Object Action Language Reference Manual

Introduction

Purpose

The purpose of this manual is to serve as a reference and general user's guide to aid in the correct specification of action semantics for UML models. Although originally designed for models used with the BridgePoint UML Suite, the language described herein can be used to define the action semantics for any UML model in any tool.

The Object Action Language ™ is written to satisfy the following goals:

- Readability - Modelers must be able to easily understand the OAL for development and reviews.
- Derivation - Event generation and data access information is captured for derivation of the Object Collaboration Diagrams and Package Dependency Diagrams for both asynchronous (event) and synchronous (data access) communication.
- Simulation - The UML models can be simulated through interpretation of the actions by using the BridgePoint Verifier tool.
- Translation - Richness of expression is provided while maintaining a specification that can be automatically translated onto a target architecture.

Basic Concepts

The Object Action Language ™ (OAL) is used to define the semantics for the processing that occurs in an action. An action can be associated with the following modeled elements:

- states
- bridge operations
- functions
- class and instance-based operations
- mathematically-dependent attributes
- interface reference operations and signals

The Object Action Language provides for five types of action processes:

- data access
- event generation
- test
- transformation
- bridge and function
- inter-component messaging

It supports these through:

- control logic
- access to the data described by the class diagram
- access to the data supplied by events initiating actions
- the ability to generate events
- access to timers and to the current time and date

In a UML model, unlike conventional programming, there is no concept of a "main" function or routine where execution starts. Rather, the models are executed in the context of a number of interacting finite state machines, all of which are considered to be executing concurrently. Any state machine, upon receipt of an event (from another state machine or from outside the system) may respond by changing state. On entry to the new state, a block of processing (an "action") is performed. This processing can in principle execute at the same time as processing associated with another state machine. (Whether this occurs in practice depends on the nature of the software and hardware architectures used to implement the system.)

The Object Action Language is used to define the processing executed during the action. The execution rules are as follows:

- Execution commences at the first statement in the action and proceeds sequentially through the succeeding lines as directed by any control logic structures.
- Execution of the action terminates when the last statement is completed.

These rules also apply to actions defined for bridge operations, functions, class and instance-based operations, and mathematically-dependent attributes.

Intended Audience

This manual is written for modelers and software engineers who use a process and set of modeling tools that support the creation, simulation, and translation of UML models. Specifically, guidance is provided for the correct specification of actions formulated in the syntax of the Object Action Language.

The following section provides guidance on using this manual to meet these needs.

Examples

There are examples presented throughout this manual. Much of the OAL in these examples was extracted from a complete model of a real-world device (an auto-sampler used for chemical testing).

In many cases the examples from the auto-sampler model are augmented with additional, contrived examples in an effort to provide further illustrations of the OAL construct in question. As with any contrived examples, these sequences of OAL are provided as examples of the syntax for the language and not as

examples of valid modeling techniques for real world applications.

Language Structure

Overall Structure

An action consists of a number of statements. Each statement can be either a simple statement (such as an access to the attributes of a class) or a control logic structure (such as an `if` construct).

OAL statements are terminated by a semi-colon except following the `if` , `for each` , and `while` control constructs, described later.

Comments

Comments may be inserted by the use of the `//` characters at any point in the line. When this pair of characters is detected, the remainder of the line (up to the new-line character) is considered to be a comment and is ignored.

Names and Keywords

OAL statements are composed of:

- keywords (See Keywords.)
- logical and arithmetic operations
- names of modeled elements (classes, attributes, class and external entity keyletters, relationship numbers and phrases, event labels and meanings, and supplemental data items)
- local variables

Keywords may be represented in all upper case, all lower case, or with the first character upper case and all other characters in lower case.

Names in OAL statements must conform to the following rules:

1. Names are case sensitive. This includes local variables, keyletters, class attributes, and event identifiers.
2. Names must not begin with a numeric character [0-9].
3. A relationship phrase ('is owned by') or event meaning ('turn off pump')
    - may contain any ASCII characters.
    - must be enclosed by tick marks. The tick marks may be omitted from an event meaning if it contains no spaces.
    - must be contained on a single line.

4. Class keyletters, event labels, and attribute names may contain only the characters [a-z][A-Z][0-9][_#]. Spaces are not permitted.

White Space

White space (spaces and tabs) may be inserted at any point in an OAL statement other than:

- within a name or keyword,
- between two consecutive colons, or
- between the characters of a comparison operator.

Keywords

Keywords

The following enumerates the list of reserved words.:

across	*and*	*any*	*assign*
assigner	*break*	*bridge*	*by*
cardinality	*class*	*continue*	*create*
creator	*delete*	*each*	*elif*
else	*empty*	*end*	*event*
false	*for*	*from*	*generate*
if	*in*	*instance*	*instances*
many	*not*	*not_empty*	*object*
of	*one*	*or*	*param*
rcvd_evt	*relate*	*related*	*return*
select	*selected*	*self*	*send*
sender	*to*	*transform*	*true*
unrelate	*using*	*where*	*while*

Any keyword can appear in full upper case, full lower case or with the first character in upper case and the remaining characters in lower case.

Data Items

Data Items Within an Action

The OAL expression of an action has access to and can produce certain data items. The following data items are available to be read at the start of and throughout an action:

- constants
- values of attributes of classes
- supplemental data items carried by the event that initiated the action
- local variables (created by statements within the action)

These data items can be produced during an action:

- local variables
- values of attributes of classes
- supplemental data items to be carried by an event generated during the action

Finally, the instance handle *self* may be used to refer to the currently executing instance in an instance statechart (but not in a class statechart), to an instance in an instance-based operation, and to the instance of a mathematically-dependent attribute.

Modeled Elements

All data items referenced or produced by an action must have a data type. The following data types are defined for class attributes, supplemental data items of an event, and local variables.

integer	string	unique ID	timer handle
real	date	instance handle	event instance
boolean	timestamp	instance handle set	state
component handle			

The analyst may also define domain-specific data types based upon these data types.

Data types within the OAL are "analysis" data types, and reflect only the set of legal values a variable can

take on.

Implicit Typing

There are no type declaration statements in the OAL. All data items are implicitly typed by the value assigned to them on their first use within an action.

Local Variables

Local variables can be of any of the data types listed in Assigning Data Types. Two of these types require special consideration:

instance handle: The identification of an instance of a class. The implementation of this type is entirely dependent on the architecture; hence, its form is unknown to the user of the OAL.

instance handle set: A set of instance handles.

Because instance handles and instance handle sets are obtained by selection from existing instances, the following situations can arise: (1) a local variable of type instance handle may not contain a valid reference to an instance, and (2) a local variable of type instance handle set may refer to an empty set. These situations can be detected by using the supplied unary set operators.

Notes

Attributes cannot be of type instance handle or instance handle set.

Strictly speaking, a local variable is not of type instance handle, but rather of type instance handle for a particular class. Any attempt to use an instance handle for one class in the context of another is an error.

In any action within an instance statechart, a special instance handle called `self` is always available. `Self` is always defined as a handle to the instance of the class that is executing the current action. The value of `self` cannot be changed by an action.

The instance handle `self` has no meaning inside of an action for a class-based operation, a bridge operation, or a function. `Self` is not defined for a class statechart.

In any action within an instance-based operation or mathematically-dependent attribute, `self` is always available. `Self` is always defined as an instance handle to the instance of a class against which the operation is being executed, or the instance for which the attribute is being read. The value of `self` cannot be changed by an action.

Assigning Data Types

The data type names used in this manual have been chosen for readability. The table lists the correspondences between the data type names used here and in BridgePoint Builder.

in this manual	in BridgePoint Builder
integer	integer
real	real
boolean	boolean
string	string
date	date
timestamp	timestamp
unique ID	unique_ID
state	state<State_Model>
timer handle	inst_ref<Timer>
instance handle	inst_ref<Object>
instance handle set	inst_ref_set<Object>
event instance	inst<event>
component handle	component_ref

Variable Initialization

Some situations arise where a variable may possibly be declared without being assigned a value. An obvious situation where this occurs is when an instance of a class with attributes is created. Though the attributes should not be accessed before they are assigned, there is nothing preventing the user from

attempting to do so.

For the purposes of the BridgePoint Verifier, unassigned variables are considered UNDEFINED. The user should not intentionally read the data from UNDEFINED variables.

For the purposes of the translated code, the value of unassigned variables lies entirely in the realm of the software architecture defined by the model compiler. Even so, use of an unassigned variable as a read value or in an expression should be avoided.

Examples

```
// Reading an uninitialized attribute
create object instance d of DOG; // Attributes of d are not
initialized
dog_name = d.name; // Error! Cannot read uninitialized value
           // Using an uninitialized instance reference
select many f_set from instances of fish; // Select an empty
instance set
for each f in f_set
  // statement block
end for;
// Instance reference f is available in this scope
// and may be uninitialized if f_set was empty.
fish_type = f.type; // Warning! f may be uninitialized.
```

Scoping

The scope of a variable is defined as the block of code in which the variable may be accessed. A block of code can be the entire OAL for the given action, or it may be a *<statements>* block within a control logic structure.

Each control logic structure contains at least one new scope. All variables that were accessible in the scope containing the structure are also accessible in the block or blocks contained by the structure, essentially causing the contained scopes to inherit variables from the parent scope. Any variables declared within a given control logic block fall out of scope when execution exits the block.

Control logic structures may contain multiple scopes, either by repeated nesting of new structures or by using the *elif* or *else* constructs in an *if* structure. When nesting of control logic is used, each new structure defines a new scope. In an if statement, each *elif* or *else* structure contained within the *if* block defines a new scope, and each new scope inherits the scope of the block containing the *if* statement.

Notes

A local variable is implicitly declared at the moment it is assigned, with a scope limited to the current block.

