AIMMS Language Reference - Procedures and Functions

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

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

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

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

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

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

Chapter 10

Procedures and Functions

Functions and procedures

Functions and procedures are pieces of execution code dedicated to a specific task that can be called either from within the graphical end-user interface or from within the model text. Both functions and procedures in AIMMS can have arguments. A function returns either a scalar value or an indexed set of values, and can be used inside expressions. Procedures are more general than functions in that they can have both multiple inputs and outputs. A procedure invocation is a single statement in AIMMS, and can be used to modify the values of global identifiers.

Procedures for initiating execution

Any computation that is part of your application must be started from within a procedure. For simple applications, execution from within the predefined procedure `MainExecution` is usually sufficient to perform all tasks. However, in more complicated applications there are often many entry points, and these can best be implemented as separate procedures.

This chapter

This chapter describes how to construct and use procedures and functions in the AIMMS language. Such procedures and functions are called *internal*. In Chapter 11 you will find additional material on how to link *external* functions and procedures written in FORTRAN and C to your application.

10.1 Internal procedures

AIMMS and internal procedures

Internal procedures are pieces of execution code to perform a dedicated task. For most tasks, and particularly large ones, it is strongly recommended that you use procedures to break your task into smaller, purpose-specific tasks. This provides code structure which is easier to maintain and run. Often it is appropriate to write procedures to obtain input data from users, databases and files, to execute data consistency checks, to perform side computations, to solve a mathematical program, and to create selected reports. Procedures can be called both inside the model text and inside the graphical user interface.

Declaration and attributes

Procedures are added by inserting a special type of node in the model tree. The attributes of a Procedure specify its arguments and execution code. All possible attributes of a Procedure node are given in Table 10.1.

Attribute	Value-type	See also page
Arguments	*argument-list*	
Property	UndoSafe	
Body	*statements*	106
Comment	*comment string*	

Table 10.1: Procedure attributes

Formal arguments

The arguments of a procedure are given as a parenthesized, comma-separated list of formal argument names. These argument names are only the formal identifier names without reference to their index domains. AIMMS allows formal arguments of the following types:

- simple and compound sets, and
- scalar and indexed parameters (either element-valued, string-valued or numerical).

The type and dimension of every formal argument is not part of the argument list, and must be specified as part of the argument's (mandatory) local declaration in a declaration subnode of the procedure.

Interactive support

When you add new formal arguments to a procedure in the AIMMS Model Explorer, AIMMS provides support to automatically add these arguments as local identifiers to the procedure. For all formal arguments which have not yet been declared as local identifiers, AIMMS will pop up a dialog box to let you choose from all supported identifier types. After finishing the dialog box, all new arguments will be added as (scalar) local identifiers of the indicated type. When an argument is indexed, you still need to add the proper IndexDomain manually in the attribute form of the argument declaration.

Range checking

If the declaration of a formal argument of a procedure contains a numerical range, AIMMS will automatically perform a range check on the actual arguments based on the specified range of the formal argument.

Input or output

In the declaration of each argument you can specify its type by setting one of the properties

- Input,
- Output,

- InOut (default), or
- Optional.

AIMMS passes the values of any Input and InOut arguments when entering the procedure, and passes back the values of Output and InOut arguments. For this reason an actual Input argument can be any expression, but actual Output and InOut arguments must be parameter references or set references.

Optional arguments

An argument can be made optional by setting the property Optional in its declaration. Optional arguments are always input, and must be scalar. When an optional argument is not provided in a procedure call, AIMMS will pass its default value as specified in its declaration.

The Body attribute

In the Body attribute you can specify the sequence of AIMMS execution statements that you want to be executed when the procedure is run. All statements in the body of a procedure are executed in their order of appearance.

