
AIMMS Language Reference - Data Initialization, Verification and Control

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 or order your hard-copy at www.lulu.com/aimms.

Copyright © 1993–2011 by Paragon Decision Technology B.V. All rights reserved.

Paragon Decision Technology B.V.	Paragon Decision Technology Inc.	Paragon Decision Technology Pte.
Schipholweg 1	500 108th Avenue NE	Ltd.
2034 LS Haarlem	Ste. # 1085	80 Raffles Place
The Netherlands	Bellevue, WA 98004	UOB Plaza 1, Level 36-01
Tel.: +31 23 5511512	USA	Singapore 048624
Fax: +31 23 5511517	Tel.: +1 425 458 4024	Tel.: +65 9640 4182
	Fax: +1 425 458 4025	

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

AIMMS is a registered trademark of Paragon Decision Technology B.V. IBM ILOG CPLEX and sc CPLEX is a registered trademark of IBM Corporation. GUROBI is a registered trademark of Gurobi Optimization, Inc. KNITRO is a registered trademark of Ziena Optimization, Inc. XPRESS-MP is a registered trademark of FICO Fair Isaac Corporation. MOSEK is a registered trademark of Mosek ApS. WINDOWS and EXCEL are registered trademarks of Microsoft Corporation. \TeX , \LaTeX , and $\AMS-\LaTeX$ are trademarks of the American Mathematical Society. LUCIDA is a registered trademark of Bigelow & Holmes Inc. ACROBAT is a registered trademark of Adobe Systems Inc. Other brands and their products are trademarks of their respective holders.

Information in this document is subject to change without notice and does not represent a commitment on the part of Paragon Decision Technology B.V. The software described in this document is furnished under a license agreement and may only be used and copied in accordance with the terms of the agreement. The documentation may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Paragon Decision Technology B.V.

Paragon Decision Technology B.V. makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Paragon Decision Technology B.V., its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claims for lost profits, fees or expenses of any nature or kind.

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors shall not be responsible under any circumstances for providing information or corrections to errors and omissions discovered at any time in this book or the software it describes, whether or not they are aware of the errors or omissions. The authors, Paragon Decision Technology B.V. and its employees, and its contractors do not recommend the use of the software described in this book for applications in which errors or omissions could threaten life, injury or significant loss.

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

Part VI

Data Communication Components

Chapter 23

Data Initialization, Verification and Control

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.

Aspects of the use of data

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.

This chapter

23.1 Data initialization

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.

Separation of model and data

There are several methods to input the initial data of the identifiers in your model. AIMMS allows you:

Supplying initial data

- 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.

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.

Interactive initialization

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

Data initialization sequence

- 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.

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 Main-Initialization procedure

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 26.2.

The attribute INITIAL DATA

23.1.1 Reading data from external sources

You can use the READ statement to initialize data from the following external data sources:

The READ statement

- user-supplied ASCII files containing constant lists and tables,
- AIMMS-generated binary case files, and
- external ODBC- and OLE DB- compliant databases.

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.

Reading from ASCII data files

Reading from ASCII files is especially useful when

When useful

- 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

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.

Reading from binary case files

Reading from case files is especially useful when

When useful

- 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.

A third (and powerful) application of the READ statement is the retrieval of data from any ODBC- or OLE DB-compliant database. This form of data initialization gives you direct access to up-to-date corporate databases.

Reading from databases

Reading from databases is especially useful when

When useful

- 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.

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.

Computing initial data

23.2 Assertions

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.

Data validity is important

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

ASSERTION declaration and attributes

by the user from within the graphical user interface. The attributes of the ASSERTION type are given in Table 23.1.

Attribute	Value-type	See also page
INDEX DOMAIN	<i>index-domain</i>	42, 207
TEXT	<i>string</i>	19, 45
PROPERTY	WarnOnly	
ASSERT LIMIT	<i>integer</i>	
DEFINITION	<i>logical-expression</i>	33, 44
ACTION	<i>statements</i>	
COMMENT	<i>comment string</i>	19

Table 23.1: ASSERTION attributes

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.

