BRIDGEPOINT® 3.3

Action Language Manual

Project Technology, Inc.
10940 Bigge Street
San Leandro, CA 94577
USA

Product Information: 800/845-1489

Product Support:
support@projtech.com
520/544-2912 (Fax)
800/482-3853 (Voice)
520/544-0808 (Voice)
http://www.projtech.com/support/bpcsa.html

User ID:_________________ Password:________________

Document Version 1.0

This manual describes the action language used in the **BRIDGEPOINT** products. The terminology, organization and much of the supporting text in this manual are based on *The Action Specification Language (ASL) Reference Guide* [Wil95] published by Kennedy-Carter. Project Technology hereby acknowledges Kennedy-Carter's copyright material (here used with permission) as well as other contributions that they have made to the development of the Shlaer-Mellor Method.

Manufacturer is
Project Technology, Inc.
10940 Bigge Street
San Leandro, CA 94577
USA

Table of Contents

1. INTRODUCTION

This chapter describes the purpose and intended audience of this manual and provides usage hints and conventions used throughout the following chapters.

1.1 PURPOSE AND AUDIENCE

This section explains the intent of this document and describes its usage.

1.1.1 Purpose

The purpose of this manual is to serve as a reference and general user's guide to aid in the correct specification of state actions in *Shlaer-Mellor OOA* models entered in the BridgePoint Model Builder.

An action language is used to define the processing which is executed upon entry to a state of a state model. The BridgePoint action language is written so that it satisfies the following goals:

- Translation - Richness of expression is provided while maintaining a specification that can be automatically translated onto a target architecture.
- Readability - Analysts must be able to easily understand the action semantics for development and reviews.
- Simulation - The OOA can be simulated through interpretation of the state actions.
- Derivation - Event generation and data access information is captured for derivation of the OCM, SCM, OAM and SAM.

1.1.2 Basic Concepts

The BridgePoint action language provides the five types of action processes as defined in *Shlaer-Mellor OOA*. These include:

- data access
- event generation
- test
- transformation
- bridge

The action language supports these through:

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

In OOA, unlike conventional programming, there is no concept of a "main" function or routine where execution starts. Rather, the OOA 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[1] execute at the same time as processing associated with another state machine.

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

- Execution commences at the first action language statement in the state 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.

1.1.3 Intended Audience

This manual is written for analysts and software engineers who are using the Shlaer-Mellor development method and will capture their analysis models in BridgePoint Model Builder. Specifically, guidance is provided for the correct specification of state actions formulated in the syntax of the BridgePoint action language.

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

1.1.4 Conventions

Considerable development work is being done on this action language in order to:

1. Whether this occurs in practice depends on the nature of the software and hardware architectures used to implement the system.

- increase its expressive power and readability.
- support the features of OOA96 [see reference Shl95].
- allow complete system generation capabilities (in addition to model translation) to be added to the BridgePoint tool suite.

Although future versions of the tools will support models created under this release, this manual is annotated with three special symbols to indicate preferred usage in the light of future development.

☺ Indicates a preferred construct or recommended use for a given construct.

😐 Indicates a feature that should be used only in the limited manner described in this manual. If the feature is used for other purposes, it is unlikely to convert properly in future releases or translate correctly onto some architectures.

☹ Marks a form or capability that is included in this release only for experimental reasons or for backward compatibility. The capability may not be supported in future releases.

Some syntactical conventions are used throughout this manual to clarify meaning whenever necessary.

- Square brackets are used to indicate optional text when describing BridgePoint Action Language syntax, except where otherwise noted.
- The `courier` font is used when referring to BridgePoint Action Language keywords in order to highlight the importance of the keyword, and to avoid ambiguity within a block of text.

1.1.5 Examples

There are examples presented throughout the manual. Much of the action language in these examples was extracted from a complete model of a real-world device (an auto-sampler used for chemical testing). The complete model is available on the Project Technology web site (`www.projtech.com`). This model is intentionally simplistic in nature to make it easy to understand. Its purpose is to illustrate as many aspects of the action language constructs as possible. It should, therefore, not be construed an example of good OOA modeling practices as several compromises have been made in the interest of simplicity.

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 action language construct in question. As with any contrived examples, these sequences of action language are provided as examples of the syntax for the language and not as examples of valid modeling techniques for real world applications.

2. LANGUAGE STRUCTURE

This chapter describes the structure and components of the action language.

2.1 LANGUAGE STRUCTURE

This section explains the structure of a state action.

2.1.1 Overall Structure

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

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

2.1.2 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.

2.1.3 Names and Keywords

Action language statements are composed of:

- keywords (See Appendix B: Keywords.)
- logical and arithmetic operations
- names of OOA elements (objects, attributes, object 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 action language statements must conform to the following rules:

1. Names are case sensitive. This includes local variables, keyletters, object attributes, and event ids.

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.[1]
   - must be contained on a single line.

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

2.1.4 White Space

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

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

2.1.5 Action Language Enhancements

Some enhancements have been added to the action language since the 3.2.1 release of Bridge-Point.

- Use of `assign`, `bridge`, and `transform` keywords is optional.
- The optional `where` clause is supported for use in `select` statements.
- The `elif` keyword may be used within `if` constructs.
- The `while` loop is a valid construct.
- `Break` and `continue` statements are supported for both `for each` and `while` loops.
- The capability of expressions is extended to include compound expressions.
- `Self` may be used anywhere an instance reference can be used, except where `self` is explicitly not allowed.

1. The tick marks may be omitted from an event meaning if it contains no spaces.

- Event instances may be created with external entities, assigners, and creators as valid destinations.
- The event meaning, designated by tick marks, is now optional for event generation and creation.
- Empty parentheses are optional for event generation and creation if there are no supplemental data items.
- Transformer and bridge invocations can be used as read values.