Local variables have a maximum scope of the entire OAL for the current action.

In the *for* statement, the local variable declared by *<instance handle>* has the same scope as the block containing the *for* statement.

The *where* clause has a special variable, *selected* , that has scope limited to the *<where expression>* .

The scope of *self* for an instance's action, operation, or attribute, is the entire OAL for the current action.

Example

```
// begin action
// scope1 - global scope for this action.  Variables declared
here
// are accessible anywhere in this action.
delta = self.destination - self.current_position;
if (delta == 0)
  // scope2 - Variables declared here are only accessible
  // within if statement.
  spin_spot = CARPIO::carousel_spin(car_id:self.carousel_ID);
end if; // All variables declared in scope2 are not accessible
    // after the end if.
select many rows from instances of ROW;
for each row in rows
  // scope3 - Variables declared here are only accessible within
  // for each statement.
  st = row.sampling_time;
end for; // All variables declared in scope3 are not accessible
after
     // the end for.
// row is available in the scope containing the for each
statement
// (scope1).
if (delta <= 2)
  // scope4 - Variables declared here are accessible within this
  // if block and in scope4.1 and scope4.2.
  if (CARPIO::angle(car_id:self.carousel_ID) == 30)
    // scope4.1 - Variables declared here are only accessible
    // within this if block.
  end if;
```

```
  if (CARPIO::angle(car_id:self.carousel_ID) == 60)
    // scope4.2 - Variables declared here are only accessible
within
    // this if block.
  end if;
end if;
// end action
```

Data Type Strength

A strict data typing scheme where all types, including numerics, are incompatible is enabled by default. Action language preferences are available for the user to weaken the rules affecting reals and integers. These preferences can be found at *Window > Preferences… > BridgePoint > Action Language*. The settings are stored on a per-workspace basis.

Two settings are available:

- *Allow promotion of integer to real* : The analyst may choose to allow OAL statements that implicitly promote integer values to real values.

- *Allow lossy assignment of real to integer* : If the former value is set to *“Yes”*, then the analyst may also choose to allow OAL statements that implicitly demote real values to integer values with possible loss of data.

Control Structures

If Construct

Syntax

```
// Note that there is no semi-colon following the
// "if <boolean expression>"
if <boolean expression>
  <statements>  // Executed if <boolean expression> is TRUE
end if;
if (<boolean expression>)
  <statements>  // Executed if above boolean expression
evaluates to TRUE
elif (<boolean expression>)
  <statements>  // Executed if above boolean expression
evaluates to TRUE
          // and previous boolean expression is FALSE
else
  <statements>  // Executed if both boolean expressions
evaluate to FALSE
end if;
```

<boolean expression> is an expression evaluating to *TRUE* or *FALSE* .

Notes

The *if* construct may contain as many *elif* clauses as desired.

Only one *else* clause may be used, and it must appear at the end of the *if* construct.

Example

The following example shows an *if/elif/else* construct:

```
// Assign x with a different number for each name.
if (name == "John")
  x = 1;
elif (name == "Bill")
```

```
  x = 2;
elif (name == "Michael")
  x = 3;
else
  // If not a known name, assign x to 4.
  x = 4;
end if;
```

This example shows nested *if* constructs:

```
// Carousel:  Going
self.destination = rcvd_evt.destination;
delta = self.destination - self.current_position;
if ( delta == 0 )
  generate C2:there to self;
else
  select any probe from instances of SP
    where (selected.current_position == "down");
  if (not_empty probe)
    generate C2:there to self;
  else
    spin_spot = CARPIO::carousel_spin(
      car_id:self.carousel_ID, destination:delta );
  end if;
end if;
```

For Each Loop

The *for each* loop allows for the iteration over a set of instance handles in an instance handle set.

Syntax

```
for each <instance handle> in <instance handle set> // Note no
semi-colon
  <statements>
end for;
```

<instance handle> is a local variable referring to a single instance.
<instance handle set> is a local variable referring to a set of instance handles.

Notes

The statements in the *for each* construct are executed once against each instance in *<instance handle set>* .

The order in which the particular instances are processed is undefined.

Because the statements in the *for each* construct can, in principle, be executed in parallel (as when instances are dispersed over multiple processors), the concept of a loop counter is undefined. Consequently, the analyst should not attempt to defeat this restriction.

Example

```
// C is the keyletter for the Child object.
// children is an implicitly typed variable of <instance handle
set> of C.
select many children from instances of C;
for each child in children
  generate C1:'time for bed' () to child;
end for;
```

While Loop

The *while* construct is used to sequentially execute the code it contains for as long as the condition is evaluated as *TRUE* .

Syntax

```
while (<boolean expression>) // Note no semi-colon
  <statements>
end while;
```

<boolean expression> is an expression evaluating to *TRUE* or *FALSE*
<statements> are zero or more OAL statements.

Note

When the *while* loop is executed, the statements in the *while* construct are executed consecutively as long as the *<boolean expression>* evaluates to *TRUE* .

Example

```
// Create 20 doors with IDs 1-20
i = 1;
while (i <= 20)
  create object instance d of DOOR;
  d.ID = i;
  i = i + 1;
end while;
```

Break

The *break* statement allows the early termination of both *for each* and *while* loops. This can have some significant performance implications in the case of large loops that need not step through the entire iteration.

Note

The *break* statement only applies to the current *for each* or *while* loop containing it. To break out of nested loops, the *break* must be repeated for each loop construct the user wishes to exit.

Example

```
// Create and relate a B to every A while CTL says to keep
// creating.
while (CTL::create())
  breakout = FALSE;
  for each a in aset
    // If this a has name equal to "Jeff", break out of
    // for each loop.
    if (a.name == "Jeff")
      breakout = TRUE;
      break;
    end if;
    // Create and relate a new b to the given a.
    create object instance b of B;
    relate b to a across R1;
  end for;
  // If "Jeff" was found, break out of while loop also.
  if (breakout)
    break;
  end if;
end while;
```

Continue

The *continue* statement causes the next iteration of the enclosing *for each* or *while* loop to begin, avoiding execution of the loop's remaining code.

Note

The *continue* statement only applies to the current loop containing it.

Example

```
// Create and relate a B to each A, except for As with ID of 13.
for each a in aset
  // If a.ID is 13, don't create and relate a B to it and
  // continue to the next.
  if (a.ID == 13)
    continue;
  end if;
  create object instance b of B;
  relate b to a across R1;
end for;
```

Nested Control Logic

Control logic may be nested to any depth.

Example

```
// Send a 'time for bed' event to all children 5 and under.
select many children from instances of C;
for each child in children
  if (child.age <= 5)
    while (child.awake)
      generate C1:'time for bed' () to child;
      if (not lights.out)
        generate C2:'turn off lights' () to child;
      end if;
    end while;
  end if;
end for;
```

Class Manipulations

Creating Instances

Creation of an instance of a class is achieved by use of the *create* statement.

Syntax

```
create object instance <instance handle> of <keyletter>;
create object instance of <keyletter>;
```

<keyletter> is the keyletter of a class in the model.

Notes

The *<instance handle>* returned is the handle of the newly created instance of class *<keyletter>* . The handle can be used only to refer to an instance of class *<keyletter>* .

The *<instance handle>* in the *create* statement cannot be *self* .

The value of all identifying attributes must be set by the analyst before completion of the action in which an instance is created. Note that identifying attributes of type unique ID cannot be assigned values, as they are initialized by the system.

All unconditional relationships that involve the newly created instance should be satisfied before completing the action in which the instance is created. This can be done either by directly relating the associated instance or by generating an event that will cause the newly created instance to be properly related.

Examples

```
// Create instances of autosampler classes
create object instance car of C;
create object instance row of ROW;

create object instance of SP; // no instance handle needed
```

Selecting Instances

The *select* statement can be used to assign an instance or set of instances to either an instance handle or a instance handle set respectively. An optional *where* clause can be used at the end of the *select* statement to limit the selection. Within the *where* clause, the *selected* instance handle refers to each of the instances in the entire set defined by *<keyletter>* . The instance handle *selected* is meant to be used as an instance handle in a boolean comparison to form the *where* expression. The instance or set of instances returned match the criteria of the *where* expression, and may be empty.

Syntax

```
select any <instance handle> from instances of <keyletter>;
select many <instance handle set> from instances of <keyletter>;
select any <instance handle> from instances of <keyletter>
where <where expression>;
select many <instance handle set> from instances of <keyletter>
where <where expression>;
```

<instance handle> is the handle for an instance of the class specified by *<keyletter>* .

<instance handle set> is a set of handles for all selected instances of the class specified by *<keyletter>* .

<where expression> is a type of boolean expression using *selected* keyword.

Notes

If the optional *where* clause is used, the returned instance or set of instances meet the criteria of the *where* expression. This implies that the instance handle may be empty, or the instance handle set may be empty if no instance fulfills the criteria.

If the *select any* form is used, an arbitrary instance will be obtained from the selected set.

If the *select many* form is used then the entire set of instances will be obtained.

If the *select any ... where* form is used, an arbitrary instance that fulfills the *<where expression>* will be obtained.

If the *select many ... where* form is used then the set of instances that fulfill the *<where expression>* will be obtained.

The instance handle *selected* is valid only within < *where expression*> .

Example

```
// Select an arbitrary instance.
select any dp_one from instances of DP;

// Select all instances.
select many dp_set from instances of DP;

// Select an instance of DP whose available attribute is TRUE.
select any dp_avail from instances of DP where selected.
available == TRUE;

// Select a set of instances from DP whose available attribute
is FALSE.
select many dp_unavail_set from instances of DP
  where selected.available == FALSE;
```

Making selections across relationships is describe in Relationship Navigation.

Writing Attributes

Attributes of instances may be set to specified values by use of the *assign* statement.

