
AIMMS Language Reference - Model Structure and Modules

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. $\text{T}_{\text{E}}\text{X}$, $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$, and $\text{A}_{\text{M}}\text{S}_{\text{L}}\text{A}_{\text{T}}\text{E}_{\text{X}}$ 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 $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ and the LUCIDA font family.

Chapter 33

Model Structure and Modules

This chapter discusses the common structuring components of a model, namely the *main model* and model *sections*. With the use of sections, you can provide depth to the model tree in the AIMMS **Model Explorer**, which allows you to structure your model in any logical manner that makes sense to you. Imposing a clear and logical structure to your model will strongly add to the overall maintainability of your modeling application.

Model and sections

The next concept introduced in this chapter is that of a *module*, which is basically a model section with its own, separate, namespace. Modules allow you to share sections of model source between multiple models, without the risk of running into name clashes. AIMMS uses modules to implement those parts of its functionality that can be best expressed in the AIMMS language itself. The available AIMMS system modules include

Modules

- a (customizable) implementation of the outer approximation algorithm,
- a scenario generation module for stochastic programming, and
- sets of constants used in the graphical 2D- and 3D-chart objects.

Finally, this chapter discusses the concept of a *library module*, which is the source module associated with a library project (see Section 3.1 of the User's Guide). Library modules can only be added to an AIMMS model through the **Library Manager**, and are always displayed as a separate root in the model tree.

Library modules

33.1 Introduction

When a model grows larger, the need for a clear and logical storage structure of all its constituting components also grows. In the absence of such a logical storage structure, you will find that it becomes increasingly hard to find your way in the model source because of the huge amount of information it contains. To support you in structuring your model, AIMMS offers several development tools and language constructs just for this purpose.

Support for large models

To support you in structuring your model, AIMMS lets you organize all identifier declarations and procedures of your model in the form of a tree, called the *model tree*. You can access the model tree in a graphical manner, using the **Model Explorer** tool (see also Chapter 4 of the User's Guide). Several language constructs of the AIMMS language, such as collections of identifiers declarations, or the procedures and functions included in your model, are visible as separate nodes within the model tree.

The model tree

All model declarations in an AIMMS model are located underneath the root node of the model tree, the MAIN MODEL node. The MAIN MODEL node is always present in the **Model Explorer**, even when you start a new AIMMS project, and cannot be deleted. For the MAIN MODEL node itself, you can specify several attributes that have a global impact, such as the licensing arrangements for your model, or the unit convention (if any) that is applicable to your application as a whole. The attributes of the MAIN MODEL node are discussed in full detail in Section 33.2.

Main model node

As you start adding identifier declarations, procedures and functions to a new model, you will soon notice that storing all these declarations directly underneath the MAIN MODEL node will result in a nearly unmanageable list of declarations. Finding information in such a (linear) list soon becomes a daunting task. To support you in adding additional structure to your model tree AIMMS provides SECTION nodes, which allow you to add depth to the model tree, much like directories add depth to a file system.

*Model sections
...*

By adding SECTION nodes with meaningful names to the model tree, and storing all model declarations that you find relevant for these section underneath them, you can impose any logical structure on your model tree that you find useful. Because AIMMS allows identifiers to be used prior to their declaration, you do not even have to worry about the declaration order when you reorganize the model tree in this manner.

... to structure a model

In addition to providing the structuring capabilities described above, the contents of a SECTION node can also be stored in a separate source file. You can import the contents of such a source file into a section of another model, or permanently link the contents of a section to the contents of the source file. This allows you to reuse part of one model within similar applications. This method of sharing functionality, however, has its limitations. Name clashes can occur when an imported section redeclares an identifier already declared in the main model. If you run into these limitations, you are advised to use the MODULE concept discussed below.

Separate source file

The attributes of a SECTION node allow you to specify such issues as whether its contents needs to be stored in a separate source file, and if the usage of such a source file needs to be licensed. The attributes of a SECTION node are discussed in full detail in Section 33.3.

*SECTION
attributes*

When the development of modeling applications becomes the core business of an organization, this will almost certainly lead to a multitude of related modeling projects, collaborating developers, and various end-user types, all subject to frequent changes over time. Projects evolve naturally due to feedback from end-users, changing application environments, and rotating personnel. Changes in the pool of model developers are inevitable, and may cause major fluctuations in application knowledge, experience, and modeling skills. End-users of applications also change jobs, which may result in new requirements and customization requests from the newcomers.

*Complex
modeling
environments*

In such a dynamic modeling world, the exchange of information becomes a crucial element to avoid unnecessary duplication. When projects are customized for different end-users, there is apt to be quite a bit of commonality between these projects. If these commonalities are not exchanged properly, there will be multiple and differing versions of essentially the same model segments. As a result, extensive and costly human resources will be needed to maintain these multiple related models. Modularization can help to overcome these problems.

*Need for
modularization*

In Chapter 10 you were introduced to functions and procedures as the initial tools to modularize the functionality within an AIMMS project. As explained above, collections of functions and procedures, along with the required identifier declarations, can be stored in model sections. These can be exported to separate .amb files, and can subsequently be imported by, or linked into, any other AIMMS project. Every time an AIMMS project is started containing a section link, it will automatically pick up the latest version of the file. This means that when such a collection of functions, procedures and identifier declarations at a customer's site need to be updated, only their corresponding files need to be replaced.

