AIMMS Language Reference - Model Structure and Modules

AIMMS 4

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

This documentation was typeset by AIMMS B.V. using LATEX and the LUCIDA font family.

Chapter 35

Model Structure and Modules

Model and sections

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

Modules

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

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

Library modules

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

35.1 Introduction

Support for large models

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

The model tree

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

Main model node

All model declarations in an AIMMS model are located underneath the root node of the model tree, the `Model` node. The `Model` node is always present in the **Model Explorer**, even when you start a new AIMMS project, and cannot be deleted. For the `Model` node itself, you can specify several attributes that have a global impact, such as the licensing arrangements for your model, or the unit convention (if any) that is applicable to your application as a whole. The attributes of the `Model` node are discussed in full detail in Section 35.2.

Model sections . . .

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

. . . to structure a model

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

Separate source file

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

`Section` attributes

The attributes of a `Section` node allow you to specify such issues as whether its contents needs to be stored in a separate source file, and if the usage of such a source file needs to be licensed. The attributes of a `Section` node are discussed in full detail in Section 35.3.

Complex modeling environments

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

Need for modularization

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

Collections of functions and procedures

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

Name clashes

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

Modules . . .

A more structural solution to the name clash problem is provided by the concept of *modules* in AIMMS, which allow you to share common model source into multiple models, without the risk of running into name clashes. Modules are inserted into the model tree by means of `Module` nodes. These nodes are essentially `Section` nodes with a separate namespace, along with attributes to manipulate the global model namespace. The attributes of `Module` nodes are discussed in full detail in Section 35.4.

. . . avoid name clashes

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

Dividing a project into sub-projects. . .

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

. . . unsuitable for modules

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

Library projects

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

Library modules

Library modules are the source code modules associated with library projects. They can only be added to your model through the **Library Manager** discussed in Section 3.1 of the User's Guide. AIMMS will insert library modules into the model tree as a separate `LibraryModule` root node. The attributes of `LibraryModule` nodes are discussed in full detail in Section 35.5.

Library interface

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

35.2 `Model` declaration and attributes

`Model` declaration and attributes

The `Model` node defines the root node of the entire model tree associated with an AIMMS modeling project. The attributes of the main `Model` node are listed in Table 35.1. All attributes of the `Model` node have a global impact on the entire modeling project.

Attribute	**Value-type**	**See also page**
`Convention`	*convention*, *element-parameter*	
`Comment`	*comment string*	19

Table 35.1: `Model` attributes

The `Convention` attribute

With the `Convention` attribute you can indicate that all I/O with respect to identifiers in your model is to take place according to the unit convention specified in this attribute. The value of this attribute must be either a direct reference to a convention declared in your model, an element parameter into the set `AllConventions` or a string parameter holding the name of a convention within your model. You can find more detailed information about unit conventions and their usage in Section 32.8

35.3 `Section` declaration and attributes

`Section` declaration and attributes

`Section` nodes provide depth to the model tree, and offer facilities to store parts of your model in separate source files. A `Section` node is always a child of the `Model` node, of another `Section` node, or of a `Module` node. The attributes of `Section` nodes are listed in Table 35.2.

Attribute	**Value-type**	**See also page**
`SourceFile`	*string*	
`Property`	`NoSave`	
`Comment`	*comment string*	19

Table 35.2: `Section` attributes

The SourceFile attribute

With the `SourceFile` attribute you can indicate that the contents of a `Section` node in your model is linked to the specified source file. As a consequence, AIMMS will read the contents of the `Section` node from the specified file during compilation of the model. Any modifications to the part of the model contained in such a `Section` node will also be stored in this source file when you save the model. When you select an existing source file for the `SourceFile` attribute of a `Section` node in the **Model Explorer** (see also Section 4.2 of the User's Guide), any previous contents of that section will be lost.

The Property attribute

In the `property` attribute the `NoSave` property can be specified. When the property `NoSave` is set, none of the identifiers declared in the section will be saved in cases.