Example

The following example illustrates the declaration of a simple procedure in AIMMS. The body of the procedure has only been outlined.

```
Procedure ComputeShortestDistance {
    Arguments  : (City, DistanceMatrix, Distance);
    Comment    : {
        "This procedure computes the distance along the shortest path
        from City to any other city j, given DistanceMatrix."
    }
    Body: {
        Distance(j) := DistanceMatrix(City,j);

        for ( j | not Distance(j) ) do
            /*
             *  Compute the shortest path and the corresponding distance
             *  for cities j without a direct connection to City.
             */
        endfor
    }
}
```

The procedure ComputeShortestDistance has three formal arguments, which must be declared in a declaration subnode of the procedure. Their declarations within this subnode could be as follows.

```
ElementParameter City {
    Range        : Cities;
    Property     : Input;
}
Parameter DistanceMatrix {
    IndexDomain  : (i,j);
    Property     : Input;
}
Parameter Distance {
```

```
    IndexDomain  : j;
    Property     : Output;
}
```

From these declarations (and not from the argument list itself) AIMMS can deduce that

- the first actual (input) argument in a call to `ComputeShortestDistance` must be an element of the (global) set `Cities`,
- the second (input) argument must be a two-dimensional parameter over `Cities` × `Cities`, and
- the third (output) arguments must be a one-dimensional parameter over `Cities`.

Arguments declared over global sets

As in the example above, arguments of procedures can be indexed identifiers declared over global sets. An advantage is that no local sets need to be defined. A disadvantage is that the corresponding procedure is not generic. Procedures with arguments declared over global sets are preferred when the procedure is uniquely designed for the application at hand, and direct references to global sets add to the overall understandability and maintainability.

Arguments declared over local sets

The index domain or range of a procedure argument need not always be defined in terms of global sets. Also sets that are declared locally within the procedure can be used as index domain or range of that procedure. When a procedure with such arguments is called, AIMMS will examine the actual arguments, and pass the global domain set to the local set identifier *by reference*. This allows you to implement procedures performing generic functionality for which a priori knowledge of the index domain or range of the arguments is not relevant.

Local sets are read-only

When you pass arguments defined over local sets, AIMMS does not allow you to modify the contents of these local sets during the execution of the procedure. Because such local sets are passed by reference, this will prevent you from inadvertently modifying the contents of the global domain sets. When you do want to modify the contents of the global domain sets, you should pass these sets as explicit arguments as well.

Unit analysis of arguments

Whenever your model contains one or more `Quantity` declarations (see Section 32.2), AIMMS allows you to associate units of measurements with every argument. Similarly as the index domains of multidimensional arguments can be expressed either in terms of global sets, or in terms of local sets that are determined at runtime, the units of measurements of function and procedure arguments can also be expressed either in terms of globally defined units, or in terms of local unit parameters that are determined runtime by AIMMS. The unit analysis of procedure arguments is discussed in full detail in Section 32.4.1.

Local identifiers

Besides the arguments, you can also declare other local scalar or indexed identifiers in a declaration subnode of a procedure or function in AIMMS. Local identifiers cannot have a definition, and their scope is limited to the procedure or function itself.

The property RetainsValue

For each local identifier of a procedure or function that is not a formal argument, you can specify the option RetainsValue. With it you can indicate that such a local identifier must retain its last assigned value between successive calls to that procedure or function. You can use this feature, for instance, to retain local data that must be initialized once and can be used during every subsequent call to the procedure, or to keep track of the number of calls to a procedure.

Execution subnodes

In addition to AIMMS execution statements, you can include references to (named) execution subnodes to the body of a procedure. AIMMS supports several types of execution subnodes. They can either contain just execution statements or provide a graphical input form for complicated statements like the READ, WRITE and SOLVE statement. The contents of the execution subnodes will be expanded by AIMMS into the body of the procedure at the position of their references.