*Collections of
functions and
procedures*

One problem that you are likely to run into with the above approach, however, is the occurrence of name clashes. Some of the identifier names of procedures, functions, and identifiers in a model section may also occur in the model in which the section is to be included. Such name clashes will effectively prevent AIMMS from importing or linking the section into your model. A possible solution to this problem would be to rename the offending identifiers, either in your model or in the section to be included. However, using either approach, the same problems are likely to return when you get an updated version of the included model section.

Name clashes

A more structural solution to the name clash problem is provided by the concept of *modules* in AIMMS, which allow you to share common model source into multiple models, without the risk of running into name clashes. Modules are inserted into the model tree by means of MODULE nodes. These nodes are essentially SECTION nodes with a separate namespace, along with attributes to manipulate the global model namespace. The attributes of MODULE nodes are discussed in full detail in Section 33.4.

Modules ...

Like SECTION nodes, MODULE nodes can be exported to a separate source file, which can be imported or linked into another model. However, because all identifiers declared within the MODULE node only live in its associated namespace, importing a module into another project will not lead to name clashes anymore.

... avoid name clashes

When a project becomes larger, the operational demands and sheer amount of work involved in implementing the project, may become too demanding for a single modeler to keep up with. It is then time to divide the project into a number of manageable sub-projects, on which individual developers can work more or less independently.

Dividing a project into sub-projects...

Modules, as discussed above, are not necessarily the most suitable instrument to facilitate a division into sub-projects. This is mainly due to the fact that the module concept does not allow identifiers in the module to be strictly private to that module. Because of this, other developers can, in principle, refer to all identifiers in the module, and, consequently, the chances of a single structural change in any of the modules breaking the entire application are considerable.

... unsuitable for modules

To address the problem of allowing multiple developers to work independently on manageable sub-projects of a big AIMMS project more thoroughly, AIMMS supports the concept of *library projects*. Library projects go far beyond modules—they do not only support independent model development, but a developer can also create end-user pages and menus as part of the library project. When a library project is included in a main project, the associated overall application can then be composed by combining the model source, pages, and menus created as part of all its included libraries. Library projects are discussed in full detail in Chapter 3 of the User's Guide.

Library projects

Library modules are the source code modules associated with library projects. They can only be added to your model through the **Library Manager** discussed in Section 3.1 of the User's Guide. AIMMS will insert library modules into the model tree as a separate LIBRARY MODULE root node. The attributes of LIBRARY MODULE nodes are discussed in full detail in Section 33.5.

Library modules

As with ordinary modules, library modules have an associated namespace, which helps to avoid name clashes when including a library project into an AIMMS project. In addition, however, library modules provide a public *interface* to the rest of the model. Within the library project, all identifiers declared in the library can be freely used in the source of the library module, its pages and menus. The main project, and all other library projects included in the main project, however, can only access the identifiers that are part of the interface of the library. This allows a developer of a library to freely change any declaration that is not part of the library interface, without the risk of breaking the entire application.

Library interface

33.2 MAIN MODEL declaration and attributes

The MAIN MODEL node defines the root node of the entire model tree associated with an AIMMS modeling project. The attributes of the MAIN MODEL node are listed in Table 33.1. All attributes of the MAIN MODEL node have a global impact on the entire modeling project.

MAIN MODEL declaration and attributes

Attribute	Value-type	See also page
LICENSE FILE	<i>string</i>	
MODULE CODE	<i>integer</i>	
USER DATA	<i>string-identifier</i>	
CONVENTION	<i>convention, element-parameter</i>	
COMMENT	<i>comment string</i>	19

Table 33.1: MAIN MODEL attributes

Through the LICENSE FILE attribute you can indicate that the use of your model is subject to a VAR license, which you as the owner of the model must distribute to all your customers using the model. The value of this attribute must be a string containing the name of the VAR license file itself, or of the directory containing all VAR license files specific to particular AIMMS licenses. The use and creation of VAR licenses is discussed in full detail in Section 21.2 of the User's Guide.

The LICENSE FILE attribute

With each model or model section protected by a VAR license, you can associate an integer module code. This code allows you to distinguish several AIMMS applications protected by a VAR license. Through the MODULE CODE attribute you can indicate with which module code the current model is to be associated. If this does not correspond to the module code stored in the VAR license file specified in the LICENSE FILE attribute, AIMMS will not allow the

The MODULE CODE attribute

model to be run. You can find more information on module codes in Section 21.2 of the User's Guide.

In addition to protecting your model through a VAR license, you can also use the VAR license to store specific customer-specific string information, which might be used by your application. Through the USER DATA attribute you can retrieve the string information stored in the VAR license file during its creation (see also Section 21.2.1 of the User's Guide). The value of the USER DATA attribute must be a scalar string parameter. This parameter allows you to retrieve the string information stored in the VAR license file after AIMMS has checked the VAR license.

The USER DATA attribute

With the CONVENTION attribute you can indicate that all I/O with respect to identifiers in your model is to take place according to the unit convention specified in this attribute. The value of this attribute must be either a direct reference to a convention declared in your model, an element parameter into the set AllConventions or a string parameter holding the name of a convention within your model. You can find more detailed information about unit conventions and their usage in Section 30.8

The CONVENTION attribute

33.3 SECTION declaration and attributes

SECTION nodes provide depth to the model tree, and offer facilities to store parts of your model in separate source files. A SECTION node is always a child of the MAIN MODEL node, of another SECTION node, or of a MODULE node. The attributes of SECTION nodes are listed in Table 33.2.