Syntax

```
[assign] <instance handle>.<attribute> = <expression>;
```

<instance handle> is a handle to an instance of a class.

<attribute> is the name of an attribute of the class.

<expression> is either a boolean, string, or arithmetic expression.

The assign statement takes the value in *<expression>* and assigns it to the attribute *<attribute>* for the instance specified by *<instance handle>* .

The *assign* keyword is optional.

Notes

The *<expression>* must evaluate to the data type of *<attribute>* , unless *<attribute>* is either a real or an integer, in which case *<expression>* can be either a real or integer value.

A value cannot be assigned to a referential attribute. Relationships must be maintained via the *relate* and *unrelate* constructs.

A value cannot be assigned to an attribute of type unique ID, as such an attribute is initialized by the system when the instance is created.

Mathematically-dependent attributes cannot be written to, only read. The value of a mathematically-dependent attribute must be set within the action of the attribute. See Writing Mathematically-Dependent Attributes.

Example I

```
// Account is a class with attributes branch, account_number,
// and balance.  Assume new_account, this_branch, and
initial_deposit
// are event supplemental data items.

// First, create a new instance.
create object instance my_account of ACCT;

// Now set attribute values using the returned instance handle.
my_account.branch = rcvd_evt.this_branch;
my_account.account_number = rcvd_evt.new_account;
my_account.balance = rcvd_evt.initial_deposit;
```

Example II

```
// Create and initialize a row in the autosampler carousel.
create object instance row of ROW;
relate row to car across R1;
row.radius = 10;
row.current_sampling_position = 0;
row.maximum_sampling_positions = 5;
```

```
row.sampling_time = 5000;
row.needs_probe = false;
```

Writing Mathematically-Dependent Attributes

Mathematically-dependent attributes are attributes that have their value derived from other modeled elements. It is not possible to directly write to a mathematically-dependent attribute as described above. Instead, OAL must be used to specify the derived value. When the attribute is read in an action, the value of the attribute is calculated from the OAL specified in the action for the mathematically-dependent attribute.

Syntax

From within the action of a mathematically-dependent attribute:

```
[assign] self.<attribute> = <expression>;
```

<attribute> is the name of the mathematically-dependent attribute.

<expression> is either a boolean, string, or arithmetic expression.

The assign statement takes the value in *<expression>* and assigns it to the attribute *<attribute>* for the instance specified by *self* .

The *assign* keyword is optional.

Notes

The *<expression>* must evaluate to the data type of *<attribute>* , unless *<attribute>* is either a real or an integer, in which case *<expression>* can be either a real or integer value.

A value cannot be assigned to a referential attribute. Relationships must be maintained via the *relate* and *unrelate* constructs.

A value cannot be assigned to an attribute of type unique ID, as such an attribute is initialized by the system when the instance is created.

The action parsing routine for attributes checks to see if the variable *self* is written somewhere in the state action, and if not, a parse error is reported.

Care should be taken to make sure that all paths inside an action actually set the attribute.

The *param* and *return* keywords are not supported in the action of the attribute.

Example

```
// Mathematically-dependent attribute (MDA) action
self.volume = self.length*self.width*self.height;

// Action reading the MDA volume
v = cube.volume;
```

Reading Attributes

A class attribute may be referenced in an expression using the form:

Syntax

```
<instance handle>.<attribute>
```

<instance handle> is a handle to an instance of a class.

<attribute> is the name of an attribute of the class.

Notes

You may read the value of any attribute, including referential attributes.

<instance handle>.<attribute> is an expression and can be used in any OAL construct specifying an expression.

When you wish to obtain an associated instance, use the relationship navigation constructs rather than reading (a succession of) referential attributes. This ensures that code generators can detect the purpose of the read and therefore produce accurate and effective translation.

Example

```
// Create new instance and get handle.
Create object instance myrobot of R;
```

```
// Use the instance handle to read attribute values.
myx = myrobot.x_position;
myy = myrobot.y_position;

// Position the row for sampling.
select one car related by self->C[R1];
self.next_sampling_position = self.current_sampling_position +
1;
next =
  ROW::convert_dest( radius:self.radius,
  next_sampling_position:self.next_sampling_position );
generate C1:go(destination:next) to car;
```

Deleting Instances

Syntax

```
delete object instance <instance handle>;
```

Notes

This statement deletes the instance specified by *<instance handle>* .

When an instance of a class is deleted, it is no longer available to the domain where the class is defined. However, sophisticated software architectures can be imagined which support the notion that the instance is kept for logging purposes although the defining domain cannot see it.

Depending on the architecture, deleting an instance may or may not be sufficient to specify deletion of any attached relationships. Because certain architectures may fail if such dangling relationships are used at run time, we recommend that the analyst explicitly delete relationships before deleting the participating class instances.

Example

```
// Delete every instance of DG with name equal to Fido.
select many dogs from instances of DG where (selected.name ==
"Fido");
for each dog in dogs
  select one owner related by dog->OWN[R23];
  unrelate dog from owner;
  delete object instance dog;
end for;
```

Relationships

Relationship Specifications

A relationship specification identifies exactly which relationship is required to be created, navigated, or deleted.

Syntax

```
R<number>
r<number>
R<number>.<relationship phrase>
r<number>.<relationship phrase>
```

<number> the number of the relationship as shown on the class diagram.(e.g., R1).

<relationship phrase> is the text that appears at the destination end of the relationship, enclosed in tick marks and contained on a single line.

Note

Either *R* or *r* may be used when referring to a relationship.

Examples

```
r5
R10.'owns'
r10.'is owned by'
R22.'uses'
R1.'Is rotated by'
R1.'Contains'
R2.'Is assigned to'
```

Creating an Instance of a Relationship

Syntax

```
relate <source instance handle> to <destination instance
handle> across <relationship specification>;
```

```
relate <source instance handle> to <destination instance
handle> across <relationship specification> using <associative
instance handle>;
```

<source instance handle> is the handle of the first class instance to be related.

<destination instance handle> is the handle of the second class instance to be related.

<relationship specification> is the specification of the relationship from the source class to the destination class. This can be any of the forms described in the previous section.

<associative instance handle> is the handle of an existing class instance that is used as the associative class instance for this relationship instance.

Notes

The relationship specification should be framed as if navigating from source class to destination class.

The source, destination, and associative instance handles may be *self* .

If an attempt is made to relate two instances via the same relationship more than once, this is regarded as a run time error by the BridgePoint Verifier unless the relationship is (M:M)-M.

The *using* *<associative instance>* form is used when an associative relationship is being instantiated. The associative instance must have already been created before the relationship is instantiated.

Examples

```
select any dp_inst from instances of DP;
select any d_inst from instances of D;
relate dp_inst to d_inst across R1;
```

```
select any a_inst from instances of A;
select any b_inst from instances of B;
create object instance c_inst of C;
relate a_inst to b_inst across R1 using c_inst;
```

```
// State 3. "Assigning Probe to Row"
select any row from instances of ROW
```

```
        where ( selected.needs_probe == true );
    select any probe from instances of SP
      where ( selected.available == true );
    probe.available = false;
    row.needs_probe = false;
    create object instance assignment of PA;
    relate row to probe across R2 using assignment;
    generate PA_A3:probe_assigned() to PA class;
    generate ROW2:probe_assigned() to row;
```

Deleting an Instance of a Relationship

Syntax

```
unrelate <source instance handle> from <destination instance
handle> across <relationship specification>;

unrelate <source instance handle> from <destination instance
handle> across <relationship specification> using <associative
instance handle>;
```

<source instance handle> is the handle of the first class instance to be unrelated.

<destination instance handle> is the handle of the second class instance to be unrelated.

<relationship specification> is the specification of the relationship from the source to the destination class.

<associative instance handle> is the handle of the associative class instance that captures the relationship instance.

Notes

The relationship specification should be framed as if navigating from the source class to the destination class.

An attempt to unrelate two instances that are not related by the specified relationship is regarded as a run time error by the BridgePoint Verifier.

The source, destination, and associative instance handles may be *self* .

If an associative relationship is unrelated then the associative class instance(s) will not be deleted. The

analyst must specify this explicitly.

If an unconditional relationship is deleted, instances of participating classes will not automatically be deleted to remain consistent with the Class Diagram. It is the responsibility of the analyst to ensure that the Class Diagram is respected. This applies equally to super/subtype relationships.

Examples

```
unrelate a_inst from b_inst across R1;
unrelate a_inst from b_inst across R1 using c_inst;
delete object instance c_inst;
```

Relationship Navigation

Relationship navigation is the function whereby relationships specified on the Class Diagram are read in order to determine the instance or set of instances that are related to an instance of interest.

Syntax

```
select one <instance handle> related by <start> ->
<relationship link> -> ... <relationship link>;

select any <instance handle> related by <start> ->
<relationship link> -> ... <relationship link>;

select many <instance handle set> related by <start> ->
<relationship link> -> ... <relationship link>;

select one <instance handle> related by <start> ->
<relationship link> -> ... <relationship link> where <where
expression>;

select any <instance handle> related by <start> ->
<relationship link> -> ... <relationship link> where <where
expression>;

select many <instance handle set> related by <start> ->
<relationship link> -> ... <relationship link> where <where
expression>;
```

<start> is an *<instance handle set>* or *<instance handle>* obtained from a previous *select* statement.

<relationship link> is a *<keyletter>[<relationship specification>]* , where the square brackets are literal and do not indicate optional text.

```
<keyletter> is the keyletter of the class reached by the
specified relationship.

<relationship specification> is the specification of the
relationship from the source to the destination class.
```

<where expression> is a type of boolean expression using the *selected* keyword.

Notes

A *relationship link chain* is the sequence of *<relationship link>* 's used to specify the path from the starting instance or set of instances to the destination.

Use the *select one* form if at most one instance handle can be returned by navigating the relationship link chain.

