ISA Dialog Manager

User-defined Attributes and Methods

A.06.03.b

This manual explains the use of user-defined attributes and methods.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Germany

Notation Conventions

DM will be used as a synonym for Dialog Manager.

The notion of UNIX in general comprises all supported UNIX derivates, otherwise it will be explicitly stated.

< >	to be substituted by the corresponding value
color	keyword
.bgc	attribute
{ }	optional (0 or once)
[ ]	optional (0 or n-times)
<A> | <B>	either <A> or <B>

Description Mode

All keywords are bold and underlined, e.g.

variable **integer** **function**

Indexing of Attributes

Syntax for indexed attributes:

[I]

[I,J] meaning [row, column]

Identifiers

Identifiers have to begin with an uppercase letter or an underline ('_'). The following characters may be uppercase or lowercase letters, digits, or underlines.

Hyphens ('-') are ***not*** permitted as characters for specifying identifiers.

The maximal length of an identifier is *31* characters.

Description of the permitted identifiers in the Backus-Naur form (BNF)

<identifier>	::=	<first character>{<character>}
<first character>	::=	_ | <uppercase>
<character>	::=	_ | <lowercase> | <uppercase> | <digit>

<digit> ::= 1 | 2 | 3 | … 9 | 0

<lowercase> ::= a | b | c | … x | y | z

<uppercase> ::= A | B | C | … X | Y | Z

Table of Contents

1 Introduction

In this manual you will find a summary of the definition of Dialog Manager

- user-defined attributes
- user-defined methods

User-defined Attributes

User-defined attributes serve the definition of application-specific information. They help separate the dialog from the application, i.e. the application does not need any information about the dialog any more.

User-defined Methods

User-defined methods are used to define application-specific information. With these methods you can program a dialog "object-oriented".

2 User-defined Attributes

In this chapter we describe methods to separate dialog and application completely, i.e. the application does not need any information about the dialog. (See chapter "Using user-defined attributes and object-oriented programming" in "Programming Techniques")

The existing limit of 16 parameters per application is still valid. However each parameter can be a structure of its own.

On the one hand, the attribute *.userdata* provided by the Dialog Manager for each object could be used to define application-specific information. For "normal" objects, like e.g. statictext, this attribute can be used only once per object. For objects with listing character (e.g. listbox, poptext, tablefield), it can be used per item/entry.

On the other hand, **user-defined attributes** have been made available. These attributes are treated in the same way as internal DM attributes, i.e. they can also be inherited from a model or a default. There are the following possibilities to define these attributes:

- scalar attributes
- indexed attributes
- attributes referencing to other attributes (shadow attributes)

2.1 Definition of User-defined Attributes

Syntax

Scalar Attributes

```
<data type> <attribute name> { := <value> } ;
```

Indexed Attributes

```
<data type> <attribute name> [ <integer value> ] ;
```

shadow Attributes

```
<data type> <attribute name> shadows { instance }
    <object> <attribute> { [ <index> ] } ;
```

<data type>
One of the data types available in the IDM, e.g. *integer*, *object*, *string*, *boolean*… can be used.

<attribute name>
The name of the attribute, it can be arbitrarily chosen by the user. The name has to correspond to the DM name convention for identifiers (uppercase letters at the beginning, etc.)

Example

» Declaration without point:

```
pushbutton MyButton
{
  integer Position;
}
```

» Referencing with point:

```
MyButton.Position := 1;
```

<value>
Value that corresponds to the data type defined before.

<integer value>
Number which defines the field size.

An attribute can be defined with the keyword **shadows**. It does not memorize its value, but accesses (by reading and writing) the following indicated object attribute. When the additional keyword **instance** is used, this means when the model is instantiated, the reference is set to the newly created instance and does not stay on the model.

<object>
Object which the attribute accesses.

<attribute>
Attribute of the object which the attribute accesses.

<index>
Indicates rows and columns for list objects and their attributes.