SECTION declaration and attributes

Attribute	Value-type	See also page
SOURCE FILE	<i>string</i>	
LICENSE FILE	<i>string</i>	540
MODULE CODE	<i>integer</i>	540
USER DATA	<i>string identifier</i>	541
COMMENT	<i>comment string</i>	19

Table 33.2: SECTION attributes

With the SOURCE FILE attribute you can indicate that the contents of a SECTION node in your model is linked to the specified source file. As a consequence, AIMMS will read the contents of the SECTION node from the specified file during compilation of the model. Any modifications to the part of the model contained in such a SECTION node will also be stored in this source file when you save the model. When you select an existing source file for the SOURCE FILE

The SOURCE FILE attribute

attribute of a SECTION node in the **Model Explorer** (see also Section 4.2 of the User's Guide), any previous contents of that section will be lost.

Whenever the contents of a SECTION node is stored in a separate source file, AIMMS allows you to separately protect the contents of that source file through a VAR license in a similar fashion as you would protect the MAIN MODEL node (see page 540 and further). This enables you to restrict the usage of the functionality contained in a section to exactly those customers with an associated valid VAR license. The LICENSE FILE, MODULE CODE and USER DATA attributes of a SECTION node must hold similar data as the corresponding attributes of the MAIN MODEL node.

The LICENSE FILE, MODULE CODE and USER DATA attributes

When you decide to license the use of a section of your model, the following steps are required.

Licensing sections

- Make sure that your AIMMS license is supplied with a VAR identification code. Without it, you will not be able to create VAR license files, nor will the licensing-related attributes of a SECTION node be visible in its attribute form.
- Create a VAR license file for the SECTION node to be licensed (see also Section 21.2.1 of the User's Guide).
- Enter the name of this license file in the LICENSE FILE attribute of the SECTION node to be licensed. This will automatically copy the module code stored in the VAR license file into the MODULE CODE attribute.
- Write the contents of the section to a source file by specifying the SOURCE FILE attribute. As a result, AIMMS will write the section to the specified file in an encrypted form, and replace the current contents of the section by a link to the source file.
- During subsequent sessions you still have developer access to the contents of the section, because the VAR identification code stored in the source file matches the one stored in your AIMMS license.
- If some of your code in SECTION node needs access to the user data stored in a VAR license file (which may be different for every VAR license issued), you should enter the name of the string parameter (preferably declared in the section itself). Through this string parameter you can access the data referred to in the USER DATA attribute.

When you want to add a link to a licensed source file as part of a SECTION node in your model, the following steps are required.

Using licensed sections

- Create an empty SECTION node in your model, to which the source file should be linked.
- Create the link by specifying the name of the source file into the SOURCE FILE attribute of the SECTION node.
- Because the source file is licensed, AIMMS will request you to specify a VAR license file with a VAR identification and module code that match

the codes stored in the licensed source file. As a result of this step, AIMMS will enter the name of the license file and the corresponding module code in the LICENSE FILE and MODULE CODE attributes of the section.

- If the VAR identification code of the section matches the VAR identification code in your own AIMMS license, you will be granted developer access to the section. Otherwise, you will only be granted end-user access to the section, and the subtree will remain locked.

Whenever you add a SECTION node to the model tree, the name of the SECTION node (with spaces replaced by underscores), will also be available within your model as an implicit subset of the predeclared set AllIdentifiers. The contents of this subset is fixed, and is defined as the set of all identifiers declared within the subtree corresponding to the SECTION node. You can use this implicitly created set, for instance, in the EMPTY statement to empty all section identifiers using only a single statement.

Section names as identifier subsets

33.4 MODULE declaration and attributes

MODULE nodes create a subtree of the model tree along with a separate namespace for all identifier declarations in that subtree. Like SECTION nodes, the model contents associated with a MODULE node can be stored in a separate source file. A MODULE node is always a child of the MAIN MODEL node, of a SECTION node, or of another MODULE node. The attributes of MODULE nodes are listed in Table 33.3.

MODULE declaration and attributes

Attribute	Value-type	See also page
SOURCE FILE	<i>string</i>	541
LICENSE FILE	<i>string</i>	540
MODULE CODE	<i>integer</i>	540
USER DATA	<i>string-identifier</i>	541
PREFIX	<i>identifier</i>	
PUBLIC	<i>identifier-list</i>	
PROTECTED	<i>identifier-list</i>	
COMMENT	<i>comment string</i>	19

Table 33.3: MODULE attributes

Like with ordinary SECTION nodes, the contents of a MODULE node can also be stored in a separate source file, dynamically linked into a MODULE node in your model through the use of the SOURCE FILE attribute, and licensed through the use of the LICENSE FILE, MODULE CODE and USER DATA attributes. The details of storing a MODULE node in a separate source file and for licensing a MODULE node

The SOURCE FILE, LICENSE FILE, MODULE CODE and USER DATA attributes

and using a licensed MODULE are identical as for SECTION nodes, and have been explained in full detail in Section 33.3.