The DEFINITION attribute

```

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) ;

```

Examples

The assertion SupplyExceedsDemand is a global check. The assertion CheckTransportData(i,j) is verified for every pair of cities i and j for which Distance(i,j) assumes a nonzero value. AIMMS will terminate further verification when the assertion fails for the third time.

The TEXT attribute 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 error/message window when the ASSERTION is called through the ASSERT statement in a procedure body, or displayed in a message dialog box to the user when an ASSERTION is linked to a page object in the AIMMS GUI. 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 TEXT 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 `PROPERTY` 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 `ASSERT LIMIT` 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 `ACTION` attribute

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.

The `FailCount` operator

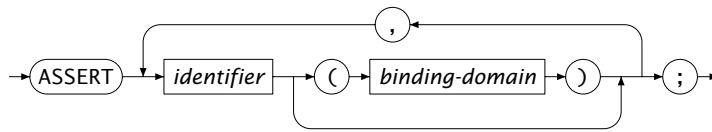
Assertions can be verified in two ways:

Verifying assertions

- 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.

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.

The `ASSERT` statement

*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);
```

23.3 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?

AIMMS offers the following data control tools:

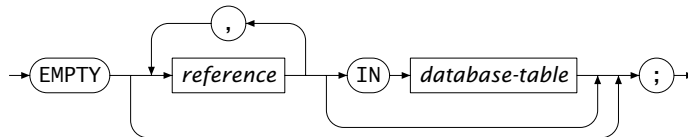
Facilities

- the EMPTY statement to remove the contents from all or a selected number of identifiers,
- the CLEANUP and CLEANDEPENDENTS statements to clean up all, or a selected number of identifiers,
- the REBUILD statement to manually instruct AIMMS to reclaim unused memory from the internal data structures used to store the data of a selected number of identifiers,
- the procedure FindUsedElements to find all elements of a particular set that are in use in a given collection of indexed model identifiers, and
- the procedure RestoreInactiveElements to find and restore all inactive elements of a particular set for which inactive data exists in a given collection of indexed model identifiers.

The EMPTY statement can be used to discard the complete contents of all or selected identifiers in your model. Its syntax follows.

The EMPTY statement

empty-statement :



The EMPTY operator operates on a list of references to AIMMS identifiers and takes the following actions.

Empty AIMMS identifiers

- 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.

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 25.4. You should note, however, that applying the EMPTY statement to

Use in databases

a subset of AllIdentifiers does *not* apply to any database table contained in the subset to avoid inadvertent deletion of data.

The following statements illustrate the use of the EMPTY operator.

Examples

- 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 ;
```

When you remove some but not all elements from a domain set, AIMMS will not automatically discard the data associated with those elements for every identifier defined over the particular domain set. AIMMS will also not automatically discard data that does not satisfy the current domain restriction of a given identifier. Instead, it will consider such data as *inactive*. During the execution of your model, no reference will be made to inactive data, but such data may still be visible in the user interface. In addition, AIMMS will not directly reclaim the memory that is freed up when the cardinality of a multidimensional identifier in your model decreases.

Inactive data

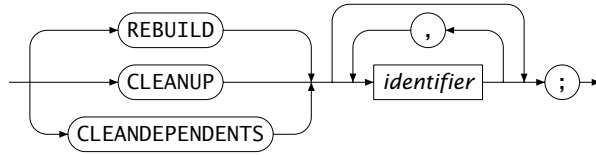
The facility to create inactive data in AIMMS allows you to temporarily remove elements from domain sets when this is required by your model. You can then restore the data after the relevant parts of the model have been executed.

When useful

If you want to discard inactive data that has been introduced in a particular data set, you can apply the CLEANUP statement to parameters and variables, or the CLEANDEPENDENTS statement to root sets in your model. Through the REBUILD statement you can instruct AIMMS to reclaim the unused memory associated with one or more identifiers in your model. The syntax follows.