Examples

» Definition of a ***pushbutton*** Model with a user-defined attribute “Position” which will be inherited by all instances with the initial value *17*.

```
model pushbutton MyModel
{
  integer Position := 17;
}
```

» Definition of a ***pushbutton*** Model with an indexed, user-defined attribute(array) “Active”, which can contain *5* items of the data type *boolean*.

```
model pushbutton MyModel
{
  boolean Active[5];
}
```

» Definition of a ***pushbutton*** Model whose user-defined *string* attribute “CurrentValue” gets its value from the contents of the ***edittextes*** “Et”.

```
model pushbutton MyModel
{
  string CurrentValue shadows Et.content;
}
```

» Definition of a ***window*** Model with a user defined attribute “CurrentValue”, which refers to the instance of ***edittext*** “Et” when the Model is instantiated.

```
model window MyModel
{
  string CurrentValue shadows instance Et.content;
  child edittext Et
  {
  }
}
```

2.2 Access to User-defined Attributes

The access to user-defined attributes is the same as the access on internal DM attributes.

Example

```
MyModel.Position := 28;
```

or

```
MyModel.Active[0] := true;
```

or

```
MyModel.Currentvalue := "NewContent"
```

This assignment actually changes the contents of the shadow object (compare above edittext).

2.3 Attributes of User-defined Attributes

User-defined attributes have attributes of their own which can be queried and changed in the rules.

With the following constructions, you can access and change user-defined attributes during runtime.

» *.type[attributename]*
returns the data type of the indicated attribute

» *.count[attributename]*
for indexed attributes .count contains the actual size of the given attribute (= 0 with non-indexed attributes and shadow attributes)

» *.shadowobject[attributename]*

provides the referenced object for shadow attributes

» *.shadowindex[attributename]*
provides the index of the referenced attribute for shadow attributes

» *.shadowattr[attributename]*
provides the attribute of the referenced attribute for shadow attributes

Example

At the instance of a hierarchical model, the attributes of a record are to be set to the instance objects

```
model window WModel
{
   child edittext ET1
   {   .ytop 1;
      .xleft 1;
      .format "NNNN";
   }
   child edittext ET2
   {
      .ytop 3;
      .xleft 1;
   }
   record WData
   {
      string [20]   Value1;
      string [20]   Value2;
   }
}
```

The generating of instances and the conversion of record attributes in the rules can be done as follows:

```
rule void CreateInstance ( )
{
   variable object Obj;

   Obj:= create (WModel, WModel.module);
   if   (Obj <> null) then
      Obj.WData.shadowobject [.Value1]
         := Obj.ET1;
      Obj.WData.shadowattr [.Value1]
         := .content;
      OBj.WData.shadowobject [.Value2]
         := Obj.ET2;
      Obj.WData.shadowattr [.Value2]
         := .content;
   endif
}
```

User-defined attributes can be created also dynamically by setting the attribute type, e.g.

Declaration:

```
listbox LBFiles
{
  .ytop 1;
  .yleft 1;
}
```

Subsequent generating in a rule:

```
LBFiles.type[".CurrentFile"] := string;
```

This corresponds to the following declaration:

```
listbox LBFiles
{
  .ytop 1;
  .yleft 1;
  string CurrentFile
}
```

2.4 Associative Arrays

The dialog designer may indicate his own objects, so-called user-defined objects, for each object. These can be used to deposit state information or other data directly at the object. Dialog Manager offers the possibility to use simple attributes, field attributes and associative arrays.

Associative arrays are described below in more detail and examples illustrate the application possibilities.

2.4.1 Associations and Associative Arrays

An association is an imaginary connection between things or activities which actually have nothing in common.

A good example illustrating an association is a TV station. With "MTV", a viewer would associate "Music", a jurist "Commercial TV", an advertising man "Quota" and a television engineer the channel "65", for example. Accordingly these associations may be programmed in a program in the way that the television engineer no longer needs to memorize the channel, but "MTV" only. The program makes work much easier and time-saving for him. The program must take over the conversion, provided that the associations can be programmed. Dialog Manager supports the dialog programmer by handing him over an enormous structuring element - the associative arrays. In general an association is directed, i.e. it is not easily reversible. A television engineer would rather associate "65" with pension than with a TV channel, or a jurist would rather associate "Commercial TV" with a lot of work and contracts than with "MTV". In the Dialog Manager, a single association can be easily defined, even with normal attributes, or has to be defined as unnecessary, since these associations are usually programmed directly into the code. A television engineer, for example, would associate certain

channels with other TV stations. For such groups or arrays of similar associations the Dialog Manager offers the means of associative arrays. This relationship might also be understood as a function: The source of the association, for example "MTV", is given as parameter and, as a result, channel 65 will be received. Programming this function takes a lot of time, as it is necessary to program a "case" for all values in a rule; and the function must cover the possible range of values or have static associations only. In contrast to associative arrays, maintenance consumes too much time. These arrays or fields may be indexed with the source of the association, the contents of this field being the corresponding end. In the channel example the array was indexed with the string "MTV" having returned the result 65, which is the content of the array.