Section names as identifier subsets

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

35.4 `Module` declaration and attributes

Module declaration and attributes

`Module` nodes create a subtree of the model tree along with a separate namespace for all identifier declarations in that subtree. Like `Section` nodes, the model contents associated with a `Module` node can be stored in a separate source file. A `Module` node is always a child of the main `Model` node, of a `Section` node, or of another `Module` node. The attributes of `Module` nodes are listed in Table 35.3.

The SourceFile attribute

Like with ordinary `Section` nodes, the contents of a `Module` node can also be stored in a separate source file, dynamically linked into a `Module` node in your model through the use of the `SourceFile` attribute.

Attribute	Value-type	See also page
`SourceFile`	*string*	610
`Property`	`NoSave`	610
`Prefix`	*identifier*	
`Public`	*identifier-list*	
`Protected`	*identifier-list*	
`Comment`	*comment string*	19

Table 35.3: `Module` attributes

Modules and namespaces

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

Nested modules

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

Scoping rules

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

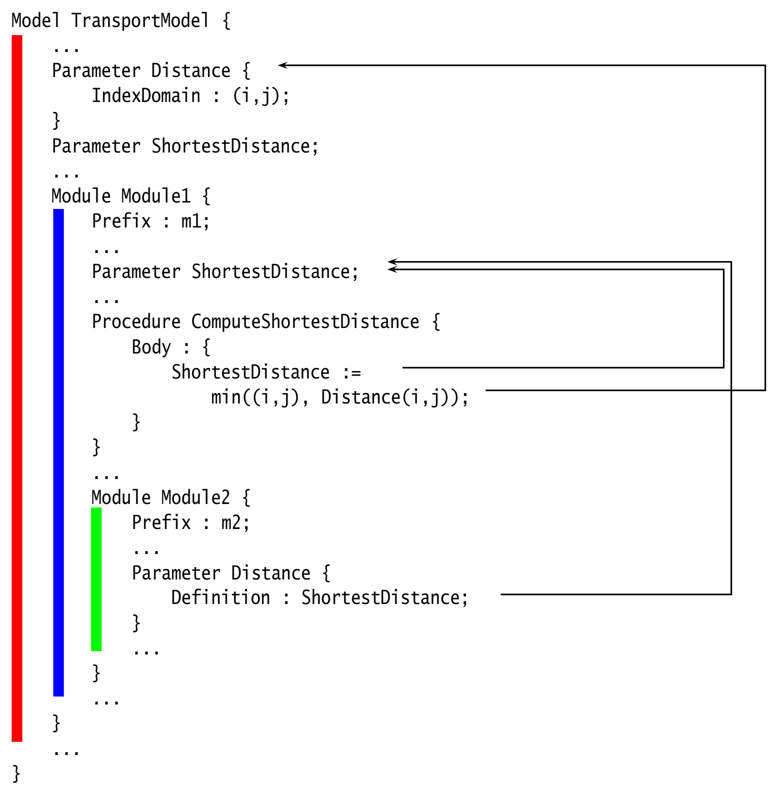
- If the referenced identifier is declared in the namespace associated with the `Module` (or `Model`) in which the procedure, function or identifier is contained, AIMMS will use that particular declaration.
- If the referenced identifier cannot be found, AIMMS will repeatedly search the next higher namespace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

Consequences

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

Example

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



```
Model TransportModel {
    ...
    Parameter Distance {
        IndexDomain : (i,j);
    }
    Parameter ShortestDistance;
    ...
    Module Module1 {
        Prefix : m1;
        ...
        Parameter ShortestDistance;
        ...
        Procedure ComputeShortestDistance {
            Body : {
                ShortestDistance :=
                    min((i,j), Distance(i,j));
            }
        }
        ...
        Module Module2 {
            Prefix : m2;
            ...
            Parameter Distance {
                Definition : ShortestDistance;
            }
            ...
        }
        ...
    }
    ...
}
```

above.

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

Accessing protected identifiers

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

The `Prefix` attribute

With the *mandatory* `Prefix` attribute of a `Module` node, you must specify a module-specific prefix to be used in conjunction with the `::` operator. The value of the `Prefix` attribute should be a unique name within the namespace of the surrounding module (or main model), and will subsequently be added to this namespace. In conjunction with the `::` operator the prefix unambiguously identifies the namespace from which a particular identifier should be taken.