Use the *select any* or *select many* form if more than one instance handle can be returned by navigating the relationship link chain. *Select any* returns a single instance, and *select many* returns all instances that meet the selection criteria.

The *select any* form returns the instance handle of an arbitrary instance of the class at the end of the relationship link chain.

The *select many* form returns an instance handle set containing all the instances of the class at the end of the relationship link chain.

The *select any ... where* form returns the instance handle of an arbitrary instance of the class at the end of the relationship link chain that fulfills the *<where expression>* criteria.

The *select many ... where* form returns an instance handle set containing all the instances of the class at the end of the instance chain that fulfill the *<where expression>* criteria.

The relationship phrases in the relationship link chain must be given in the direction of navigation.

If the starting *<instance handle>* or *<instance handle set>* is empty, then the result will be considered a run time error.

The returned *<instance handle>* or *<instance handle set>* can be empty if any of the relationships in the chain are conditional in the direction of navigation.

If the optional *where* clause is added, the returned instance or set of instances will meet the criteria of *<where expression>* . This implies that the instance handle or the instance handle set may be empty if no instance(s) matched.

Example I

```
select one cat related by owner->C[R1];
select any dog related by owner->D[R2];
select many dogs related by owner->D[R2];


select any assignment from instances of PA here ( selected.
probe_ID ==   self.probe_ID );
select any dog related by owner->D[R2] where ( selected.name ==
"Fido" );
select many dogs related by owner->D[R2] where selected.color
== "black";
```

Example II

```
select any student from instances of STU;
select many major_courses_offered related by   student->PROF
[R34]->DEPT[R23]->COUR[R40];
```

Events

Receiving Event Data

The keyword *rcvd_evt* is the name of a structure containing all of the supplemental data items received with an event.

Syntax

```
rcvd_evt.<supplemental data item>
```

<supplemental data item> is the name of the data item.

Note

rcvd_evt.<supplemental data item> is an *<expression>* and so can be used in any OAL construct specifying an expression.

Example

```
select any robot from instances of R;
robot.from = rcvd_evt.source;
robot.to = rcvd_evt.destination;
//
self.destination = rcvd_evt.destination;
```

Event Generation

Syntax

```
generate <event label> to <target>;
generate <event label>:<event meaning> to <target>;
generate <event label> (<event parameters>) to <target>;
generate <event label>:<event meaning> (<event parameters>) to
<target>;
generate <instance handle>.<attribute>;
```

<event label> is *<keyletter><event number>* .

<event meaning> is the meaning of the event, enclosed by tick marks as in *'turn off the light'* ; the tick marks may be omitted if the event meaning contains no spaces.

<event parameters> provides the supplemental data items (if any) to be carried by the event. Each data item is given in the form *<supplemental data item>:<expression>* . When multiple supplemental data items are required, separate the *<supplemental data item>:<expression>* pairs by commas. If there are no supplemental data items, the parentheses may be omitted.

> *<supplemental data item>* is the name of a data item to be sent with the event.
>
> *<expression>* is a string, arithmetic, boolean, simple, or compound expression. The data type of the expression must match the data type defined for the given data item.

<target> is specified in a variety of ways, depending on the destination of the event.

- For an event directed to an existing instance of a class, *<target>* is an instance handle .
- For a creation event, *<target>* is: *<keyletter> creator* .
- For an event directed at a single-instance assigner, *<target>* is: *<keyletter> class* .
- For an event directed at an external entity, *<target>* is the external entity's keyletters.

<instance_handle> is a handle to an instance of a class.

<attribute> is the name of an attribute of the class.

Notes

The *to* clause provides all the information necessary to identify the destination of the event.

If an event has no supplemental data items, the empty parentheses around *<event parameters>* may be omitted.

Supplemental data items may appear in any order in *<event parameters>* .

All supplemental data items defined for the event must be supplied.

The *<event meaning>* field is optional. It must be enclosed in tick marks if it contains spaces, and it must be contained on a single line.

If *<event meaning>* is not used, the colon after *<event label>* must be omitted.

Examples

```
// event to existing instance
// State 1. "Up"
self.current_position = "up";
select one row related by self->ROW[R2];
generate ROW4:sample_complete() to row;

// creation event
generate S1:'Create sale' (dept:dept_no, amount:sale_value) to
S creator;

// event to assigner
generate PA_A1:row_needs_probe() to PA class;

// event to external entity
generate PIO7:'Motor start' (motor_no:motor_id) to PIO;
```

Event Pre-creation

An event may be created without sending it by using the *create event* statement. This statement should be used only

1. to create an event for an analysis timer.
2. as directed by the rules of the particular architecture onto which you plan to translate your models.

Syntax

```
create event instance <event instance> of <event label> to
<target>;

create event instance <event instance> of <event label>:<event
meaning> to <target>;

create event instance <event instance> of <event label> (<event
parameters>) to <target>;

create event instance <event instance> of <event label>:<event
meaning> (<event parameters>) to <target>;
```

<event label> , *<event meaning>* , *<event parameters>* and *<target>* are as defined in Event Generation.

<event instance> is a local variable of type event instance.

Notes

The local variable *<event instance>* can be used to refer to an event instance of any type.

Event instances can be assigned to attributes of classes that are of type event instance.

Please refer to the notes in *Event Generation* .

Sending a Pre-created Event

Event instances may be sent using the *generate* statement.

Syntax

```
generate <event instance>;

generate <instance handle>.<attribute>;
```

<event instance> is a local variable of type event instance.

<instance handle> is a handle to an instance of a class.

<attribute> is an attribute of the class.

Notes

Sends a pre-created event to its intended recipient.

This feature is provided for use with analysis timers. These timers will eventually be replaced with delayed events, at which time, support for pre-created events will likely be removed.

Expressions

Simple Expressions

Simple expressions are single unary or binary operations. An expression is not a complete OAL statement, but is evaluated as part of a full OAL statement such as *assign* , *if* , *where* , etc. Logical binary operators *and* and *or* are supported for both compound and simple expressions.

Syntax

```
<read value>
<unary operator> <read value>
<read value> <binary operator> <read value>
```

<*read value*> is a constant, a local variable, the attribute of a class, a supplemental data item received from an event, an operation invocation, a bridge invocation, or a function invocation. It can also be a parameter specified with the *param* keyword in the action of a bridge operation, function, or operation.

<*unary operator*> is any unary operator appropriate for the data type to which the expression evaluates. For boolean read values, the unary operator is *not* . For arithmetic read values, the unary operators are + and - . For instance handle and instance handle set read values, the unary operators are *empty* , *not_empty* , and *cardinality* .

<*binary operator*> is any binary operator appropriate for the data types to which the expressions evaluate. For boolean read values, the binary operators are *and* and *or* . For arithmetic read values, the binary operators are + , - , * , / , and % . For instance handle and instance handle set read values, the binary operators are == and != . For string read values, the binary operator is + .

Example

```
not (CHK::get_status())
x + y
name == "Jeff"
"Bridge" + "Point"
cust1.age - cust2.age
```

Compound Expressions

Compound expressions can be used to combine simple expressions, allowing for multiple tests and more complex assignment arithmetic. Logical binary operators *and* and *or* are supported for both compound and simple expressions.

Syntax

```
<operator> <expression>
<read value> <operator> <expression>
<expression> <operator> <read value>
<expression> <operator> <expression>
```

<expression> is a simple or a compound expression.

<operator> is any operator appropriate for the data types to which the expressions evaluate.

<read value> is a constant, a local variable, the attribute of a class, a supplemental data item received from an event, an operation invocation, a bridge invocation, or a function invocation.

Notes

The analyst can depend on the following rules regarding the order of evaluation of expressions:

- Parentheses can be used to override all other ordering rules.
- Standard mathematical precedence governs the order of evaluation for all mathematical operations.
- All subexpressions with operators of equal precedence are evaluated from left to right, starting with the operators of highest precedence. This is repeated until the compound expression has been completely evaluated.
- A short-circuit with regard to compound expressions means that an expression can be fully evaluated based upon the value of one of its subexpressions. Whether or not short-circuiting occurs depends entirely on the implementation of the software architecture or the simulator being used. The analyst should therefore avoid writing OAL that depends on short-circuiting of expressions.

Examples

```
// examples of compound expressions:
not (arm.available and servo.on)
2 * (x + y) + TIM::timer_remaining_time(timer_inst_ref:timer_1)
(a + b) / (c - d)


// examples of OAL statements using
// compound expressions:
if ((i == 1) AND (name == "Doug"))
  x = 0.5 * (y + z);
end if;
```

```
x = x * ((x + 1) / (x + 2));
```

Arithmetic Expressions

Arithmetic expressions are defined for real and integer data types only. These data types may be mixed for any given expression. Multiplicative operators are *, /, and %. Additive operators are + and -. Multiplicative operators take precedence over additive operators. Parentheses may be used to force precedence in arithmetic expressions.

Syntax

```
<unary arithmetic operator> <expression>
<expression> <binary arithmetic operator> <expression>
```

<expression> is any of the following that evaluates to a real or integer value: numeric constant, local variable, attribute of a class, simple expression, compound expression, operation invocation, bridge invocation, function invocation, or supplemental data item received from an event.

<unary arithmetic operator> is + or -.

<binary arithmetic operator> is +, -, *, /, or % (remainder from arithmetic division).

Note

If any data item in the expression is real, the expression will evaluate to a data type of real.

Examples

```
-27
2 + 2
(x + y) / 2
0.707 * voltage
(plane.offset + ALT::get_altitude())
```

Boolean Expressions

A boolean expression is any expression that evaluates to either a *TRUE* or *FALSE* value. Boolean expressions are often used for comparison in statements like *if* and *while*, and also in *where* clauses. Although boolean expressions usually contain other expression types (such as arithmetic or string expressions), they can also be used to compare time values, handles, and unique IDs. There is also

one unary operator, *not* , which can be used to logically negate a boolean expression.