Discard inactive data

cleanup-statement :



The following rules apply when you call the CLEANUP statement.

Rules

- When you apply the CLEANDEPENDENTS statement to a set, all inactive elements are discarded from the set itself and from all of its subsets. In addition, AIMMS will discard all inactive data throughout the model caused by the changes to the set.
- When you apply the CLEANUP statement to a parameter or variable, all inactive data associated with the identifier is removed. This includes inactive data that is caused by changes in domain and range sets, as well as data that has become inactive by changes in the domain condition of the identifier.
- When you apply the CLEANDEPENDENTS, CLEANUP, or REBUILD statement to a section, AIMMS will remove the inactive data of all sets, or parameters and variables declared in it, respectively.

After using the CLEANUP or CLEANDEPENDENTS statement for a particular identifier, all its associated inactive data is permanently lost.

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 root set elements

- Resorting will only take place if the current storage order of a root set differs from its current ordering principle.
- AIMMS will not resort sets for which explicit elements are used in the model formulation.

As a call to CLEANDEPENDENTS requires a complete rebuild of all identifiers defined over the root sets involved, the CLEANDEPENDENTS statement may take a relatively long time to complete. For a more detailed description of the precise manner in which root set elements and multidimensional data is stored in AIMMS refer to Section 13.2.7. This section also explains the benefits of resorting a root set.

You should not call the CLEANDEPENDENTS statement in procedures that have been linked to edit actions in graphical objects in an AIMMS end-user GUI via the **Procedures** tab of the object **Properties** dialog box. During these actions, AIMMS does not expect the element numbering to change.

*Restricted usage
in AIMMS GUI*

If you want to apply the CLEANDEPENDENTS statement to multiple sets, applying the operation to all sets in a single call of the CLEANDEPENDENTS statement will, in general, be more efficient than using a separate call for every single set. If an identifier depends on two or more of the sets to which you want to apply the CLEANDEPENDENTS operation, the data of such an identifier will only be traversed and/or rebuild once, rather than multiple times.

*Efficiency
considerations*

- 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'.

Examples

```
Cities -= 'Amsterdam' ;
cleandependents Cities ;
```

- The following CLEANUP statement will remove the data of the identifier Transport for all tuples that either lie outside the current contents of Cities, or do not satisfy the domain restriction.

```
cleanup Transport;
```

- Consider a parameter $A(i, j)$ where i is an index into a set S and j an index into a set T , then

```
cleandependents S, T;
```

will be more efficient than

```
cleandependents S;
cleandependents T;
```

because the latter may require $A(i, j)$ to be rebuilt twice.

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:

*Finding used
elements*

- 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.

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*.

Example

```
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;
```

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:

Finding and restoring inactive elements

- 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.

The following call to `RestoreInactiveElements` will verify whether inactive data exists for the set `Cities` in `AllIdentifiers`.

Example

```
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;
```

If the cardinality of a multidimensional identifier in your model decreases, AIMMS will not automatically reclaim the memory that is freed up because of the decreased amount of data to store. Instead, it will keep the memory available to store additional data that is associated with subsequent changes to the identifier. If the cardinality of an identifier decreases dramatically during a run the of a model, this may lead to a huge amount of memory getting stuck up with a single identifier in your model.

*Reclaiming
memory*

In addition, if a model is running for a prolonged period of time, and an identifier has undergone huge amounts of structural changes during that time, the memory associated with that identifier may become heavily fragmented. In the long run, memory fragmentation may lead to decreased performance of your model. Rebuilding the internal data structures associated with such an identifier will resolve the fragmentation problem.

*Memory
fragmentation*

Prior to solving a mathematical program, AIMMS will perform a quick check comparing the total amount of memory used by an identifier to the amount of unused memory associated with that identifier. By adding to and removing elements from identifiers, memory may become fragmented and the fraction of unused memory may grow. If the fraction of unused memory compared to the total amount of memory in use becomes too large, AIMMS will automatically rebuild such an identifier in order to reclaim the unused memory. AIMMS will also reclaim the memory of an identifier whenever it becomes empty during the run of a model.