The `::` namespace resolution operator

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

Using global identifiers in `ModuleS`

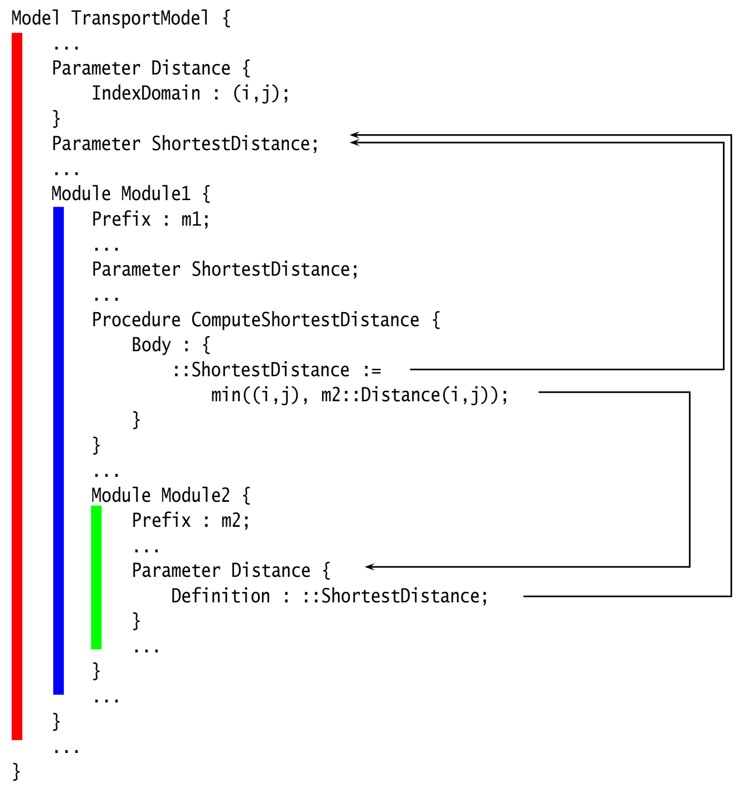
If you want to refer to an identifier in the main model, that is also declared elsewhere along the path from the current module to the main model, you can use the `::` operator *without a prefix*. This indicates to AIMMS that you are interested in an identifier declared in the global namespace associated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

Examples

Consider the model outlined in the example above.

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

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



```
Model TransportModel {
    ...
    Parameter Distance {
        IndexDomain : (i,j);
    }
    Parameter ShortestDistance;
    ...
    Module Module1 {
        Prefix : m1;
        ...
        Parameter ShortestDistance;
        ...
        Procedure ComputeShortestDistance {
            Body : {
                ::ShortestDistance :=
                    min((i,j), m2::Distance(i,j));
            }
        }
        ...
        Module Module2 {
            Prefix : m2;
            ...
            Parameter Distance {
                Definition : ::ShortestDistance;
            }
            ...
        }
        ...
    }
    ...
}
```

The `Public` attribute

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

Example

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

```
Module Module2 {
    Prefix : m2;
    Public : {
        data { Distance }
    }
    ...
```

```
        Parameter Distance {
            Definition : ShortestDistance;
        }
        ...
    }
```

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

Propagation of public identifiers

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

The `Protected` attribute

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

`Public` versus `Protected` responsibilities

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

- The `Public` attribute is intended for the *developer* of a module to define a public interface to the module. If the module is stored in a separate `.amb` file, to be imported by other AIMMS applications, the contents of the `Public` attribute is stored inside the module-specific `.amb` file.
- The `Protected` attribute is intended for the *user* of a module to override the public character of certain identifiers as specified by the developer of the module. As the contents of the `Protected` attribute is not an integral

part of the module, but may be specified differently by every user of the module, it is never stored in a module-specific `.amb` file, but rather in the importing module or main model.