Top-down implementation

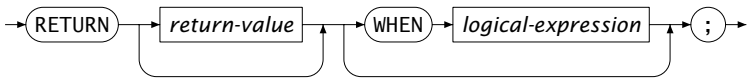
By partitioning the body of a long procedure into several execution subnodes, you can effectively implement the procedure in a self-documenting top-down approach. While the body can just contain the outermost structure of the procedure's execution, the implementation details can be hidden behind subnode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

The RETURN statement

In some situations, you may want to return from a procedure or function before the end of its execution has been reached. You use the RETURN statement for this purpose. It can be subject to a conditional WHEN clause similar to the SKIP and BREAK statements in loops. The syntax follows.

Syntax

return-statement :



Return value

Procedures in AIMMS can have an (integer) return value, which you can pass by means of the RETURN statement. You can use the return value only in a limited sense: you can assign it to a scalar parameter, or use it in a logical condition in, for instance, an IF statement. You cannot use the return value in a compound numerical expression. For more details, refer to Section 10.3.

The Property attribute

In the Property attribute of internal procedures you can specify a single property, UndoSafe. With the UndoSafe property you can indicate that the procedure, when called from a page within the graphical end-user interface of a model, should leave the stack of end-user undo actions intact. Normally, procedure calls made from within the end-user interface will clear the undo stack, because such calls usually make additional modifications to (global) data based on end-user edits.

Procedures summarized

The following list summarizes the main characteristics of AIMMS procedures.

- The arguments of a procedure can be sets, set elements and parameters.
- The arguments, together with their attributes, must be declared in a local declaration subnode.
- The domain and range of indexed arguments can be in terms of either global or local sets.
- Each argument is of type Input, Output, Optional or InOut (default).
- Optional arguments must be scalar, and you must specify a default value. Optional arguments are always of type Input.
- AIMMS performs range checking on the actual arguments at runtime, based on the specified range of the formal arguments.

10.2 Internal functions

Similar to procedures

The specification of a function is very similar to that of a procedure. The following items provide a summary of their similarities.

- Arguments, together with their attributes, must be declared in a local declaration subnode.
- The domain and range of indexed arguments can be in terms of either global or local sets.
- The units of arguments can be expressed in terms of globally defined units of measurement, or in locally defined unit parameters.
- Optional arguments must be scalar, and you must specify a default value.
- AIMMS performs range checking on the actual arguments at runtime.
- Both functions and procedures can have a RETURN statement.

There are differences

There are also differences between a function and a procedure, as summarized below:

- Functions return a result that can be used in numerical expressions. The result can be either scalar-valued or indexed, and can have an associated unit of measurement.
- Functions cannot have side effects either on global identifiers or on their arguments, i.e. every function argument is of type Input by definition.

Not allowed in constraints

AIMMS only allows the (possibly multi-dimensional) result of a function to be used in constraints if none of the function arguments are variables. Allowing function arguments to be variables, would require AIMMS to compute the Jacobian of the function with respect to its variable arguments, which is not a straightforward task. External functions in AIMMS do support variables as arguments (see also Section 11.4).

Example: the Cobb-Douglas function

The Cobb-Douglas (CD) function is a scalar-valued function that is often used in economical models. It has the following form:

$$q = CD_{(a_1,\ldots,a_k)}(c_1,\ldots,c_k) = \prod_f c_f^{a_f},$$

where

q	is the quantity produced,
c_f	is the factor input f,
a_f	is the share parameter satisfying $a_f \geq 0$ and $\sum_f a_f = 1$.

In its simplest form, the declaration of the Cobb-Douglas function could look as follows.

```
Function CobbDouglas {
    Arguments  : (a,c);
    Range      : nonnegative;
    Body       : {
        CobbDouglas := prod[f, c(f)^a(f)]
    }
}
```

The arguments of the CobbDouglas function must be declared in a local declaration subnode. The following declarations describe the arguments.

```
Set InputFactors {
    Index        : f;
}
Parameter a {
    IndexDomain  : f;
}
Parameter c {
    IndexDomain : f;
}
```

Function attributes

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

Attribute	Value-type	See also page
Arguments	*argument-list*	
IndexDomain	*index-domain*	44
Range	*range*	45
Unit	*unit-expression*	47
Property	RetainsValue	
Body	*statements*	106
Comment	*comment string*	

Table 10.2: Function attributes

Returning the result

By providing an index domain to the function, you indicate that the result of the function is multidimensional. Inside the function you can use the function name with its indices as if it were a locally defined parameter. The result of the function must be assigned to this 'parameter'. As a consequence, the body of any function should contain at least one assignment to itself to be useful. Note that the RETURN statement cannot have a return value in the context of a function body.