*Automatic
reclamation*

Through the `REBUILD` statement you can manually instruct AIMMS to rebuild the internal data structures associated with one or more identifiers. During the `REBUILD` statement AIMMS uses a more thorough check to verify whether a rebuild of an identifier is worthwhile then prior to solving a mathematical program.

*the REBUILD
statement*

23.4 Working with the set `AllIdentifiers`

Throughout your model you can use the predefined set `AllIdentifiers` to construct and work with dynamic collections of identifiers in your model. Several operators in AIMMS support the use of a subset of `AllIdentifiers` instead of an explicit list of identifier names, while other operators support the use of an index into `AllIdentifiers` instead of a single explicit identifier name.

*Working with
`AllIdentifiers`*

AIMMS offers a number of constructs that can help you to construct a meaningful subset of `AllIdentifiers`. They are:

*Constructing
identifier sets*

- set algebra with other predefined identifier subsets, and
- dynamic selection based on model query functions.

When compiling your model, AIMMS automatically creates an identifier set for every section in your model. Each such set contains all the identifier names that are declared in the corresponding section. In addition, for every identifier type, AIMMS fills a predeclared set *AllIdentifierType* (e.g. `AllParameters`, `AllSets`) with all the identifiers of that type. The complete list of identifier type related sets defined by AIMMS can be found in the AIMMS Function Reference. You can use both type of sets to perform set algebra to construct particular identifier subsets of interest to your model.

*Predefined
identifier sets*

If your model contains a section `Unit Model`, you can assign the collection of all parameters in that section to a subset `UnitModelParameters` of `AllIdentifiers` through the assignment

Example

```
UnitModelParameters := Unit_Model * AllParameters;
```

Another method to construct meaningful subsets of `AllIdentifiers` consists of using the functions provided to query aspects of those identifiers. Selected examples are:

*Model query
functions*

- the function `IdentifierDimension` returning the dimension of the identifier,
- the function `IdentifierType` returning the type of the identifier as an element of `AllIdentifierTypes`,
- the function `IdentifierText` returning the contents of the TEXT attribute, and
- the function `IdentifierUnit` returning the contents of the UNIT attribute.

These functions take as argument an element in the set `AllIdentifiers`.

In addition to the functions lists above, the functions `Card` and `ActiveCard` also accept an index into the set `AllIdentifiers`. They will then return the cardinality of the identifier represented by the index, or the cardinality of the active elements of that identifier, respectively. You can also use these functions to dynamically construct a subset of `AllIdentifiers`.

*Functions
accepting
identifier index*

The set expression

Example

```
{ IndexIdentifiers in UnitModelParameters |
  IdentifierDimension( IndexIdentifier ) = 3 }
```

refers to the collection of all 3-dimensional parameter in the section `Unit Model`.

The following operators in AIMMS support identifier subsets to represent a collection of individual identifiers:

*Working with
identifier sets*

- the `READ` and `WRITE` operators,
- the `EMPTY`, `CLEANUP`, `CLEANDEPENDENTS`, and `REBUILD` operators.

If you are interested in the contents of an identifier subset, you can use the `DISPLAY` operator, which will just print the identifier names contained in the set, rather than the contents of the identifiers referred to in the identifier set as is the case for the `WRITE` statement.

In addition to the operators above, the following AIMMS functions also operate on subsets of `AllIdentifiers`:

*Functions
accepting
identifier sets*

- `GenerateXML`,
- `CaseCompareIdentifier`,
- `CaseCreateDifferenceFile`,
- `IdentifierMemory`,
- `GMP::Solution::SendToModelSelection`,
- `VariableConstraints`,
- `ConstraintVariables`,
- `ScalarValue`,
- `SectionIdentifiers`,
- `AttributeToString`,
- `IdentifierAttributes`.

See also Section "Model Query Functions" on page 2 of AIMMS the Function Reference.