Syntax

```
not <boolean expression>
<expression> <boolean operator> <expression>
<time value> <boolean operator> <time value>
<handle> <boolean operator> <handle>
<unique_id> <boolean operator> <unique_id>
```

<expression> is any expression, simple or compound. Both expressions must evaluate to the same type, either boolean, arithmetic, or string.

<boolean expression> is a simple or compound expression that evaluates to a boolean value.

<boolean operator> is a logical operator.

<time value> a date or a timestamp variable (or an operation, bridge, or function invocation that returns a date or a timestamp).

<handle> an instance handle, instance handle set, or a timer handle (or an operation, bridge, or function invocation that returns a handle to a timer).

<unique id> a variable of type unique ID (or an operation, bridge, or function invocation that returns a unique ID).

Note

The left and right values of a binary boolean expression must evaluate to the same data type, with the exception of integer and real.

Examples

```
x == 1
id != "abc"
CTL::error() or flag
(account.balance == 0.00) and ((TIM::get_current_time() -
last_pay_time)
  >= max_wait)
```

Logical Operator	*Meaning*	*Valid Data Types*
==	equals	integer, real, boolean, date, timestamp, string, instance handle, unique ID, handle set, timer handle
!=	does not equal	integer, real, boolean, date, timestamp, string, instance handle, unique ID, handle set, timer handle
<	less than	integer, real, date, timestamp, string
>	greater than	integer, real, date, timestamp, string
<=	less than or equal to	integer, real, date, timestamp, string
>=	greater than or equal to	integer, real, date, timestamp, string
and	logical and	boolean
or	inclusive logical or	boolean
not	logical negation	boolean

String Expressions

A string expression is any expression that evaluates to a string value. String expressions can be either a simple string or a concatenation of one or more simple strings.

Syntax

```
<simple string>
<simple string> + ... + <simple string>;
```

<simple string> is any of the following that evaluates to a string value: string constant, local variable, attribute of a class, operation invocation, bridge invocation, function invocation, or a supplemental data item received from an event.

Examples

```
"Hello, world!"
"Executable" + "-" + "UML"
cust.first_name + " " + cust.last_name
CHS::get_date_string(date:TIM::current_date())
```

Where Expressions

A *where* expression is a special type of boolean expression used in a *select* statement. The instance handle *selected* is valid only within the *where* expression. The *selected* keyword should be used as an instance reference to access the instances of the given set for the *select* statement containing the *where* expression. The *where* expression must evaluate to a boolean value, and must use the *selected* keyword.

Note

The *where* expression can only be used in the *where* clause of a *select* statement.

Examples

```
select any firstname in EMP where selected.name == "Bob";
select many accounts in ACC where (selected.status == "Ok") and
  (selected.balance > (min_bal + 200));


// Use where clause to find a particular probe.
select any probe from instances of SP
       where selected.probe_ID == param.probe_id;
generate SP3:probe_in_position to probe;
```

Assignment of Variables

Calculations are performed using the following form of the *assign* statement:

Syntax

```
[assign] <boolean var> = <boolean expression>;
[assign] <arithmetic var> = <arithmetic expression>;
[assign] <string var> = <string expression>;
```

<boolean expression> is an expression evaluating to *TRUE* or *FALSE* .

<arithmetic expression> is an expression evaluating to a real or an integer value.

<string expression> is an expression evaluating to a string value.

<boolean var> is a boolean variable or a boolean attribute of a class instance.

<arithmetic var> is a real or integer variable or a real or integer attribute of a class instance.

<string var> is a string variable or a string attribute of a class instance.

Notes

Arithmetic expressions are defined for real and integer data types only.

If *<arithmetic var>* is a local variable that is being assigned for the first time, it will be of type integer if *<arithmetic expression>* evaluates to type integer, and of type real if *<arithmetic expression>* evaluates to type real.

Once an *<arithmetic var>* has been assigned, it will not change data type from real to integer or from integer to real. Real and integer variables can, however be assigned integer or real values respectively.

If a real value is assigned to an integer variable, the fractional component is truncated.

The actual precision and truncation rules for arithmetic calculation depend on the software architecture and implementation domains in use.

The *assign* keyword is optional.

Examples

```
assign x = 1;
pass = TRUE;
assign name = "Rover";
f = RTR::get_frequency() + 100;
```

Constants

In many of the examples, constants have been used as parts of expressions. While this serves well for the

purposes of illustration, it should be noted that most analysis models require minimal use of constants since such data is more commonly stored as attributes of specification classes.

Syntax

The syntax depends on the base data type:

Integer:	1, 42, -127, etc.
Real:	1.0, 4.5, -56.0, etc.
String:	"string"
Boolean:	TRUE, FALSE

Notes

Constants may be defined for the above data types only.

A constant may be used in any construct requiring an expression.

Additional Unary Operators

Three set operators have been provided to allow the analyst to determine the size of an instance handle set or whether or not an instance handle is defined. These operations may be performed anywhere an expression may be used.

Syntax

```
empty <handle>
not_empty <handle>
cardinality <handle>
```

<handle> is an *<instance handle>* or *<instance handle set>* .

Notes

empty and *not_empty* return a value of type boolean.

cardinality returns an integer value.

Example

```
select one d_inst related by self->D[R1];
if (not_empty d_inst)
  // Statements here protected against access to empty d_inst.
end if;
```

Date and Time

External and Internal Time

This section describes statements that support date and time. The OAL supports two different concepts of time:

> *External time* : Time as known in the external world. For example, 12 October 1492, 13:25:10. The accuracy of external time is dependent on the architecture and implementation.
>
> *Internal time:* An internal system clock that measures time in "ticks". The value of a tick is dependent upon the architecture and implementation.

External Time

Syntax

To create a *<date variable>* , write

```
[bridge] <date variable> = TIM::create_date (day:<arithmetic
expression>,   month:<arithmetic expression>, year:<arithmetic
expression>,   second:<arithmetic expression>, minute:
<arithmetic expression>,   hour:<arithmetic expression>);
```

To read the current date or time, write

```
[bridge] <date variable> = TIM::current_date ();
```

To extract components of a *<date variable>*

```
<integer variable> = TIM::get_day (date:<date variable>);
<integer variable> = TIM::get_month (date:<date variable>);
<integer variable> = TIM::get_year (date:<date variable>);
<integer variable> = TIM::get_second (date:<date variable>);
<integer variable> = TIM::get_minute (date:<date variable>);
```

```
<integer variable> = TIM::get_hour (date:<date variable>);
```

<arithmetic expression> is an arithmetic expression evaluating to an integer value.

<date variable> is a local variable or a class attribute of type *date* .

<integer variable> is a local variable or a class attribute of type *integer* .

Note

External time is represented by a 24-hour clock.

Internal Time

Syntax

To read the internal system clock, write

```
[bridge] <time variable> = TIM::current_clock ();
```

<time variable> is a local variable or a class attribute of type *timestamp* .

Note

The system clock counts time in ticks. The size of a tick is dependent on the architecture and implementation.

Operations, Bridges, and Functions

Operation Invocation

The analyst may define class-based and instance-based operations as desired using the Operation Data Editor for a class under the Class Diagram. An operation invocation can be used as a stand-alone statement or it can be used in an expression.

Syntax

As an operation expression:

```
<keyletter>::<class-based operation name> (<data item>:
<expression>, ...)

<instance handle>.<instance-based operation name> (<data item>:
<expression>, ...)
```

As a stand-alone operation assignment statement:

```
<variable> = <keyletter>::<class-based operation name> (<data
item>:<expression>, ...);

<variable> = <instance handle>.<instance-based operation name>
(<data item>:<expression>, ...);
```

<keyletter> is the keyletter of a class.

<instance handle> is a handle to an instance of a class.

<class-based operation> and *<instance-based operation>* are the name of the operation.

<data item> is the name of a data item defined as input for *<class-based operation name>* or *<instance-based operation name>* .

<expression> is a string, arithmetic, boolean, simple, or compound expression. The data type of the expression must match the data type defined for the given data item.

<variable> is a class attribute or a local variable.

Notes

An operation expression may be used anywhere an expression is valid (e.g., assignment statement read values, control logic expressions, etc.).

The stand-alone operation must always be a stand-alone statement and cannot be used as an expression within another statement.

Since an operation expression may be used anywhere an expression may be used, stand-alone operation assignment statements are not necessary. The user may simply use an operation expression as a read value and use an *assign* statement to assign the return value to *<variable>* .

Parentheses are required even if there are no data items.

If *<variable>* is a local variable that is being assigned for the first time, it will be of the same data type as the return value of *<operation name>* .

If the type of *<variable>* has already been established (that is, if it is the attribute of a class or a local variable that has been previously assigned), then either:

- *<variable>* must be of the same data type as the output value of the operation,
  or
- the output value of the operation is of type *integer* or *real* and *<variable>* is of type *integer* or *real* .

Example

```
// operation expressions without assigning the return value
DD::open(wait:20);
window.update(title:"Dialog");


// operation expression as an assignment statement read value
volume = DD::get_volume();


// operation expression within a while loop
while (MOD::status() != 1)
       value = this_mod.poll();
```

```
end while;


// stand-alone operation assignment statement
branch = TR::get_next_branch();
```

Bridge Invocation

Syntax

As a bridge expression:

```
<eekeyletter>::<bridge name> (data item>:<expression>, ...)
```

As a stand-alone bridge assignment statement:

```
<variable> = <eekeyletter>::<bridge name> (<data item>:
<expression>, ...);
```

<eekeyletter> are the keyletters of an external entity.

<bridge name> is the name of a bridge assigned to the external entity.