The distinguishing feature of modules is that each module is supplied with a separate namespace. This means that all identifiers, procedures and functions declared within a module are, without using the module prefix, only visible within that module. In addition, within a module it is possible to redeclare identifier names that have already been declared outside the module.

Modules and namespaces

Modules in an AIMMS model can be nested. This implies that with each AIMMS model containing one or more MODULE nodes, one can associate a corresponding tree of nested namespaces. This tree of namespaces starts with the global namespace of the MAIN MODEL node as the root node. As a consequence, you can associate a path of namespaces with every identifier, procedure or function declaration in the model tree. This path of namespaces starts with the global namespace down to the namespace associated with the module in which the declaration is contained.

Nested modules

When AIMMS encounters an identifier reference during the compilation of a procedure or function body or in one of the attributes of an identifier declaration, AIMMS will search for a declaration of the identifier at hand in the following order.

Scoping rules

- If the referenced identifier is declared in the namespace associated with the MODULE (or MAIN MODEL) in which the procedure, function or identifier is contained, AIMMS will use that particular declaration.
- If the referenced identifier cannot be found, AIMMS will repeatedly search the next higher namespace until a declaration for the identifier is found.

As a result of these scoping rules, whenever the corresponding identifier name is referenced within a module, AIMMS will always refer to the identifier declaration within the same module rather than to a possibly contradicting declaration for an identifier with the same name anywhere higher up, or sideways, in the model tree. This feature enables multiple developers to work truly independently on different modules used within a model.

Consequences

Consider the following model with two (nested) modules, called Module1 and Module2. The following can be concluded by applying the scoping rules listed above.

Example

- The reference to ShortestDistance in the procedure ComputeShortestDistance in the module Module1 refers to the declaration ShortestDistance within that module, and *not* to the declaration ShortestDistance in the main model.
- The reference to Distance in the procedure ComputeShortestDistance in the module Module1 refers to the declaration Distance(i,j) in the main

```

MAIN MODEL TransportModel
...
PARAMETER:
  identifier : Distance
  index domain : (i,j) ;
PARAMETER:
  identifier : ShortestDistance ;
...
MODULE Module1
  PREFIX : m1
  ...
  PARAMETER:
    identifier : ShortestDistance ;
  ...
  PROCEDURE ComputeShortestDistance
    body:
      ShortestDistance :=
        min((i,j), Distance(i,j));
  ...
MODULE Module2
  PREFIX : m2
  ...
  PARAMETER:
    identifier : Distance
    definition : ShortestDistance ;
  ...
ENDMODULE ;
...
ENDMODULE ;
...
ENDMODEL ;

```

model, and *not* to the scalar declaration `Distance` within the nested module `Module2`.

- The reference to `ShortestDistance` in the module `Module2` refers to the declaration `ShortestDistance` within the module `Module1`, and *not* to the declaration `ShortestDistance` in the main model.
- The parameter `Distance` in the module `Module2` does not conflict with the declaration of `Distance(i,j)` in the main model, because the former is only visible within the scope of the module `Module2`.

The separate namespace of every module actively prevents identifiers within a module from being “seen” outside the module. For this reason, identifiers declared within a module are also referred to as *protected* identifiers. AIMMS, however, still allows you to reference protected identifiers anywhere else in your model through the use of the *namespace resolution operator* `::`. In combination with a module-specific prefix, the `::` operator accurately lets you indicate that you are referring to a protected identifier declared in the particular module associated with the prefix.

Accessing protected identifiers

With the *mandatory* `PREFIX` attribute of a `MODULE` node, you must specify a module-specific prefix to be used in conjunction with the `::` operator. The value of the `PREFIX` attribute should be a unique name within the namespace of

The PREFIX attribute

the surrounding module (or main model), and will subsequently be added to this namespace. In conjunction with the `::` operator the prefix unambiguously identifies the namespace from which a particular identifier should be taken.

With the *namespace resolution operator* `::` you instruct AIMMS to look for the identifier directly following the `::` operator within the module associated with the prefix in front of it. The `::` operator may be optionally surrounded with spaces. By stacked use of the `::` operator you can indicate that you want to refer to an identifier declared in a nested module. Each next prefix should refer to the `PREFIX` attribute of the module declared directly within the module associated with the previous prefix.

*The ::
namespace
resolution
operator*

If you want to refer to an identifier in the main model, that is also declared elsewhere along the path from the current module to the main model, you can use the `::` operator *without a prefix*. This indicates to AIMMS that you are interested in an identifier declared in the global namespace associated with the main model.

*Using global
identifiers in
MODULES*

Consider the model outlined in the example above.

Examples

- Within the main model, a reference `m1::ShortestDistance` would refer to the parameter `ShortestDistance` declared within the module `Module1`, and not to the parameter `ShortestDistance` declared in the main model itself.
- Within the main model, a reference `m1::m2::Distance` would refer to the parameter `Distance` declared in the module `Module2` nested within the module `Module1`.
- Within the module `Module1`, a reference to `::ShortestDistance` would refer to the parameter `ShortestDistance` declared in the main model, and not to the parameter `ShortestDistance` declared in `Module1`.
- Within the module `Module2`, a reference to `::Distance` would refer to the parameter `Distance` declared in the main model, and not to the parameter `Distance` declared in `Module2`.

The following model outline, which is a variation of the model outline of the previous example, further illustrates the consequences of the use of the `::` operator.