Dialog Manager may define and use such associative arrays for any object (window, poptext, pushbutton, as well as dialog, module or application). These associative arrays may be defined almost in the same way normal arrays of user-defined attributes are defined. The index, however, is indicated as data type instead of array.

Associative arrays are defined as follows:

```
<DM Datatype 1> <Attribute Name> [ <DM Datatype 2> ];
.<Attribute Name>[ <Value 2> ] := <Value 1>;
```

Example

```
window WMain
{
integer channel [ string ];
  .channel [ "MTV" ] := 65;
  .channel [ "CNN" ] := 14;
  .channel [ "NBC" ] := 11;
}
```

In this example the window has an associative array in which the relevant channels for different strings (the stations) are saved.

The example above can be extended to a functional dialog.

```
dialog StationChoice

window WMain
{
integer channel [ string ];
    . channel [ "MTV" ] := 65;
    . channel [ "CNN" ] := 14;
    . channel [ "NBC" ] := 11;
          child poptext Station
    {
          .text [ 1 ]  "MTV";
          .text [ 2 ]  "CNN";
          .text [ 3 ] "NBC";
         on select
    {
```

```
          print WMain. Channel [ this.content ];
    }
    }
          on close { exit(); }
}
```

If the user selects a TV station from the poptext, the currently selected string from the poptext (this.-content) will be used as index for the associative array Channel. This is how one gets the corresponding or associated channel.

Please note that there is **no** order in the associative arrays. Ordering is often not possible or difficult to calculate. In the example above the stations are not ordered.

2.4.2 Dynamic Administration

Dialog Manager manages the complete administration of associative arrays. To generate a new item it is not necessary to amplify the array or to do other administration work. One just needs to assign the content to a field. If the index is not available yet, the field will be generated automatically, i.e. there is no need to call an :insert() method.

The following example illustrates a stack which is able to save any kind of Dialog Manager data (anyvalue):

```
dialog StackAdministration
record Stack
{
      integer Top := 0;
      anyvalue  Entry[ integer ];
      object Push := Rule_Push;
      object Pop := Rule_Pop;
}
rule void Rule_Push ( anyvalue Value)
{
      Stack.Entry[Stack.Top] := Value;
      Stack.Top := Stack.Top + 1;
}
rule anyvalue Rule_Pop ()
{
       if Stack.Top = 0
       then
              !! Error, stack is empty
        else
              Stack.Top := Stack.Top - 1;
        endif
        return (Stack.Entry[ Stack.Top ]);
}
on dialog Start
{
```

```
    Stack.Push( "Item" );    !! "Item"
    Stack.Push( "String" );  !! "Item" - "String"
    Stack.Push( 42 );        !! "Item" - "String" - 42
    Stack.Push( .visible );  !! "Item" - "String" - 42 - .visible
    print Stack.Pop();       !! "Item" - "String" - 42
    Stack.Push( .width );    !! "Item" - "String" - 42 - .width
    print Stack.Pop();       !! "Item" - "String" - 42
    print Stack.Pop();       !! "Item" - "String"
    print Stack.Pop();       !! "Item"
}
```

By using this functional example, an individual stack administration can easily be constructed. Line 11 shows how a new item is generated without having to do any administration work. The example above does not index with strings or construct associations, but uses integer numbers. Here, not the natural order of numbers is used, but for certain numbers a relation or association is built up. The example can be modified in the way that it is counted in different step sizes instead in one-by-one steps. The behavior does not change.