The Range attribute

Through the Range attribute you can specify in which numerical, set, element or string range the function should assume its result. If the result of the function is numeric and multidimensional, you can specify a range using multidimensional parameters which depend on all or only a subset of the indices specified in the IndexDomain of the function. This is similar as for parameters (see also page 45). Upon return from the function, AIMMS will verify that the function result lies within the specified range.

The Unit attribute

Through the Unit attribute of a function you can associate a unit with the function result. AIMMS will use the unit specified here during the unit consistency check of each assignment to the result parameter within the function body, based on the units of the global identifiers and function arguments that are referenced in the assigned expression. In addition, AIMMS will use the value of the Unit attribute during unit consistency checks of all expressions that contain calls to the function at hand. You can find general information on the use of units in Chapter 32. Section 32.4.1 focusses on unit consistency checking for functions and procedures.

Example: computing the shortest distance

The procedure ComputeShortestDistance discussed in the previous section can also be implemented as a function ShortestDistance, returning an indexed result. In this case, the declaration looks as follows.

```
Function ShortestDistance {
    Arguments    : (City, DistanceMatrix);
    IndexDomain  : j;
    Range        : nonnegative;
    Comment      : {
        "This procedure computes the distance along the shortest path
         from City to any other city j, given DistanceMatrix."
    }
    Body         : {
        ShortestDistance(j) := DistanceMatrix(City,j);

        for ( j | not ShortestDistance(j) ) do
            /*
             *  Compute the shortest path and the corresponding distance
             *  for cities j without a direct connection to City.
             */
        endfor
    }
```

10.3 Calls to procedures and functions

Consistency with prototype

Functions and procedures must be called from within AIMMS in accordance with the prototype as specified in their declaration. For every call to a function or procedure, AIMMS will verify not only the number of arguments, but also whether the arguments and result are consistent with the specified domains and ranges.

Example procedure call

Consider the procedure ComputeShortestDistance defined in Section 10.1. Further assume that DistanceMatrix and ShortestDistanceMatrix are two-dimensional identifiers defined over Cities $\times$ Cities. Then the following assignment illustrates a valid procedure call.

```
for ( i ) do
    ComputeShortestDistance(i, DistanceMatrix, ShortestDistanceMatrix(i,.)) ;
endfor;
```

As you will see later on, the "." notation used in the third argument is a shorthand for the corresponding domain set. In this instance, the corresponding domain set of ShortestDistanceMatrix(i,.) is the set Cities.

Domain checking of arguments

In analyzing the resulting domains of the arguments, AIMMS takes into account the following considerations.

- Due to the surrounding FOR statement the index i is bound, so that the first argument is indeed an element in the set Cities.

- The second argument DistanceMatrix is provided without an explicit domain. AIMMS will interpret this as offering the complete two-dimensional identifier DistanceMatrix. As expected, the argument is defined over Cities × Cities.
- Because of the binding of index i, the third argument ShortestDistanceMatrix(i,.) results into the (expected) one-dimensional slice over the set Cities in which the result of the computation will be stored.

Thus, the domains of the actual arguments coincide with the domains of the formal arguments, and AIMMS can correctly compute the result.

Example function call

Now consider the function ShortestDistance defined in Section 10.1. The following statement is equivalent to the FOR statement of the previous example.