<data item> is the name of a data item input to *<bridge name>* .

<expression> is a string, arithmetic, boolean, simple or compound expression. The data type of the expression must match the data type defined for the given data item.

<variable> is a class attribute or a local variable.

Notes

It is strongly recommended that BridgePoint UML Suite's external entities be used only to represent domains.

A bridge expression may be used anywhere an expression is valid (e.g., assignment statement read values, control logic expressions, etc.).

The stand-alone bridge assignment statement must always be a stand-alone statement and cannot be used as an expression within another statement.

Since a bridge expression may be used anywhere an expression may be used, stand-alone bridge assignment statements are not necessary. The user may simply use a bridge expression as a read value and use an *assign* statement to assign the return value to *<variable>* .

Parentheses are required even if there are no data items.

If *<variable>* is a local variable that is being assigned for the first time, it will be of the same data type as the output value of *<bridge name>* .

If the type of *<variable>* has already been established (that is, if it is the attribute of a class or a local variable that has previously been assigned), then either:

- *<variable>* must be of the same data type as the output value of the bridge,
  or
- the output value of the bridge is of type *integer* or *real* and *<variable>* is of type *integer* or *real* .

Example

```
// bridge expression without assigning the return value
OT::start();


// bridge expression as an assignment statement read value
cur_time = bridge TIM::current_time();


// bridge expression within the test part of an if statement
if (TIM::timer_add_time(timer_inst_ref:my_timer,
microseconds:500))
       wait = wait + 500;
end if;


// stand-alone bridge assignment statement
bridge my_timer = TIM::timer_start(microseconds:500, event_inst:
my_evt);


// State 4. "Raising"
needle_position = SPPIO::raise_needle (
         radial_position:self.radial_position,
```

```
theta_offset:self.theta_offset,
probe_id:self.probe_ID );
```

Avoiding Ambiguity in Invocations

Ambiguity among class and bridge operations can arise if the following conditions are present within a single domain:

- An external entity (EE) and a class share the same keyletters.
- The EE has a bridge operation with the same name as a class operation associated with the class.

When set, the Class Keyletters check-box under *Preferences* | *User* | *Audits* will report any EE's and Classes that share the same key letters. The key letter ambiguity should be removed by changing the key letter of either the EE or the class.

Function Invocation

A function is an operation that is global to the domain being modeled. Unlike class operations and bridge operations, a function's scope is at the domain level.

Syntax

As a function expression:

```
::<function name> (data item>:<expression>, ...)
```

As a stand-alone function assignment statement:

```
<variable> = ::<function name> (<data item>:<expression>, ...);
```

<function name> is the name of a function.

<data item> is the name of a data item input to *<function name>* .

<expression> is a string, arithmetic, boolean, simple or compound expression. The data type of the expression must match the data type defined for the given data item.

<variable> is a class attribute or a local variable.

Notes

It is strongly recommended that functions be named according to some convention that conveys their meaning throughout the domain.

A function expression may be used anywhere an expression is valid (e.g., assignment statement read values, control logic expressions, etc.).

The stand-alone function assignment statement must always be a stand-alone statement and cannot be used as an expression within another statement.

Since a function expression may be used anywhere an expression may be used, stand-alone function assignment statements are not necessary. The user may simply use a function expression as a read value and use an *assign* statement to assign the return value to <*variable*> .

Parentheses are required even if there are no data items.

If <*variable*> is a local variable that is being assigned for the first time, it will be of the same data type as the output value of <*function name*> .

If the type of <*variable*> has already been established (that is, if it is the attribute of a class or a local variable that has previously been assigned), then either:

- <*variable*> must be of the same data type as the output value of <*function name*>
  or
- the output value of <*function name*> is of type integer or real and <*variable*> is of type *integer* or *real* .

Example

```
// function expression without assigning the return value
::start();


// function expression as an assignment statement read value
cur_ext_time = ::current_external_time();


// function expression within the test part of an if statement
if ( ::shutdown() )
             generate S1:'Shutdown'() to S Creator;
end if;
```

```
// stand-alone function assignment statement
status = ::shutdown();
```

Object Action Language for Invocations

OAL can be specified for class and instance-based operations, bridge operations, and functions. The syntax rules applied to actions for these differ slightly from those applied to state actions. These differences are described in detail below.

Return Statement

Since class operations, bridge operations, and functions can return a value, the *return* statement is accepted within their actions.

Syntax

```
return <expression>;
return;
```

<expression> is a string, arithmetic, boolean, simple or compound expression. The data type of the expression must match the data type defined for the return value of the class operation, bridge operation, or function.

Notes

When executed, the *return* statement causes control to be returned to the caller.

The value returned to the caller is *<expression>* .

If the return value of the class operation, bridge operation, or function is void, then *<expression>* must be omitted.

Example

```
select any dog from instances of DOG;
return dog.weight;


// SSPIO:  Bridge "Lower needle"
select any probe from instances of SP
         where selected.probe_ID == param.probe_id;
generate SP3:probe_in_position to probe;
```

```
return "down";
```

Parameters

Class and instance-based operations, bridge operations, and functions can accept parameters. These parameters can be accessed as read values by using the *param* keyword within the action. They can be modified if they are by-reference parameters.

Syntax

```
param.<parameter>
```

<parameter> is the name of a parameter.

Note

param.<parameter> is an *<expression>* and so can be used in any OAL construct specifying an expression.

Example

```
// For an invocation like MATH::SQR(x:3)
// Return x**2
return param.x * param.x;
```

Use of self keyword

Instance-based operations can use the keyword *self* to reference the instance to which the operation is currently being applied. The *self* keyword can be used anywhere that is valid.

Note

Class-based operations, bridge operations, and functions can not use the *self* keyword since they are not related to a specific instance.

Example

```
// attribute access
self.a = self.b;
// event generation
generate E1:'one'() to self;
```

Other Differences

Class and instance-based operations, bridge operations, and functions may not refer to the *rcvd_evt* keyword. This keyword is only allowed within a state action since these are the only actions that can receive events.

Components and Interfaces

Interface Invocation

The analyst may send messages between components. The term ‘message’ is used hereafter to mean either sending a signal or calling an operation on a component.

Syntax

As a message expression:

```
[send] <Interface Name>::<Message Name>(<Message Parameter Name>:
<value>) [to <Component Reference>]
```

or

```
[send] <Port Name>::<Message Name>(<Message Parameter Name>:
<value>) [to <Component Reference>]
```

As a stand-alone operation assignment statement:

```
<variable> = [send] <Interface Name>::<Message Name>(<Message
Parameter Name>: <value>) [to <Component Reference>];
```

or

```
<variable> = [send] <Port Name>::<Message Name>(<Message Parameter
Name>: <value>) [to <Component Reference>];
```

If a port has no name, then the first syntax is mandatory. If a component requires or provides the same interface more than once, then a uniquely named port is required for one or both of the ports. The alternative syntax proposed above is required for this situation. In both syntaxes above, the keyword *'send'* is optional. The same syntax allows is used for both synchronous and asynchronous inter-component communication. The analyst can optionally specify an explicit component reference to which the message should be sent.

<Interface Name> is the name of the target interface that contains the message being sent.

<Port Name> is the name of the target port though which the message is being sent.

<Message Name> is the name of the message being sent.

<Message Parameter Name> is the name of a parameter of the message.

<value> is the value of the associated message parameter.

<Component Reference> is the name of the component reference (as defined in the user's OAL) to send the message to.

<variable> is a class attribute or a local variable.

Notes

General

A message expression may be used anywhere an expression is valid (e.g., assignment statement read values, control logic expressions, etc.).

The stand-alone message must always be a stand-alone statement and cannot be used as an expression within another statement.

Since an message expression may be used anywhere an expression may be used, stand-alone message assignment statements are not necessary. The user may simply use a message expression as a read value and use an *assign* statement to assign the return value to *<variable>*.

Parentheses are required even if there are no data items.

If *<variable>* is a local variable that is being assigned for the first time, it will be of the same data type as the return value of *<Message Name>*.

If the type of *<variable>* has already been established (that is, if it is the attribute of a class or a local variable that has been previously assigned), then either:

- *<variable>* must be of the same data type as the output value of the operation,
  or
- the output value of the message is of type *integer* or *real* and *<variable>* is of type *integer* or *real*.

Implicit Component Addressing

There are two cases where there is no ambiguity about which component is the recipient of a given message. One, when a component interface is 1 to 1. Two, when architectural support is turned on for caching component identity for a given thread of control. If the optional target component reference is not specified, then the default component is determined by the architecture. Implicit component addressing requires special architectural support, and there are some cases where component ambiguity exists. To allow users to avoid potential problems associated with these situations, a per model preference (*`Window > Preferences > BridgePoint > Action Language > Allow implicit component addressing`*) shall force all signal references to use explicit component addressing only. When this preference is selected, the *`'sender'`* keyword shall be legal in interface activities only, where it shall refer to the identity of the component that triggered the interface activity.

Implicit and explicit component addressing may be mixed (always provided implicit component addressing is enabled). So, where a component other than the default component must be addressed, this can be achieved by caching and explicitly addressing the desired component.

Name Scoping and Precedence

The syntax allows for ambiguities with other pre-existing model elements. These are class operations and bridges. This is resolved with a precedence hierarchy. Precedence shall flow from local to global; i.e. operation, message, then bridge. The default may be overridden as required, using the *`'send'`* or *`'bridge'`* keywords.

Example

```
// messages sent without assigning the return value
Interface1::Signal2(parm1: 3);
send Port5::Signal1();

// message as an assignment statement read value
result = Controller::RequestTime(unit:Time::milliseconds);

// send message to specific component references
send Port3::Signal11(parm1:rcvd_evt.x, parm2:rcvd_evt.y) to rcvd_evt.
requestor;
```

Use of sender keyword

The default component may be addressed using the syntax, *`'to sender'`* or by omitting the *`'to <Component Reference>'`* syntax altogether. The *`sender`* keyword can be used anywhere that is valid.