2.5 Working with Associative Arrays

It is often necessary to process asssociative arrays entirely or to delete individual associations which are no longer needed. With help of the attribute .itemcount it is possible to query the number of available items.

```
window WMain
{
integer channel [ string ];
  .channel [ "MTV" ] := 65;
  .channel [ "CNN" ] := 14;
  .channel [ "NBC" ] := 11;
}
rule integer StationNumber()
{
  return ( WMain.itemcount[ .channel ]);  !! in example, 3
}
```

2.5.1 Method index

It is also possible to query the internal order of associative arrays. Dialog Manager uses the internal structure for administration purposes only, in order to run through its associative array.

Attention

The internal structure of associate arrays is used for the internal administration only and may change. If you need structured data, which means that the associative arrays have been misused, you must calculate it yourself. In this case, however, we recommend using normal arrays.

Dialog Manager offers the possibility to run through an associative array entirely. Please do note that you may calculate the index from the internal structure by using the method :index().

```
<Object>:index ( attribute <Name>, integer <InnerIndex> );
```

<Object> is the object for which the array <Name> is defined. <InnerIndex> is the index specifying the internal structure which is defined with integer numbers from 1 up to the number of items.

Example

```
window WMain
{
  integer channel [ string ];
  .channel [ "MTV" ] := 65;
  .channel [ "CNN" ] := 14;
  .channel [ "NBC" ] := 11;
}
rule void PrintStation()
{
  variable integer I;
  for I : = 1 to WMain.itemcount[ .channel ]
  do
    print WMain.channel[ WMain:index(.channel, I ) ];
  endfor
}
```

In row 11 the loop is run through from 1 up to the number of items according to the internal structure. In row 13 the index is calculated (WMain:index(.channel, I)) which indexes the associative array .channel.

2.5.2 Method delete

To delete any number of items, the user may apply the method :delete().

```
<Object>:delete( attribute <Name>, anyvalue <Index>);
```

In this case <Object> will be the object, having defined the array <Name>. <Index> indicates the item to be deleted.

Example

```
window WMain
{
  integer channel [ string ];
  .channel [ "MTV" ] := 65;
  .channel [ "CNN" ] := 14;
  .channel [ "NBC" ] := 11;
}
rule void DeleteStation()
{
```

```
  variable integer I;
  for I : = WMain.itemcount[ .channel ] to 1 step -1
  do
    WMain:delete( .channel, WMain:index(.channel, I ) );
  endfor
}
!! Second, more elegantly programmed deletion routine
rule void DeleteStation_2()
{
  while (WMain.itemcount > 0)
  do
    WMain:delete( .channel, WMain:index(.channel, 1 ) );
  end
}
```

Both rules have an identical function. The first rule makes use of the internal structure and runs through the loop from the start to end. As deletion naturally changes the internal structure the items will be deleted from behind here. As long as there are items available the first item will correspond to the internal structure. The second rule which does always delete the frist item profits from this.

2.6 Methods for Arrays of User-defined Attributes

The methods **:clear()**, **:delete()**, **:exchange()**, **:insert()** and **:move()** are used for non-associative arrays of user-defined attributes and will be described in more detail here.

2.6.1 :clear()

This method deletes the contents of elements in fields (indexed, non-associative, user-defined attributes).

In contrast to the **:delete()** method, **:clear()** only deletes the contents of the items. The items themselves remain as “empty” items, which will have the default value if available. Thus **:clear()** will not reduce the number of items.

Particularity

The **:clear()** method can also be used to delete **all** items of an **associative** array. In this case, neither the *Start* nor the *Count* parameter may be specified. Therefore, **:clear()** cannot be used to delete single items from an associative array. Additionally, not only the contents but the elements themselves are deleted in this case. Thus, the number of elements afterwards will be *0*.

Definition

```
boolean :clear
(
      attribute Attr input
  { , integer Start input }
```

```
  { , integer Count input }
)
```

Parameters

attribute *Attr* input

This parameter defines the user-defined attribute on which the method should be applied.

*integer **Start** input*

This parameter defines the position from which the contents of the items should be deleted.
The value range for *Start* is *0 … .count[Attr]*, i.e. the default value can also be deleted.
If neither *Start* nor *Count* is specified, the contents of all elements are deleted.

*integer **Count** input*

This optional parameter defines the number of items whose contents are to be deleted.
If neither *Start* nor *Count* is specified, the contents of all elements are deleted.
If *Start* is specified, the default value of *Count* is *1*.