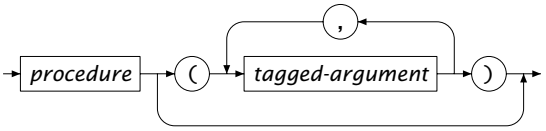
```
ShortestDistanceMatrix(i,j) := ShortestDistance(i, DistanceMatrix)(j) ;
```

In this example index binding takes place through the indexed assignment. Per city i AIMMS will call the function ShortestDistance once, and assign the one-dimensional result (indexed by j) to the one-dimensional slice ShortestDistanceMatrix(i,j).

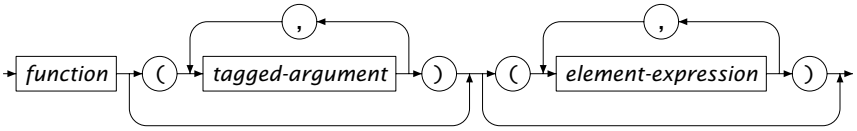
Call syntax

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

procedure-call :



function-call :



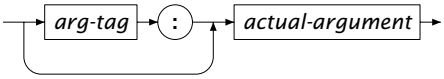
Actual arguments

Each actual argument can be

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

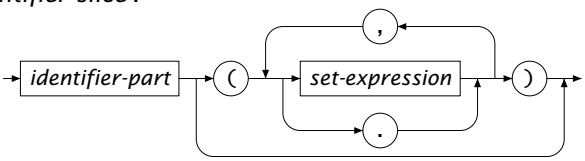
Actual arguments can be tagged with their formal argument name used inside the declaration of the function or procedure. The syntax follows.

tagged-argument :



actual-argument :

identifier-slice :



Scalar and set arguments

For scalar and set arguments that are of type `Input` you can enter any scalar or set expression, respectively. Scalar and set arguments that are of type `InOut` or `Output` must contain a reference to a scalar parameter or set, or to a scalar slice of an indexed parameter or set. The latter is necessary so that AIMMS knows where to store the output value.

No slices of indexed sets

Note that AIMMS does not allow you to pass slices of an indexed set as a set arguments to functions and procedures. If you want to pass the contents of a slice of an indexed set as an argument to a procedure or function, you should assign the contents to a simple (sub)set instead, and pass that set as an argument.

Multi-dimensional arguments

For multidimensional actual arguments AIMMS only allows references to identifiers or slices thereof. Such arguments can be indicated in two manners.

- If you just enter the name of a multidimensional identifier, AIMMS assumes that you want to pass the fully dimensioned data block associated with the identifier.
- If you enter an identifier name plus
  - a "`.`",
  - a set element, or
  - a set expression

  at each position in the index domain of the identifier, AIMMS will pass the corresponding identifier *slice* or *subdomain*.

The "." notation

When passing slices or subdomains of a multidimensional identifier argument, you can use the "`.`" shorthand notation at a particular position in the index domain. With it you indicate that AIMMS should use the corresponding domain set of the identifier at hand at that index position. Recall the argument `ShortestDistanceMatrix(i,.)` in the call to the procedure `ComputeShortestDistance` discussed at the beginning of this section. As the index domain of `Short-`

estDistanceMatrix is the set Cities × Cities, the "." reference stands for a reference to the set Cities.

Slicing

By specifying an explicit set element or an element expression at a certain index position of an actual argument, you will decrease the dimension of the resulting slice by one. The call to the procedure ComputeShortestDistance discussed earlier in this section illustrates an example of an actual argument containing a one-dimensional slice of a two-dimensional parameter.

Dimensions must match

Note that AIMMS requires that the dimensions of the formal and actual arguments match exactly. This implies AIMMS does not allow an actual argument ActualArg(c) where c is a compound index, when the formal argument is defined as FormalArg(i,j), and vice versa, even when c and (i,j) are compatible at the relation level.

Subdomains

By specifying a subset expression at a particular index position of an indexed argument, you indicate to AIMMS that the procedure or function should only consider the argument as defined over this subdomain.

