid l < x < \infty\}$
- **Formula** : $f(x) = \frac{\beta\, l^{\beta}}{x^{\beta+1}}$

Extreme Value distribution

The Extreme Value distribution, just like the previous few distributions, is also not easy to characterize. It has 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.



- **Input parameters** : Mode m and scale s
- **Input check** : $s > 0$
- **Permitted values** : $\{x \mid -\infty < x < \infty\}$
- **Formula** : $f(x) = \dfrac{ze^{-z}}{s}, \quad z = e^{(\frac{x-m}{s})}$

A.3 Statistical operators

Statistical operators

The statistical operators discussed in this section can help you to analyze the results of an experiment. The following operators are available in AIMMS:

- the Mean (or Average) 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.

Mean

The following mean computation methods are supported: arithmetic mean or average, geometric mean, harmonic mean and root mean square (RMS) mean. 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 mean is defined as the square root of the sum of squares. It is mostly used for averaging the measurements of a physical process.

- **Operator** : Mean(*domain*,*expression*)
- **Formula** : $\dfrac{1}{n}\sum_{i=1}^{n} x_i$

- **Operator** : GeometricMean(*domain*,*expression*)
- **Formula** : $\sqrt[n]{\prod_{i=1}^{n} x_i}$

- **Operator :** `HarmonicMean(`*domain*,*expression*`)`
- **Formula :** $\dfrac{n}{\sum_{i=1}^{n} \dfrac{1}{x_i}}$

- **Operator :** `RootMeanSquare(`*domain*,*expression*`)`
- **Formula :** $\sqrt{\dfrac{1}{n} \sum_{i=1}^{n} x_i^2}$

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 average distance of a set of values from the mean. There are two kinds of standard deviation: the standard deviation of an 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 :** $\sqrt{\dfrac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \dfrac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)}$

whereas the standard deviation of a population can be determined by

- **Operator :** `PopulationDeviation(`*domain*,*expression*`)`
- **Formula :** $\sqrt{\dfrac{1}{n}\left(\sum_{i=1}^{n} x_i^2 - \dfrac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)}$

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. Lies the value 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 :** $\dfrac{\sum_{i=1}^{n}(x_i-\mu)^3}{\sigma_{n-1}^3}$

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).

- **Operator :** `Kurtosis(`*domain*,*expression*`)`
- **Formula :** $\dfrac{\sum_{i=1}^{n}(x_i-\mu)^4}{\sigma_{n-1}^4}$

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 :** $$\frac{n\sum_{i=1}^{n}x_iy_i-\sum_{i=1}^{n}x_i\sum_{i=1}^{n}y_i}{\sqrt{\left(n\sum_{i=1}^{n}x_i^2-\left(\sum_{i=1}^{n}x_i\right)^2\right)\left(n\sum_{i=1}^{n}y_i^2-\left(\sum_{i=1}^{n}y_i\right)^2\right)}}$$

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 :** $1 - \dfrac{6 \sum_{i=1}^{n} (\mathrm{Rank}(x_i) - \mathrm{Rank}(y_i))^2}{n(n^2 - 1)}$

Index