Return value

The method returns *true* if the contents of the items could be deleted.

If an error has occurred when deleting the contents, the method will return a `fail`.

2.6.2 :delete()

This method deletes items from arrays (indexed user-defined attributes).

Unlike the **:clear()** method, **:delete()** completely deletes the elements, i.e. the number of elements is reduced.

Definition

```
boolean :delete
(
      attribute Attr input,
      anyvalue Position input
  { , integer Count := 1 input }
)
```

Parameters

attribute *Attr* input

This parameter defines the user-defined attribute on which the method should be applied.

anyvalue *Position* input

In this parameter the index of the first item to be deleted is passed. For non-associative arrays *Position* has to be of the data type *integer*, for associative arrays it has to be of the data type the associative array is indexed with.

The value range for *Position* with non-associative fields is *0 … .count[Attr]*, i.e. the default value can also be deleted. If the default value is deleted, the value of the first item after the deleted items becomes the default value.

integer ***Count*** *:= 1 input*

This optional parameter defines the number of items that are deleted. If the parameter is not specified, *1* is taken as default.
Count cannot be used with associative arrays, only *1* item can be deleted at once.

Return value

The method returns *true* if the rows or columns could be deleted.

If an error has occurred when deleting, the method will return a `fail`.

2.6.3 :exchange()

This method exchanges two items of an array (indexed, non-associative, user-defined attribute).

Definition

```
boolean :exchange
(
  attribute Attr input,
  integer Position1 input,
  integer Position2 input
)
```

Parameters

attribute *Attr* input

This parameter defines the user-defined attribute on which the method should be applied.

integer *Position1* input
integer *Position2* input

In these parameters the indices of the two items to be exchanged with each other are passed. The value range for both parameters is *0 … .count[Attr]*, i.e. the default value can also be exchanged with the value of another item.

Return value

The method returns *true* if the items could be exchanged.

If an error has occurred when exchanging, the method will return a `fail`.

2.6.4 :insert()

This method inserts new items into arrays (indexed, non-associative, user-defined attributes). The newly inserted items are “empty” or have the default value if available.

Definition

```
boolean :insert
(
      attribute Attr input,
      integer Position input
  { , integer Count := 1 input }
)
```

Parameters

attribute *Attr* input
This parameter defines the user-defined attribute on which the method should be applied.

integer *Position* input
In this parameter the index where the new elements should be inserted is passed. If its value is 0, the elements are appended to the end.

*integer **Count** := 1 input*
This optional parameter defines the number of items that are inserted. If the parameter is not specified, *1* is taken as default.

Return value

The method returns *true* if the items could be inserted.

If an error has occurred when inserting, the method will return a `fail`.

2.6.5 :move()

This method moves items of an array (indexed, non-associative, user-defined attribute) to another position in the array.

Definition

```
void :move
(
  attribute Attr input,
  integer Position input,
  integer Target input,
  integer Count input
)
```

Parameters

attribute *Attr* input
This parameter defines the user-defined attribute on which the method should be applied.

integer *Position* input

This parameter specifies the index of the first item to be moved.

integer *Target* input

This parameter determines where the items should be moved to.

integer *Count* input

This parameter defines how many items are moved.

Return value

None.

3 User-defined Methods

Methods are rules at the object. They can be defined there in the same way as named rules in the dialog. These methods are called in a similar way as predefined methods.

Example

```
dialog Example
window Win
{
  child statictext Stext
  {
    !! definition and declaration of method
    rule void PrintEvent(string S)
    {
      this.text := "Win was " + S;
    }
  }
  on move
  {
    !! call of method
    Stext:PrintEvent("moved");
  }
  on resize
  {
    Stext:PrintEvent("resized");
  }
}
```

3.1 Object this

So far the keyword this has had the following meaning in the Rule Language: representation of object which has triggered the rule processing. This meaning has not been changed. Within methods, however, the meaning has been changed. In "this" the object, with which the method has been called, is saved. If it is necessary in methods to access also the object at which the triggering event (not the method!) occurred you can query this with the object thisevent.object.

In the dialog example above it is implicitly shown that in the method already "this" is the "statictext Stext" and not "window Win" where the event occurred. If another method is called in a method the allocation of "this" will be changed accordingly.