Unique global representation

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

- `Distance(i,j)`
- `ShortestDistance`
- `m1::ShortestDistance`
- `m1::ComputeShortestDistance`
- `m1::m2::Distance`

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

Display and data transfer

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

35.5 `LibraryModule` declaration and attributes

`LibraryModule` declaration and attributes

`LibraryModule` nodes create a separate tree in the model tree along with a separate namespace for all identifier declarations in that subtree. The model contents associated with a `LibraryModule` is always stored in a separate source file. Contrary to `Section` and `Module` nodes, the name of this source file cannot be specified in the `LibraryModule` declaration. Rather, it is specified when you add the library to your project using the **Library Manager**, as discussed in Section 3.1 of the User's Guide. The attributes of `LibraryModule` nodes are listed in Table 35.4.

Library modules and namespaces

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

Attribute	**Value-type**	**See also page**
`Prefix`	*identifier*	
`Interface`	*identifier-list*	
`Property`	`NoSave`	610
`Comment`	*comment string*	19

Table 35.4: `LibraryModule` attributes

The `Interface` *attribute*

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

The `Prefix` *attribute*

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

No propagation to global namespace

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

Library initialization and termination

When creating a new library, AIMMS will automatically add `LibraryInitialization`, `PostLibraryInitialization`, `PreLibraryTermination` and `LibraryTermination` procedures to it. These procedures will be executed during the initialization and termination of your model. The distinction between these steps are explained in more detail in Section 25.1.

35.6 Runtime Libraries and the Model Edit Functions

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

Use case: automating modeling tasks

You may want to improve the maintainability of your application by

- Generating similar statements that act on dynamic selections of identifiers, or

- Generate necessary parameters and database table identifiers with their mapping attributes by querying a relational database schema when setting up a database link with your model.

Use case: Cooperative model development

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

Use case: Proprietary user knowledge

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

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

Use case: ad hoc user queries

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

Runtime editing of identifiers

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

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

Runtime identifiers and libraries

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

Separation between main application and runtime libraries

Model edit functions are only allowed to operate on runtime identifiers. Runtime identifiers exist at runtime but do not yet exist at compile time; the names of runtime identifiers cannot be used directly in the main model. This enforces a separation between identifiers in the main application and runtime identifiers as depicted in Figure 35.1. On the left side of this architecture there is a main application consisting of a main model and zero, one or more libraries. On the right there are zero, one or more runtime libraries. Compilation errors can occur within runtime libraries at runtime. The identifiers inside the main application are not affected by such an error; that is, provided it has local error handling, any procedure inside the main application can continue execution in the presence of compilation errors on identifiers in a runtime library. This is an important advantage of the separation: for several of the use cases presented above, this separation enables continuation in the presence of errors.

