AIMMS Language Reference - Introduction to the AIMMS Language

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit www.aimms.com.

AIMMS B.V.
Diakenhuisweg 29-35
2033 AP Haarlem
The Netherlands
Tel.: +31 23 5511512

AIMMS Inc.
11711 SE 8th Street
Suite 303
Bellevue, WA 98005
USA
Tel.: +1 425 458 4024

AIMMS Pte. Ltd.
55 Market Street #10-00
Singapore 048941
Tel.: +65 6521 2827

AIMMS
SOHO Fuxing Plaza No.388
Building D-71, Level 3
Madang Road, Huangpu District
Shanghai 200025
China
Tel.: ++86 21 5309 8733

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

Part I

Preliminaries

Chapter 1

Introduction to the AIMMS language

Example makes a good starting point

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

Familiarity with algebraic notation

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

What to expect in this chapter

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

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

1.1 The depot location problem

The modeling process

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

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

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

Problem description

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

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

Use of sets

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

Initial set declarations

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

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

Parameters for input data

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

Example

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

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

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

Scalar data

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

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

Restricting permitted routes

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

Example

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

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

Explanation

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

Applying domain restrictions

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

Example

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

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

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

Effects of domain restriction

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

Additional parameter declarations

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

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

The AIMMS declarations are as follows.

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

1.2 Formulation of the mathematical program

Constraint-oriented modeling

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

Variables as unknowns

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

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

Example

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

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

The Range attribute

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

Constraints description

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

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

Example

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

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

Satisfying demand

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

Proper domain

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

Satisfying capacity

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

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

The objective function

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

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

Defined variables

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

The mathematical program

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

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

Explanation

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

Solving the mathematical program

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

```
Solve DepotLocationDetermination ;
```

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

1.3 Data initialization

Separation of model and data

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

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

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

Data sources

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

- commercial databases,
- text data files,

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

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

A simple data set in text format

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

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

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

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

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

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

Reading in the data

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

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

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

Automatic initialization

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

1.4 An advanced model extension

This section

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

Finding a robust solution

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

Algorithm in words

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

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

Additional identifiers

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

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

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

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

Outline of algorithm

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

```
<<Initialize algorithmic parameters>>

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

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

Initializing model parameters

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

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

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

TotalDemandDeviation         := 0;
TotalSquaredDemandDifference := 0;

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

Explanation

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

The .NonVar suffix

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

Determining depot frequencies

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

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

    Solve DepotLocationDetermination;

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

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

Explanation

Inside the WHILE statement the following steps are executed.

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

Functions used

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

Selecting a new depot

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

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

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

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

Explanation

In the above AIMMS program the following steps are executed.

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

- Register the new value of the estimated standard deviation in total demand.
- Assign 1 to all permanently selected depots, and fix their nonvariable status accordingly.

Functions used

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

1.5 General modeling tips

From beginner to advanced

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

Separation of model and data

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

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

Intellectual challenge

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

Levels of abstraction

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

Finding the proper level

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

From small to large-scale

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

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

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

Formulating proper domains of definition

For large applications you should always ask the following questions.

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

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

Example

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

Reformulation of algorithm

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