Through the `PUBLIC` attribute you can indicate that a set of identifiers declared within the module is public. These identifiers can then be referenced without the `::` operator within the importing module (or main model). The value of the `PUBLIC` attribute must be a constant set expression. You might consider the identifiers specified in the `PUBLIC` attribute as the public interface of a module. As a result, AIMMS will effectively add the names of these identifiers to the namespace of the importing module, as if they were declared within the importing module itself.

*The PUBLIC
attribute*

```

MAIN MODEL TransportModel
...
PARAMETER:
  identifier : Distance
  index domain : (i,j) ;
PARAMETER:
  identifier : ShortestDistance ;
...
MODULE Module1
  PREFIX : m1
  ...
  PARAMETER:
    identifier : ShortestDistance ;
  ...
  PROCEDURE ComputeShortestDistance
    body:
      ::ShortestDistance :=
        min((i,j), m2::Distance(i,j));
  ...
MODULE Module2
  PREFIX : m2
  ...
  PARAMETER:
    identifier : Distance
    definition : ::ShortestDistance ;
  ...
ENDMODULE ;
...
ENDMODULE ;
...
ENDMODEL ;

```

Consider the model outline of the first example, and assume that the declaration of module `Module2` is augmented as follows. *Example*

```

MODULE Module2
  PREFIX : m2
  PUBLIC : data { Distance } ;
  ...
  PARAMETER:
    identifier : Distance
    definition : ShortestDistance ;
  ...
ENDMODULE ;

```

As a result of the `PUBLIC` attribute, `Distance` will be added to the namespace of `Module1`, and the compilation of the procedure `ComputeShortestDistance` will fail because `Distance` will now refer the scalar declaration in `Module2` rather than to the 2-dimensional declaration in the main model. In addition, it is possible, within the main model, to refer to the parameter `Distance` in `Module2` through the expression `m1::Distance`, because `Distance` has been effectively added to the namespace of module `Module1`.

When an identifier is added to the PUBLIC attribute of an imported module, it is, as explained above, effectively added to the namespace of the importing module. This creates the possibility to add a public identifier of an imported module to the PUBLIC attribute of the importing module as well. In this way you can propagate the public character of such an identifier to the next outer namespace. For example, by adding the identifier `Distance` in the example above, to the PUBLIC attribute of the module `Module1` as well, it would also become public in the main model. Obviously, in this case, adding `Distance` to the PUBLIC attribute of `Module1` would cause a name clash with the global identifier `Distance(i, j)`.

*Propagation
of public
identifiers*

Once you import a module into an existing AIMMS application, one or more identifiers in the public interface of the imported module can cause name clashes with existing identifiers in the application, like `Distance` in the example of previous paragraph. When you run into such a problem, AIMMS allows you to override the PUBLIC status of one or more identifiers of a module through its PROTECTED attribute. The value of the PROTECTED attribute must be a constant set expression, and its contents must be a subset of the set of identifiers specified in the PUBLIC attribute. By adding an identifier to the PROTECTED attribute, it is, again, only accessible outside of the module by using the `::` operator.

*The PROTECTED
attribute*

The responsibilities for specifying the PUBLIC and PROTECTED attributes are substantially different, and result in a different storage of the values of these attributes. This is similar to the SOURCE FILE-related attributes discussed earlier in this chapter. The following rules apply.

*PUBLIC versus
PROTECTED
responsibilities*

- The PUBLIC attribute is intended for the *developer* of a module to define a public interface to the module. If the module is stored in a separate `.amb` file, to be imported by other AIMMS applications, the contents of the PUBLIC attribute is stored inside the module-specific `.amb` file.
- The PROTECTED attribute is intended for the *user* of a module to override the public character of certain identifiers as specified by the developer of the module. As the contents of the PROTECTED attribute is not an integral part of the module, but may be specified differently by every user of the module, it is never stored in a module-specific `.amb` file, but rather in the importing module or main model.

For each identifier in an AIMMS model, there is a unique global representation. If the identifier is contained in the global namespace of the main model, the global representation is the identifier name itself. If an identifier is only contained in the namespace of a particular module, its unique representation based on the namespace PREFIX of the module and the `::` operator. Thus, for the first example of this section (without PUBLIC attributes), the unique global representations of all identifiers are:

*Unique global
representation*

- `Distance(i, j)`

- ShortestDistance
- m1::ShortestDistance
- m1::ComputeShortestDistance
- m1::m2::Distance

With the PUBLIC attribute of Module2 defined as in the previous example, the unique global representation of the parameter Distance in Module2 becomes m1::Distance, as it effectively causes Distance to be contained in the namespace of Module1.

Whenever AIMMS is requested to DISPLAY or WRITE the contents of one or more identifiers in your model, it will use the unique global representation discussed in the previous paragraph. Also, when you READ data from a file, AIMMS expects all identifiers for which data is provided in the file to be identified by their unique global representation.

Display and data transfer

33.5 LIBRARY MODULE declaration and attributes

LIBRARY MODULE nodes create a separate tree in the model tree along with a separate namespace for all identifier declarations in that subtree. The model contents associated with a LIBRARY MODULE is always stored in a separate source file. Contrary to SECTION and MODULE nodes, the name of this source file cannot be specified in the LIBRARY MODULE declaration. Rather, it is specified when you add the library to your project using the **Library Manager**, as discussed in Section 3.1 of the User's Guide. The attributes of LIBRARY MODULE nodes are listed in Table 33.4.