Example

Consider the Cobb-Douglas function discussed in the previous section, and assume the existence of a parameter a(f) and a parameter c(f), both defined over a set Factors. Then the statement

```
Result := CobbDouglas(a,c) ;
```

will compute the result by taking the product of exponents over all factors f. If SubFactors is a subset of Factors, satisfying the condition on the share parameter a(f), then the following call will compute the result by only taking the product over factors f in the subset SubFactors.

```
Result := CobbDouglas( a(SubFactors), c(SubFactors) );
```

Global subdomains

Whenever a formal argument refers to an indexed identifier defined over global sets, it could be that an actual argument in a function or procedure call refers to an identifier defined over a superset of one or more of these global sets. In this case, AIMMS will automatically restrict the domain of the actual argument to the domain of the formal argument. Likewise, if an index set of an actual argument is a real subset of the corresponding global index set of a formal argument, the values of the formal argument, when referred to from within the body of the procedure, will assume the default value of the formal argument in the complement of the index (sub)set of actual argument.

Local subdomains

Whenever a formal argument refers to an indexed identifier defined over local sets, the domain of the actual argument can be further restricted to a subdomain as in the example above. In any case, the (sub)domain of the actual argument determines the contents of the local set(s) used in the formal arguments. Note that consistency in the specified domains of the actual arguments is required when a local set is used in the index domain of several formal arguments.

Tagging arguments

In order to improve the understandability of calls to procedures and functions the actual arguments in a reference may be tagged with the formal argument names used in the declaration. In a procedure reference, it is mandatory to tag all *optional* arguments which do not occur in their natural order.

Permuting tagged arguments

Tagged arguments may be inserted at any position in the argument list, because AIMMS can determine their actual position based on the tag. The non-tagged arguments must keep their relative position, and will be intertwined with the (permuted) tagged arguments to form the complete argument list.

Example

The following permuted call to the procedure `ComputeShortestDistance` illustrates the use of tags.

```
for ( i ) do
    ComputeShortestDistance( Distance       : ShortestDistanceMatrix(i,.),
                             DistanceMatrix : DistanceMatrix,
                             City           : i );
endfor;
```

Using the return value

As indicated in Section 10.1 procedures in AIMMS can return with an integer return value. Its use is limited to two situations.

- You can assign the return value of a procedure to a scalar parameter in the calling procedure. However, a procedure call can never be part of a numerical expression.
- You can use the return value in a logical condition in, for instance, an `IF` statement to terminate the execution when a procedure returns with an error condition.

You can use a procedure just as a single statement and ignore the return value, or use the return value as described above. In the latter case, AIMMS will first execute the procedure, and subsequently use the return value as indicated.

Example

Assume the existence of a procedure `AskForUserInputs(Inputs,Outputs)` which presents a dialog box to the user, passes the results to the `Outputs` argument, and returns with a nonzero value when the user has pressed the OK button in the dialog box. Then the following `IF` statement illustrates a valid use of the return value.

```
if ( AskForUserInputs( Inputs, Outputs ) )
then
   ... /* Take appropriate action to process user inputs */
else
   ... /* Take actions to process invalid user input */
endif ;
```

10.3.1 The APPLY operator

Data-driven procedures

In many real-life applications the exact nature of a specific type of computation may heavily depend on particular characteristics of its input data. To accommodate such data-driven computations, AIMMS offers the APPLY operator which can be used to dynamically select a procedure or function of a given prototype to perform a particular computation. The following two examples give you some feeling of the possible uses.

Example: processing events

In event-based applications many different types of events may exist, each of which may require an event-type specific sequence of actions to process it. For instance, a ship arrival event should be treated differently from an event representing a pipeline batch, or an event representing a batch feeding a crude distiller unit. Ideally, such event-specific actions should be modeled as a separate procedure for each event type.

Example: product blending

A common action in the oil-processing industry is the blending of crudes and intermediate products. During this process certain material properties are monitored, and their computation for a blend require a property-specific *blending rule*. For instance, the sulphur content of a mixture may blend linearly in weight, while for density the reciprocal density values blend linear in weight. Ideally, each blending rule should be implemented as a separate procedure or function.