Note

Rules that are defined at the dialog are not dialog methods. The meaning of existing dialogs is thus not changed.

3.2 Models and Methods

Dialog Manager supports methods in models and also for inheritance, i.e. methods can be defined at the model or at the default. This possibility is an additional efficient instrument for developing. Instances of the models can access and use these model methods dynamically

```
dialog Example
model statictext Mstatic
{
  rule void PrintEvent (string S)
  {
    this.text := "Win was " + S;
  }
}
window Win
{
  child Mstatic S1 {}
  child Mstatic S2 {}
  on move
  {
    !! call of method
    S1:PrintEvent("moved");
  }
  on resize
  {
    S2:PrintEvent("resized");
  }
}
```

The above example shows that the method was defined at the model Mstatic. You may use the object "this" in the same way as in known event rules. "this" will always be allocated with the corresponding object with which the method actually has been called. In the example at the event move of window Win "this" has the value S2. With this mechanism general rules or better methods which may be fixed at the model can be easily defined.

3.3 Calling Model Methods

The problem with methods is that general methods are formulated at the default or at superordinate models. At subordinated objects these methods will then be specialised. To avoid code doubling the methods of superordinate models can also be used, i.e. they can be called by hierarchically sub-ordinate methods. This, however, is not done automatically as it is the case with event rules (with before and after), which can control this automatism. The method :super which calls the cor-responding model method has made available.

```
object:super( [Parameter ...] );
```

Example

```
dialog Example

model window MWin
{
  integer Status := 0;
  rule void CleanAfterClose()
  {
    !! Reset attribute
    this.Status := 0;
  }

  !! Both rules react when the window is getting
  !! invisible and will trigger centrally
  !! "clear rule".
  on close before
  {
    this:CleanAfterClose();
  }
  on .visible changed before
  {
    if this.visible = false then
      this:CleanAfterClose();
    endif
  }
}
model MWin MMainWin
{
  integer MainStatus := 0;
  rule void CleanAfterClose()
  {
    !! Reset only those attributes which
    !! have been defined here.
    this.MainStatus := 0;
    !! Clear remaining attributes.
    this:super();
  }
}
MMainWin MainWin
{
}
```

This example shows how certain objects are reset after closing the window. The methods are directly at the object which the attributes have been defined, i.e. the information is saved centrally at the object. The order of the method processing has to be set in the model hierarchy with the method :super.

The Dialog Manager is not able to foresee which conditions are to be complied with in order to call the method of the underlying model. this:super() always searches the closest method in the model hierarchy of the current object.

3.4 Indirect Method Calling

Dialog Manager has the data type method. In variables or attributes you can save and call methods or better method names. The same problem occurs with attributes. If an attribute is saved in a variable, it is not enough to write object.variable. The Dialog Manager is not able to recognize whether the variable name is also an attribute of this object. For attributes this problem can be solved via getvalue (Object,Variable). The same problem occurs also with methods. The Dialog Manager cannot interpret Object:Variable due to possible ambiguities. This is why there is a ":call" builtin method available.

The call is

```
Object:call(<Method>, [Parameter...]);
```

If a method has been saved in a variable this method can be called with object:call(variable). Calling :call results in the indirect call of the given method. The return value corresponds to the indirectly called method. The :call parameter can have a maximum of 15 other parameters. Please note this when you design methods. The method :call can also call builtin methods such as :insert or :delete.

Example

```
dialog Example
record Rec
{
  string String;
  rule void Method {}
  rule boolean Show (object Obj input) {}
}

on dialog start
{
  Rec:call(:Method);      // calls method Method
  Rec:call(:Show, this);  // calls method Show with
                          // the dialog being the object
}
```

3.5 Existence of a Method

The Dialog Manager can query during runtime, i.e. dynamically, whether a certain object has a certain method or a certain attribute. This kind of querying is carried out with the method :has. This call is made via object:has(attribute or method).

```
Object:has(<Method> | <Attribute>);
```

Example

```
dialog Example
record Rec
{
  string String;
  rule void Method {}
}
on dialog start
{
  print Rec:has(:Method);  // results in true
  print Rec:has(.String);  // results in true
  print Rec:has(:Print);   // results in false
  print Rec:has(.Data);    // results in false
}
```

Index

T

U