LIBRARY MODULE declaration and attributes

Attribute	Value-type	See also page
LICENSE FILE	<i>string</i>	540
MODULE CODE	<i>integer</i>	540
USER DATA	<i>string-identifier</i>	541
PREFIX	<i>identifier</i>	
INTERFACE	<i>identifier-list</i>	
COMMENT	<i>comment string</i>	19

Table 33.4: LIBRARY MODULE attributes

Like with SECTION and MODULE nodes, the contents of a LIBRARY MODULE node can also be licensed through the use of the LICENSE FILE, MODULE CODE and USER DATA attributes. The details of using a licensed LIBRARY MODULE are identical as for SECTION nodes, and have been explained in full detail in Section 33.3.

The LICENSE FILE, MODULE CODE and USER DATA attributes

Like a normal module, each LIBRARY MODULE is supplied with a separate namespace. Compared to normal modules, however, the visibility rules for identifiers in a library modules are different. They are more in line with the intended use of libraries, i.e. to enable a single developer to work independently on the model source of a library.

Library modules and namespaces

Through the INTERFACE attribute of a LIBRARY MODULE you can specify the list of identifiers in the module that you want to be part of its public interface. Only identifiers in the library interface can be accessed in model declarations, pages and menu items that are not part of the library at hand. Library identifiers not in the interface are strictly private to the library, and can never be used outside of the library.

The INTERFACE attribute

With the *mandatory* PREFIX attribute of a LIBRARY MODULE node, you must specify a module-specific prefix to be used in conjunction with the :: operator. The value of the PREFIX attribute should be a unique name within the main model.

The PREFIX attribute

Even though identifiers in the interface of the library are visible outside of the library, AIMMS *always* requires the use of the library prefix to reference such identifiers. Library modules do not support the PUBLIC attribute of ordinary modules to propagate identifiers to the global namespace.

No propagation to global namespace

33.6 Runtime Libraries and the Model Edit Functions

Runtime libraries and the AIMMS Model Edit Functions permit applications to adapt to modern flexibility requirements on the model; at runtime you can create identifiers and use them subsequently. A few use cases, whereby the need for flexibility on the model grows, are briefly sketched below.

You may want to improve the maintainability of your application by

- Generating similar statements acting on (dynamic) selections of identifiers, or
- Generate the necessary parameters and database table identifiers with their mapping attributes by querying a relational database schema when setting up a database link with your model.

Use case: automating modeling tasks

In cooperative model development, a model is developed together with the users of that model. For instance, an existing application framework is visualized to the users and subsequently the suggestions from these users are taken into account. Such a suggestion might be to add structural nodes or arcs. Or such a suggestion might be to add a particular restriction on these nodes and arcs.

Use case: Cooperative model development

Moreover, not all structural information may be available at model development time; some users need to add their proprietary knowledge to the model at runtime. Examples of such proprietary knowledge are:

- Pricing rules for the valuation of portfolios.
- Blending rules for the prediction of property values of blends.

*Use case:
Proprietary user
knowledge*

A last use case of a modern flexibility requirement considers a user who has additional questions only when the results are actually presented. Such a user wants to question the model in order to understand a particular result. This person is only able to formulate the question when the (unexpected) result presents itself.

*Use case: ad hoc
user queries*

In the above use cases, applications create, manipulate, check, use and destroy AIMMS identifiers at runtime. These operations are performed by the Model Edit Functions. Such applications need to:

*Runtime editing
of identifiers*

1. Have a place to store these AIMMS identifiers in and retrieve them from. Such a place is called an AIMMS runtime library.
2. Have functions and procedures available to create, modify, check, and destroy these AIMMS identifiers. Together, these functions and procedures form the Model Edit Functions.
3. Have a way to use these identifiers inside the model.
4. *And be able to continue execution in the presence of errors.* This fourth requirement is an essential aspect to all other requirements and is central in the design of the AIMMS Runtime libraries and AIMMS Model Edit Functions. Global and local error handling is described in Section 8.4.1.

The identifiers created, modified, checked, used and destroyed at runtime are called runtime identifiers. These runtime identifiers are declared within a runtime library. A runtime library is itself also a runtime identifier; it can be created, modified, checked, used and destroyed at runtime as well. A runtime identifier can have any AIMMS type, except for quantity.

*Runtime
identifiers and
libraries*

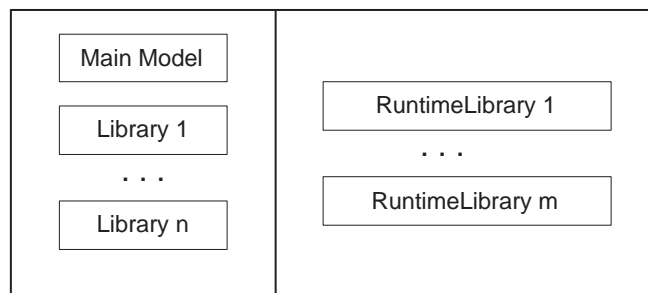


Figure 33.1: Separation between main application and runtime libraries

Model edit functions are only allowed to operate on runtime identifiers. Runtime identifiers exist at runtime and do not yet exist at compile time; the names of runtime identifiers cannot be used directly in the main model. This enforces a separation between identifiers in the main application and runtime identifiers as depicted in Figure 33.1. On the left of this architecture there is a main application consisting of a main model and 0, 1 or more libraries. On the right there are 0, 1 or more runtime libraries. Compilation errors can occur within runtime libraries at runtime. The identifiers inside the main application are not affected by such an error; provided it has local error handling, any procedure inside the main application can continue executing in the presence of compilation errors on identifiers in a runtime library. This is an important advantage of the separation; for several of the use cases presented above this separation enables continuation in the presence of errors.