The APPLY operator

With the APPLY operator you can dynamically select a procedure or function to be called. The first argument of the APPLY operator must be the name of the procedure or function that you want to call. If the called procedure or function has arguments itself, these must be added as the second and further arguments to the APPLY operator. In case of an indexed-valued function, you can add indexing to the APPLY operator as if it were a function call.

Requirements

In order to allow AIMMS to perform the necessary dynamic type checking for the APPLY operator, certain requirements must be met:

- the first argument of the APPLY operator must be a reference to a string parameter or to an element parameter into the set AllIdentifiers,
- this element parameter must have a Default value, which is the name of an existing procedure or function in your model, and

- all other values that this string or element parameter assumes must be existing procedures or functions with the same prototype as its `Default` value.

Example: processing events elaborated

Consider a set of `Events` with an index `e` and an element parameter named `CurrentEvent`. Assume that each event `e` has been assigned an event type from a set `EventTypes`, and that an event handler is defined for each event type. It is further assumed that the event handler of a particular event type takes the appropriate actions for that type. The following declarations illustrates this set up.

```
ElementParameter EventType {
    IndexDomain    : e;
    Range          : EventTypes;
}
ElementParameter EventHandler {
    IndexDomain    : et in EventTypes;
    Range          : AllIdentifiers;
    Default        : NoEventHandlerSelected;
    InitialData    : {
          DATA { ShipArrivalEvent    : DischargeShip,
                 PipelineEvent       : PumpoverPipelineBatch,
                 CrudeDistillerEvent : CrudeDistillerBatch }
    }
}
```

The `Default` value of the parameter `EventHandler(et)`, as well as all of the values assigned in the `InitialData` attribute, must be valid procedure names in the model, each having the same prototype. In this example, it is assumed that the procedures `NoEventHandlerSelected`, `DischargeShip`, `PumpoverPipelineBatch`, and `CrudeDistillerBatch` all have two arguments, the first being an element of a set `Events`, and the second being the time at which the event has to commence. Then the following call to the `APPLY` statement implements the call to an event type specific event handler for a particular event `CurrentEvent` at time `NewEventTime`.

```
Apply( EventHandler(EventType(CurrentEvent)), CurrentEvent, NewEventTime );
```

When no event handler for a particular event type has been provided, the default procedure `NoEventHandlerSelected` is run which can abort with an appropriate error message.

Use in constraints

When applied to functions, you can also use the `APPLY` operator inside constraints. This allows you, for instance, to provide a generic constraint where the individual terms depend on the value of set elements in the domain of the constraint. Note, that such use of the `APPLY` operator will only work in conjunction with external functions, which allow the use of variable arguments (see Section 11.4).

Example: product blending

Consider a set of Products with index p, and a set of monitored Properties with index q. With each property q a blend rule function can be associated such that the resulting values blend linear in weight. These property-dependent functions can be expressed by the element parameter BlendRule(q) given by

```
ElementParameter BlendRule {
    IndexDomain    : q;
    Range          : AllIdentifiers;
    Default        : BlendLinear;
    InitialData    : {
          DATA { Sulphur    : BlendLinear,
                 Density    : BlendReciprocal,
                 Viscosity  : BlendViscosity   }
    }
}
```

Thus, the computation of the property values of a product blend can be expressed by the following single constraint, which takes into account the differing blend rules for all properties.

```
Constraint ComputeBlendProperty {
    IndexDomain    : q;
    Definition     : {
        Sum[p, ProductAmount(p)  * Apply(BlendRule(q), ProductProperty(p,q))] =
        Sum[p, ProductAmount(p)] * Apply(BlendRule(q), BlendProperty(q))
    }
}
```

Depending on the precise computation in the blend rules functions for every property q, the APPLY operator may result in linear or nonlinear terms being added to the constraint.