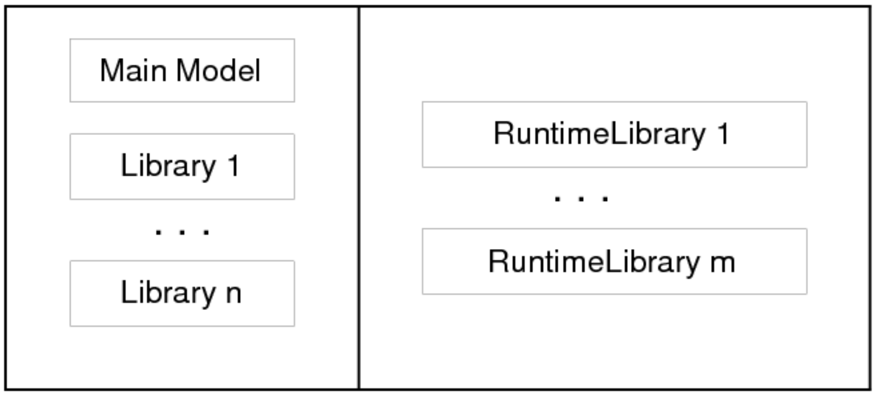


Figure 35.1: Separation between main application and runtime libraries

Example of creating an identifier

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

```
Procedure DisplayDataOfRuntimeIdentifierTabular {
    ElementParameter erp {
        Default    : 'MainExecution';
        Range      : AllIdentifiers;
    }
    StringParameter str;
    ElementParameter err {
        Range      : errh::PendingErrors;
    }
    ElementParameter err2 {
        Range      : errh::PendingErrors;
    }
    Body {
```

```
 1      block
 2          erp := me::Create("rp", 'procedure', 'MyRuntimeLibrary1', 0);
 3          me::SetAttribute(erp, 'body', "display { P } ;");
 4          me::Compile(erp);
 5          me::Compile('MyRuntimeLibrary1');
 6          Apply(erp);
 7          me::Delete(erp);
 8      onerror err do
 9          if erp then
10              block
11                  me::Delete(erp);
12              onerror err2 do
13                  if errh::Severity(err2) = 'Severe' then
14                      DialogMessage(errh::Message(err2) +
15                          "; not prepared to handle severe errors " +
16                          "and halting execution");
17                      halt ;
18                  else
19                      errh::MarkAsHandled(err2) ;
20                  endif ;
21              endblock ;
22              erp := '' ;
23          endif ;
24          errh::MarkAsHandled(err);
25          DialogMessage("Creating and executing rp failed; " + errh::Message(err) );
26      endblock ;
    }
```

A line by line explanation of this example follows below.

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

Model Edit Functions

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

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

Table 35.5: Model Edit Functions for runtime libraries

Creating and deleting

Table 35.5 lists the Model Edit Functions. A new runtime library can be created using the function `me::CreateLibrary`. if successful this function returns the library as an element in `AllSymbols`. The function `me::Create` creates a new node or identifier with name `name` of type `type` in section `ep_sec` at position `pos`. The return value is an element in `AllSymbols`. If inserted at position i ($i > 0$), the declarations previously at positions i .. n are moved to positions $i + 1$ .. $n + 1$. If inserted at position 0, the identifier is placed at the end. The procedure `me::Delete` can be used to delete both a runtime library and a runtime identifier in a library. All subnodes of `ep` in the runtime library are also deleted.

Reading and writing

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

Inspecting the tree

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

Node types

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

Attributes

The function `me::GetAttribute(ep, attr)` returns the contents of the attribute `attr` of identifier or node `ep`. The complementary procedure `me::SetAttribute (ep,attr,str)` specifies these contents. The function `me::AllowedAttribute(ep, attr)` returns 1 if attribute `attr` of identifier `ep` is allowed to have text.

Changing name or location

The procedure `me::Rename(ep, newname)` gives `ep` a new name `newname`. The text inside the library is adapted, but a corresponding entry in the namechange file is not created. The procedure `me::Move(ep, ep_p, pos)` moves an identifier from one location to another. When an identifier changes its namespace, this is a change of name, and the text in the runtime library is adapted correspondingly, but no entry in the namechange file is created. Runtime identifiers can not be moved from one runtime library to another.

Querying runtime library status

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

Compilation

The function `me::Compile(ep)` compiles the node `ep` and all its subnodes. If `ep` is the empty element, all runtime libraries are compiled. See also Section 25.4 on working with `AllIdentifiers`.

Runtime identifiers are like data

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

Storing runtime libraries in separate files

The runtime libraries themselves can be saved in text or binary model files using the function `me::ExportNode`. They can subsequently be read back using the functions `me::ImportLibrary` and `me::ImportNode` (see the function reference for more details on these functions). The data of the runtime identifiers can be written using a `write to file` statement and be read back using a `read from file` statement, see also Section 26.1.1.

Storing runtime libraries in cases

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

The `NoSave` property

When the `NoSave` property is specified for a runtime library, this runtime library will not be saved in cases.

The AIMMS model explorer

To the AIMMS model explorer, the runtime libraries are read only; it can copy runtime identifiers into the main application, but it cannot modify runtime identifiers. This is because, if the AIMMS model explorer could modify runtime identifiers, the state information maintained by the main application regarding the runtime identifiers might become inconsistent with the actual state of these runtime identifiers.

Visualizing the data of runtime identifiers

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

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

Limitations

The following limitations apply:

- Local declarations are not supported; only global identifiers corresponding to elements in `AllIdentifiers`.
- 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.