*Separation
between main
application and
runtime
libraries*

In this example a runtime procedure `rp` is created and its body specified. This procedure is created in the runtime library `MyRuntimeLibrary1` with prefix `mr1`. The purpose of the runtime procedure `rp` is to write out the runtime parameter `P` declared in the same runtime library. This example assumes that both the runtime library `MyRuntimeLibrary1` and the runtime parameter `P` already exist.

*Example of
creating an
identifier*

```
PROCEDURE:
  identifier : DisplayDataOfRuntimeIdentifierTabular
DECLARATION SECTION
  ELEMENT PARAMETER:
    identifier : erp
    default   : 'MainExecution'
    range     : AllIdentifiers;
  STRING PARAMETER:
    identifier : str ;
  ELEMENT PARAMETER:
    identifier : err
    range      : errh::PendingErrors ;
  ELEMENT PARAMETER:
    identifier : err2
    range      : errh::PendingErrors ;
ENDSECTION ;
body      :
1  block
2    erp := me::Create("rp", 'procedure', 'MyRuntimeLibrary1', 0);
3    me::SetAttribute(erp, 'body', "display { P }");
4    me::Compile(erp);
5    me::Compile('MyRuntimeLibrary1');
6    Apply(erp);
7    me::Delete(erp);
8  onerror err do
9    if erp then
10     block
11       me::Delete(erp);
12     onerror err2 do
13       if errh::Severity(err2) = 'Severe' then
14         DialogMessage(errh::Message(err2) +
15           "; not prepared to handle severe errors " +
16           "and halting execution");
17       halt ;
```

```

18         else
19             errh::MarkAsHandled(err2) ;
20         endif ;
21     endblock ;
22     erp := '' ;
23 endif ;
24 errh::MarkAsHandled(err);
25 DialogMessage("Creating and executing rp failed; " + errh::Message(err) );
26 endblock ;
ENDPROCEDURE ;

```

A line by line explanation of this example follows below.

- **Lines 1, 8, 25:** In order to handle the errors during a group of model edit actions, a BLOCK statement with an ONERROR clause is used.
- **Lines 2 - 7:** Contain the calls to the model edit functions. Note that they can and are formulated without any concern for errors because these errors are handled in line 9 - 25.
- **Line 2:** Create the procedure rp as the last procedure in the runtime library MyRuntimeLibrary1. The prefix of the library will be prefixed to the name of the identifier created; after this statement the value of the element parameter erp is 'mr1::rp'.
- **Line 3:** Set the contents of the body of that procedure. Here it is to display the parameter P in tabular format.
- **Line 4:** Check the procedure mr1::rp for errors.
- **Line 5:** Compile the entire runtime library MyRuntimeLibrary1 which will make the procedures inside that library runnable.
- **Line 6:** Execute the procedure just created.
- **Line 7:** Delete the procedure just created.
- **Lines 9 - 23:** Try to delete erp (mr1::rp) if it wasn't deleted yet.
- **Lines 13 - 20:** Ignore any errors during the deletion except severe internal errors.
- **Line 24:** Mark the error err2 as handled.
- **Lines 25:** Finally notify the application user that something has gone wrong.

Model editing is available from within the language itself; there are intrinsic functions and procedures to view, create, modify, move, rename, compile and delete identifiers. An intrinsic function or procedure that modifies the application is called a Model Edit Function. These functions and procedures reside in the predeclared module ModelEditFunctions with the prefix me. The enumeration below lists the Model Edit Functions and briefly describes them.

*Model Edit
Functions*

Table 33.5 enumerates the Model Edit Functions. A new runtime library can be created using the function me::CreateLibrary. Upon success this function returns the library as an element in AllSymbols. The function me::Create creates a new node or identifier with name name with type type in section ep_sec on position pos. The return value is an element in AllSymbols. When inserting at

*Creating and
deleting*

<pre>me::CreateLibrary(libraryName, prefixName)→AllIdentifiers me::Create(name, newType, parentId, pos)→AllIdentifiers me::Delete(runtimeId)</pre>
<pre>me::ImportLibrary(filename[, password]→AllIdentifiers me::ImportNode(esection, filename[, password]) me::ExportNode(esection, filename[, password])</pre>
<pre>me::Parent(runtimeId)→AllIdentifiers me::Children(runtimeId, runtimeChildren(i))</pre>
<pre>me::ChildTypeAllowed(runtimeId, newType) me::TypeChangeAllowed(runtimeId, newType) me::TypeChange(runtimeId, newType)</pre>
<pre>me::GetAttribute(runtimeId, attr) me::SetAttribute(runtimeId, attr, txt) me::AllowedAttribute(runtimeId, attr)</pre>
<pre>me::Rename(runtimeId, newname) me::Move(runtimeId, parentId, pos)</pre>
<pre>me::IsRunnable(runtimeId)</pre>
<pre>me::Compile(runtimeId)</pre>

Table 33.5: Model Edit Functions for runtime libraries

position i ($i > 0$), the declarations previously at positions $i .. n$ are moved to positions $i + 1 .. n + 1$. When inserting at position 0, the identifier is placed at the end. The procedure `me::Delete` can be used to delete both a runtime library and a runtime identifier in a library. All subnodes of `ep` in the runtime library are also deleted.