Note

When Implicit Component Addressing is disallowed by the user preference, the *`'sender'`* keyword is

legal in interface activities only, where it refers to the identity of the component that triggered the interface activity. Assuming no previous OAL, creating a local variable of type *'component_ref'* called *'sender'* is perfectly legal in all contexts.

Example

```
send TXRXPort::ACK() to sender;
```

Timers

Starting a Timer

Syntax

```
[bridge] <timer handle> = TIM::timer_start (microseconds:
<arithmetic expression>, event_inst:<event instance>);
```

<timer handle> is a handle to a timer instance.

<arithmetic expression> is an expression that evalues to an integer value.

<event instance> is a handle to an event instance.

This bridge operation starts a timer set to expire in *<arithmetic expression>* microseconds, generating the event *<event instance>* upon expiration. Returns the instance handle of the timer.

The *bridge* keyword is optional.

```
[bridge] <timer handle> = TIM::timer_start_recurring
(microseconds:<arithmetic expression>, event_inst:<event
instance>);
```

<timer handle> is a handle to a timer instance.

<arithmetic expression> is an expression that evalues to an integer value.

<event instance> is a handle to an event instance.

This bridge operation starts a timer set to expire in *<arithmetic expression>* microseconds, generating the event *<event instance>* upon expiration. Upon expiration, the timer will be restarted and fire again in *<arithmetic expression>* microseconds generating the event *<event instance>* . This bridge operation returns the instance handle of the timer.

The *bridge* keyword is optional.

Example

```
// State 3. "Down"
```

```
select one row related by self->ROW[R2];
st = row.sampling_time;
create event instance move_on of SP1:finished_sampling() to
self;
mo_timer = TIM::timer_start(microseconds:st, event_inst:
move_on);
```

Querying a Timer

Syntax

```
[bridge] <integer variable> = TIM::timer_remaining_time
(timer_inst_ref:<timer handle>);
```

<integer variable> is a local variable or a class attribute of type *integer* .

<timer handle> is a handle to a timer instance.

Returns the time remaining (in microseconds) for the timer specified by *<timer handle>* . If the timer has expired, a zero value is returned.

The *bridge* keyword is optional.

Manipulating a Timer

Syntax

```
[bridge] <boolean variable> = TIM::timer_reset_time
(timer_inst_ref:<timer handle>, microseconds:<arithmetic
expression>);
```

<boolean variable> is a local variable or a class attribute of type *boolean* .

<timer handle> is a handle to a timer instance.

<arithmetic expression> is an expression that evalues to an integer value.

This bridge operation attempts to set an existing timer *<timer handle>* to expire in *<arithmetic expression>* microseconds. If the timer exists (that is, it has not expired), a *TRUE* value is returned. If the timer no longer exists, a *FALSE* value is returned.

```
[bridge] <boolean variable> = TIM::timer_add_time
(timer_inst_ref:<timer handle>, microseconds:<arithmetic
expression>);
```

<boolean variable> is a local variable or a class attribute of type *boolean* .

<timer handle> is a handle to a timer instance.

<arithmetic expression> is an expression that evalues to an integer value.

This bridge operation attempts to add *<arithmetic expression>* microseconds to an existing timer *<timer handle>* . If the timer exists (that is, it has not expired), a *TRUE* value is returned. If the timer no longer exists, a *FALSE* value is returned.

The *bridge* keyword is optional.

Canceling a Timer

Syntax

```
[bridge] <boolean variable> = TIM::timer_cancel (timer_inst_ref:
<timer handle>);
```

<boolean variable> is a local variable or a class attribute of type *boolean* .

<timer handle> is a handle to a timer instance.

This bridge operation cancels and deletes the timer specified by *<timer handle>* . If the timer exists (that is, it had not expired), a *TRUE* value is returned. If the timer no longer exists, a *FALSE* value is returned.

The *bridge* keyword is optional.

Notes

When a timer fires, it is deleted unless it was created using the *timer_start_recurring* bridge operation.

In many architectures there may be a delay between the expiration of a timer and the delivery of the associated event to the receiving state machine.

Syntax Summary

Language Constructs

Control Logic

```
if (<boolean expression>)
  // Executed if above boolean expression evaluates to TRUE
  <statements>
elif (<boolean expression>)
  // Executed if above boolean expression evaluates to TRUE and
previous
  // boolean expression is FALSE
  <statements>
else
  // Executed if both boolean expressions evaluate to FALSE
  <statements>
end if;
```

```
for each <instance handle> in <instance handle set>
  <statements>
end for;
```

```
while <boolean expression>
  <statements>
end while;
```

```
break;
```

```
continue;
```

Instance Creation

```
create object instance <instance handle> of <keyletter>;
create object instance of <keyletter>;
```

Instance Selection

```
select any <instance handle> from instances of <keyletter>
[ where <where   expression> ];
select many <instance handle set> from instances of <keyletter>
[ where   <where expression> ];
```

Writing Attributes

```
[assign] <instance handle>.<attribute> = <expression>;
[assign] self.<attribute> = <expression>; // Mathematically-
dependent only
```

Reading Attributes

```
[assign] <variable> = <instance handle>.<attribute>;
```

Instance Deletion

```
delete object instance <instance handle>;
```

Creating Instances of a Relationship

```
relate <source instance handle> to <destination instance
handle> across   <relationship specification>;
relate <source instance handle> to <destination instance
handle> across   <relationship specification> using
<associative instance handle>;
```

Deleting Instances of a Relationship

```
unrelate <source instance handle> from <destination instance
handle> across   <relationship specification>;
unrelate <source instance handle> from <destination instance
handle> across   <relationship specification> using
<associative instance handle>;
```

Instance Selection by Relationship Navigation

```
select one <instance handle> related by <start> ->
<relationship link   chain> [ where <where expression> ];
select any <instance handle> related by <start> ->
<relationship link   chain> [ where <where expression> ];
select many <instance handle set> related by <start> ->
<relationship link   chain> [ where <where expression> ];
```

Creating Events

```
create event instance <event instance> of <event label>[:<event
meaning>]   [(<event parameters>)] to <target>;
```

Generating Events

```
generate <event label>[:<event meaning>] [(<event parameters>)]
to   <target>;
generate <event instance>;
generate <instance handle>.<attribute>;
```

Accessing Event Data

```
rcvd_evt.<supplemental data item>
```

Arithmetic, Logical, and String Assignment

```
[assign] <boolean var> = <boolean expression>;
[assign] <arithmetic var> = <arithmetic expression>;
[assign] <string var> = <string expression>;
```

Unary Operators

```
empty <handle>
not_empty <handle>
cardinality <handle>
```

Operations

```
[transform] <keyletter>::<operation name> (<data item>:
<expression>, ...);
[assign] <variable> = [transform] <keyletter>::<operation name>
(<data   item>:<expression>, ...);
```

Bridges

```
[bridge] <eekeyletter>::<bridge name> (<data item>:
<expression>, ...);
[assign] <variable> = [bridge] <eekeyletter>::<bridge name>
(<data item>:  <expression>, ...);
```

Functions

```
::<function name> (<data item>:<expression>, ...);
[assign] <variable> = ::<function name> (<data item>:
<expression>, ...);
```

Inter-Component Messaging

```
send <Interface Name>::<Message Name>(<Message Parameter Name>:
<value>);
send <Port Name>::<Message Name>(<Message Parameter Name>: <value>);
```

Object Action Language for Non-State Actions

```
return <expression>;
return;
param.<parameter>
```

Date and Time

```
[bridge] <date variable> = TIM::create_date (day:<arithmetic
expression>,   month:<arithmetic expression>, year:<arithmetic
expression>, second:  <arithmetic expression>, minute:
<arithmetic expression>, hour:  <arithmetic expression>);
[bridge] <date variable> = TIM::current_date ();
[bridge] <time variable> = TIM::current_clock ();
<integer variable> = TIM::get_day (date:<date variable>);
<integer variable> = TIM::get_month (date:<date variable>);
<integer variable> = TIM::get_year (date:<date variable>);
<integer variable> = TIM::get_second (date:<date variable>);
<integer variable> = TIM::get_minute (date:<date variable>);
<integer variable> = TIM::get_hour (date:<date variable>);
```

Timers

```
[bridge] <timer handle> = TIM::timer_start (microseconds:
<arithmetic   expression>, event_inst:<event instance>);
[bridge] <timer handle> = TIM::timer_start_recurring
(microseconds:  <arithmetic expression>, event_inst:<event
instance>);
[bridge] <integer variable> = TIM::timer_remaining_time
(timer_inst_ref:  <timer handle>);
[bridge] <boolean variable> = TIM::timer_reset_time
(timer_inst_ref:<timer   handle>,     microseconds:<arithmetic
```

```
expression>);
[bridge] <boolean variable> = TIM::timer_add_time
(timer_inst_ref:<timer   handle>,     microseconds:<arithmetic
expression>);
[bridge] <boolean variable> = TIM::timer_cancel (timer_inst_ref:
<timer   handle>);
```

Statement Components

<arithmetic expression>	an expression evaluating to a real or an integer value. Used in places where only an integer is required.
<arithmetic operator>	*a* + , - , * , / , or *%* (remainder from arithmetic division)
<attribute>	the name of an attribute
<binary operator>	an and, or, + , - , * , / , or *%*
<boolean expression>	an expression evaluating to *TRUE* or *FALSE*
<bridge name>	the name of a bridge defined for the external entity specified by <eekeyletter>.
<eekeyletter>	the keyletter of an external entity
<event instance>	a local variable of type event instance
<event label>	the <keyletter><event number>
<event meaning>	the meaning of the event, as in 'turn off the light'; the tick marks may be omitted if the event meaning contains no spaces.

<event parameters>	the supplemental data items (if any) to be carried by the event. Each data item is given in the form <supplemental data item>:<expression>.
<handle>	an <instance handle>, <instance handle set>, or a timer handle (or a bridge, operation, or function invocation that returns a timer handle).
<instance handle>	a local variable referring to a single instance
<instance handle set>	a local variable referring to a set of instance handles
<keyletter>	the key letters of a class
<operation name>	the name of a operation defined for the class specified by <keyletter>.
<read value>	a readable value: a constant, local variable, <instance handle>.<attribute>, `rcvd_evt` .<supplemental data item>, `param` .<parameter>, or an invocation of an operation, bridge operation, or function.
<relationship link>	a <keyletter>[<relationship specification>], where the square brackets are literal and do not indicate optional text.
<relationship link chain>	<relationship link> or <relationship link> -> ... -> <relationship link>
<relationship phrase>	the text description of the relationship enclosed in tick marks
<relationship specification>	R<number> or R<number>.<relationship phrase>
<simple string>	any of the following that evaluates to a string value: string constant, local variable, attribute, operation invocation, bridge invocation, function invocation, or a supplemental data item received from an event.
<string expression>	an expression evaluating to a string value
<supplemental data item>	the name of a supplemental data item
<unary operator>	a unary operator such as `not` for boolean expressions and + and - for arithmetic expressions.

<unique id>	a variable of type unique ID (or an operation, bridge or function invocation that returns a unique ID).
<where expression>	a special boolean expression at the end of a select statement; it must contain the *selected* keyword.

Production Rules

This section contains the production rules for the language expressed in EBNF.

action ::= block EOF ;

block ::= (statement)* ;

statement ::= (implicit_ib_transform_statement | function_statement | implicit_assignment_statement | implicit_invocation_statement | assignment_statement | control_statement | break_statement | bridge_statement | send_statement | continue_statement | create_object_statement | create_event_statement | delete_statement | for_statement | generate_statement | if_statement | relate_statement | return_statement | select_statement | transform_statement | while_statement | unrelate_statement | debug_statement | empty_statement) Semicolon ;

assignment_statement ::= ASSIGN assignment_expr ;

break_statement ::= BREAK ;

bridge_statement ::= BRIDGE (((member_access | param_data_access) EQUAL bridge_invocation) | bridge_invocation) ;

send_statement ::= SEND (((member_access | param_data_access) EQUAL message_invocation) | message_invocation) ;

continue_statement ::= CONTINUE ;

create_event_statement ::= CREATE EVENT INSTANCE local_variable OF event_spec ;

create_object_statement ::= CREATE OBJECT INSTANCE ((local_variable OF)=> local_variable)? OF object_keyletters ;

delete_statement ::= DELETE OBJECT INSTANCE inst_ref_var ;

empty_statement ::= ;

for_statement ::= FOR EACH local_variable IN inst_ref_set_var block END FOR ;

generate_statement ::= GENERATE (event_spec | member_access) ;

if_statement ::= IF expr block ((ELIF expr block)+)? (ELSE block)? END IF ;

implicit_assignment_statement ::= assignment_expr ;

implicit_invocation_statement ::= invocation ;

implicit_ib_transform_statement ::= transform_ib_invocation ;

relate_statement ::= RELATE inst_ref_var TO inst_ref_var ACROSS relationship (DOT phrase)?
(USING assoc_obj_inst_ref_var)? ;

return_statement ::= RETURN (expr)? ;

select_statement ::= SELECT (ONE local_variable object_spec | ANY local_variable object_spec | MANY
local_variable object_spec) ;

transform_statement ::= TRANSFORM (((member_access | param_data_access) EQUAL
transform_invocation) | transform_invocation) ;

function_statement ::= DOUBLECOLON function_invocation ;

unrelate_statement ::= UNRELATE inst_ref_var FROM inst_ref_var ACROSS relationship (DOT
phrase)? (USING assoc_obj_inst_ref_var)? ;

while_statement ::= WHILE expr block END WHILE ;

assignment_expr ::= (member_access EQUAL expr |(PARAM DOT)=> param_data_access EQUAL
expr) ;

bridge_invocation ::= ee_keyletters DOUBLECOLON bridge_function LPAREN (invocation_parameters)?
RPAREN ;

message_invocation ::= interface_or_port_name DOUBLECOLON message_name LPAREN
(invocation_parameters)? RPAREN (TO (rval))? ;

invocation ::= identifier DOUBLECOLON invocation_function LPAREN (invocation_parameters)?
RPAREN ;

bridge_expr ::= BRIDGE bridge_invocation ;

```
invocation_expr ::= invocation ;

enumerator_access ::= enum_data_type DOUBLECOLON enumerator ;

event_spec ::= event_label ( TIMES )? ( COLON event_meaning )? ( LPAREN ( supp_data )? RPAREN )?
TO ( ( ( object_keyletters CLASS )=>  object_keyletters CLASS | object_keyletters CREATOR ) |
( inst_ref_var_or_ee_keyletters ) ) ;

invocation_parameters ::= data_item COLON expr ( COMMA data_item COLON expr )* ;

inst_ref_var_or_ee_keyletters ::= ( local_variable | GENERAL_NAME | kw_as_id2 ) ;

interface_or_port_name ::= general_name ;

message_name ::= general_name ;

instance_chain ::= ( ARROW object_keyletters LSQBR relationship ( DOT phrase )? RSQBR )+ ;

object_spec ::= ( RELATED BY local_variable instance_chain ( where_spec )? | FROM INSTANCES OF
object_keyletters ( where_spec )? ) ;

supp_data ::= supp_data_item COLON expr ( COMMA supp_data_item COLON expr )* ;

function_invocation ::= function_function LPAREN ( invocation_parameters )? RPAREN ;

transform_ib_invocation ::= inst_ref_var DOT transformer_function LPAREN ( invocation_parameters )?
RPAREN ;

transform_invocation ::= object_keyletters DOUBLECOLON transformer_function LPAREN
( invocation_parameters )? RPAREN ;

where_spec ::= WHERE expr ;

assoc_obj_inst_ref_var ::= inst_ref_var ;

bridge_function ::= function_name ;

invocation_function ::= function_name ;

data_item ::= data_item_name ;

data_item_name ::= general_name ;
```

enum_data_type ::= general_name ;

enumerator ::= general_name ;

keyletters ::= general_name ;

ee_keyletters ::= keyletters ;

event_label ::= general_name ;

event_meaning ::= (phrase) ;

general_name ::= (limited_name | GENERAL_NAME | kw_as_id1 | kw_as_id2 | kw_as_id3) ;

svc_general_name ::= (limited_name | GENERAL_NAME | kw_as_id1 | kw_as_id2 | kw_as_id3 | kw_as_id4) ;

limited_name ::= ID | RELID ;

inst_ref_set_var ::= local_variable ;

inst_ref_var ::= local_variable ;

local_variable ::= root_element_label ;

root_element_label ::= (SELECTED | SELF | limited_name | kw_as_id1) ;

element_label ::= general_name ;

function_name ::= general_name ;

svc_function_name ::= svc_general_name ;

identifier ::= general_name ;

object_keyletters ::= keyletters ;

phrase ::= (TICKED_PHRASE | svc_general_name) ;

relationship ::= RELID ;

supp_data_item ::= data_item_name ;

function_function ::= svc_function_name ;

transformer_function ::= function_name ;

expr ::= sub_expr ;

sub_expr ::= conjunction (OR conjunction)* ;

conjunction ::= relational_expr (AND relational_expr)* ;

relational_expr ::= addition (comparison_operator addition)? ;

addition ::= multiplication (plus_or_minus multiplication)* ;

multiplication ::= boolean_negation | sign_expr (mult_op sign_expr)* ;

sign_expr ::= (plus_or_minus)? term ;

boolean_negation ::= NOT term ;

term ::= cardinality_op | empty_op | not_empty_op | rval | LPAREN (assignment_expr | expr) RPAREN ;

cardinality_op ::= CARDINALITY local_variable ;

empty_op ::= EMPTY local_variable ;

not_empty_op ::= NOT_EMPTY local_variable ;

instance_start_segment ::= root_element_label (array_refs)? ;

instance_access_segment ::= element_label (array_refs)? ;

member_access ::= instance_start_segment (DOT instance_access_segment)* ;

param_data_access ::= PARAM DOT member_access ;

event_data_access ::= RCVD_EVT DOT member_access ;

array_refs ::= (LSQBR expr RSQBR)+ ;

rval ::= DOUBLECOLON function_invocation | transform_ib_invocation | invocation_expr | enumerator_access | member_access | constant_value | (RCVD_EVT DOT) => event_data_access | bridge_expr | (PARAM DOT) => param_data_access | QMARK ;

constant_value ::= FRACTION | NUMBER | STRING | TRUE | FALSE ;

comparison_operator ::= DOUBLEEQUAL | NOTEQUAL | LESSTHAN | LE | GT | GE ;

plus_or_minus ::= PLUS | MINUS ;

mult_op ::= TIMES | DIV | MOD ;

References

References

The following published works were used in compiling this manual.

Mel02 Stephen J. Mellor and Marc J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture,* Addison-Wesley, Boston, MA, 2002

Shl92 Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States,* Prentice Hall, Englewood Cliffs, 1992

Shl95 Sally Shlaer and Neil Lang, *Shlaer-Mellor Method: The OOA96 Report,* Project Technology, Inc., Berkeley, California, 1995

Wil95 Ian Wilkie, Adrian King, and Mike Clarke, *The Action Specification Language (ASL) Reference Guide,* version 2.4, Kennedy-Carter, London, 1995