The procedure `me::ImportNode` reads a section, module, or library into node `ep`. When `ep` is a runtime library, an entire library is read, replacing the existing prefix. `me::ExportNode` writes the contents of the model editor tree referenced by `ep` to a file. These two procedures use the ASCII `.aim` file format.

Reading and writing

The function `me::Parent(ep)` returns the parent of `ep` or the empty element if `ep` is a root. The function `me::Children(ep, epc(i))` returns the children of `ep` in `epc(i)` whereby i is an index over a subset of Integers.

Inspecting the tree

The function `me::ChildTypeAllowed(ep, et)` returns 1 if an identifier of type `et` is allowed as the child of `ep`. The function `me::TypeChangeAllowed(ep, et)` returns 1 if the identifier `ep` is allowed to change into type `et`. The procedure `me::TypeChange(ep, et)` performs a type change attempting to retain as many attributes as possible.

Node types

The function `me::GetAttribute(ep, attr)` returns the contents of the attribute `attr` of identifier or node `ep`. The complementary procedure `me::SetAttribute`

Attributes

(ep,attr,str) specifies these contents. The function `me::AllowedAttribute(ep,attr)` returns 1 if attribute `attr` of identifier `ep` is allowed to have text.

The procedure `me::Rename(ep, newname)` gives `ep` a new name `newname`. The text inside the library is adapted, but a corresponding entry in the `namechange` file is not created. The procedure `me::Move(ep, ep_p, pos)` moves an identifier from one location to another. When the identifier changes its namespace, this is a change of name, and the text in the runtime library is adapted correspondingly, but an entry in the `namechange` file is not created. Runtime identifiers can't be moved from one runtime library to another.

Changing name or location

The function `me::IsRunnable(ep)` returns 1 if `ep` is inside a successfully compiled runtime library.

Querying runtime library status

The function `me::Compile(ep)` compiles the node `ep` and all its subnodes. If `ep` is the empty element all runtime libraries are compiled. See also the Section [23.4](#) on working with `AllIdentifiers`.

Compilation

To the main application, runtime identifiers are like data. Data operations such as creation, modification, destruction, read and write are also applicable to runtime identifiers. When saving your project, the runtime libraries are **not** saved. Runtime libraries, including the data of runtime identifiers, can be saved in two ways: as separate files or in cases.

Runtime identifiers are like data

The runtime libraries themselves can be saved in `ascii` or `binary` model files using the function `me::ExportNode`. They can subsequently be read back using the functions `me::ImportLibrary` and `me::ImportNode`, see the function reference for more details on these functions. The data of the runtime identifiers can be written using a `write to file` statement and be read back using a `read from file` statement, see also Section [24.1.1](#).

Storing runtime libraries in separate files

When saving a case, a snapshot of the data of a model, or a selection thereof (`casetype`), is saved. The data of a model includes the runtime libraries. However, the names of the runtime identifiers may vary and therefore they cannot be part of a `casetype`. Whether or not runtime libraries are saved in a case is controlled by a global option, named `Case contains runtime libraries`. When loading a case saved with this option switched on, the previously created runtime libraries will be destroyed first and then the stored runtime libraries will be recreated, both structure and data. When loading a case saved while this option was off, or a case saved with AIMMS 3.10 or earlier, any existing runtime libraries will be left intact. Datasets never contain runtime libraries.

Storing runtime libraries in cases

To the AIMMS model explorer the runtime libraries are readonly; it can copy runtime identifiers into the main application, but it can not modify runtime identifiers. Otherwise, if the AIMMS model explorer could modify runtime identifiers the state information maintained by the main application regarding the runtime identifiers might become inconsistent with the actual state of these runtime identifiers.

*The AIMMS
model explorer*

When AIMMS is in developer mode, data pages of the runtime identifiers can be opened, just like data pages of ordinary identifiers. The data of runtime identifiers can also be visualized on the AIMMS pages in the following two ways:

*Visualizing the
data of runtime
identifiers*

- The safest way is to create a subset of `AllIdentifiers` containing the selected runtime identifiers, and use this subset as "implicit identifiers" in a pivot table. When the runtime identifiers supposedly referenced in this set, do not yet exist, they will simply not be displayed.
- The runtime identifiers can also be directly visualized in the other page objects. Care should then be taken that the visualized runtime identifiers are created with the proper index domain before a page is opened containing these identifiers; if an identifier does not exist, a page containing a reference to such an identifier will not open correctly. In order to avoid the inadvertent use of runtime identifiers on pages they are not selectable via point and click in the identifier selector, but the identifier selector accepts them when typed in.

The following limitations apply:

Limitations

- Local declarations are not supported; only global identifiers correspond to elements in `AllIdentifiers`.
- Compound sets are not supported.
- Quantities are not supported.
- The source file, module code and user data attributes are not supported.
- The current maximum number of identifiers is thirty thousand.