3. Data Items

This chapter describes how to read, write, and format the data and variables supported by the action language.

3.1 DATA ITEMS

This section outlines the types and usage of data.

3.1.1 Data Items Within an Action

The action language 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 objects
- 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 objects
- 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 state model (but not in an assigner state model).

3.1.2 OOA Elements

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

integer	string	unique ID	☹ event instance[a]
real	date	☹ state[b]	instance handle
boolean	😐 timer handle	handle set	time

Table 3-1: Action Language Data Types

a. Experimental feature, unlikely to be supported in future releases.

b. In the **OOA-96 Report**, the current state attribute is entirely under the control of the architectural domain and is not available to the analyst; as a result, in future versions this data type will no longer be needed.

The analyst may also define domain-specific data types based upon these data types. (See the BridgePoint OOA manual.)

Data types within the action language are "analysis" data types, and reflect only the set of legal values a variable can take on.

Note

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

3.1.3 Local Variables

Local variables can be of any of the data types listed in “Assigning Data Types” on page 13. Two of these types require special consideration:

instance handle	This is simply the identification of an instance of an object. The implementation of this type is entirely dependent on the architecture; hence, its form is unknown to the user of the action language.
handle set	A set of instance handles.

Because instance handles and 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 handle set may refer to an empty set. These situations can be detected by using the supplied unary set operators.

Notes

- Attributes of objects cannot be of type instance handle or handle set.
- Strictly speaking, a local variable is not of type instance handle, but rather of type instance handle for a particular object. Any attempt to use an instance handle for one object in the context of another is an error.
- In any action within an instance state machine, a special instance handle called `self` is always available. `Self` is always defined as an instance reference of the object that is executing the current action. The value of `self` cannot be changed by an action.
- `Self` is not defined for an assigner state model.

3.1.4 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 the BridgePoint Model Builder.

in this manual	**in Model 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>
handle set	inst_ref_set<Object>
event instance	inst<event>

Table 3-2: Names of data types used in this manual together with the corresponding names used in the BridgePoint Model Builder.

3.1.5 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 an object 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 Model Verifier, unassigned variables are considered UNDEFINED. The user should not intentionally read the data from UNDEFINED variables.

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

Example

```
// 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.
```

```
// Select an empty instance set.
select many f_set from instances of fish;
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.
```

3.1.6 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 action language for the given state, 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 action language for the current state.
- 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 state model is the entire action language for the current state.

Example

```
// begin state action

// scope1 - global scope (for this state).  Variables declared here
// are accessible anywhere in this state 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 state action
```

4. Control Structures

This chapter explains the flow of control statements provided in the action language.

4.1 CONTROL STRUCTURES

This section describes the action language statements that control the flow of the action language execution within a state action.

4.1.1 If Construct

Syntax

```
// Note that there is no semi-colon following the "if <boolean expression>"
if <boolean expression>
    // Executed if <boolean expression> is TRUE
    <statements>
end if;

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;
```

where:

<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

```
// Assign x with a different number for each name.
if (name == "John")
   x = 1;
elif (name == "Bill")
   x = 2;
elif (name == "Michael")
   x = 3;
```

```
// If not a known name, assign x to 4.
else
   x = 4;
end if;

// 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;
```

4.1.2 For Each Loop

Syntax

```
for each <instance handle> in <handle set> // Note no semi-colon after the 'for each . . ' construct
   <statements>
end for;
```

where:

<instance handle> is a local variable referring to a single 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 <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 implicitly typed as a variable of type <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;
```

4.1.3 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 after the 'while...' construct
   <statements>
end while;
```

where:

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

Notes

- 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;
```

4.1.4 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.

Notes

- 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;
```

4.1.5 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.

Notes

- 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;
```

4.1.6 Control Stop

The `control stop` statement is used to provide a breakpoint mechanism for simulation within the Model Verifier. Simulation stops at the end of any state action that has executed a `control stop` statement at any point in the action.

Syntax

control stop;

Notes

- Simulation is halted by stopping the clock tick.

- After a `control stop` statement has been executed within a state action, execution continues until the end of that state action is reached.

- If a `control stop` statement is nested within a conditional control construct (e.g., `if`, `while`, `for each`) and does not get executed during simulation, no break in execution is caused.

- Simulation can be resumed by restarting the clock tick.

Example

```
// start of state action
if ( self.debug == true )
   control stop;
end if;
// end of state action
```

4.1.7 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;
```

5. OBJECT MANIPULATIONS

This chapter explains the techniques supported for creating, deleting, selecting and accessing the attributes of object instances.

5.1 OBJECT MANIPULATIONS

This section describes statements that support the manipulation of object instances.

5.1.1 Creating Instances

Creation of an instance of an object is achieved by use of the `create` statement.

Syntax

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

create object instance of <keyletter>;

where:

<keyletter> is the keyletter(s) of an OOA object.

Notes

- The <instance handle> returned is the handle of the newly created instance of object <keyletter>. The handle can be used only to refer to an instance of object <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 objects.
create object instance car of C;
create object instance row of ROW;
create object instance probe of SP;
```

5.1.2 Selecting Instances

The `select` statement can be used to assign an instance or set of instances to either an instance handle or a 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 reference 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 <handle set> from instances of <keyletter>;

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

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

where:

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

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

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

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 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 the `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;
// Make selection based on relationships to other instances.
select one assignment related by self->PA[R2];
// Select a row that needs service.
select any row from instances of ROW
   where (selected.needs_probe == true);
```

5.1.3 Writing Attributes

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

Syntax

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

where:

<attribute> is the name of an 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 <instance handle>.

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.

Example

```
// Account is an object 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;
```

```
// 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;
```

5.1.4 Reading Attributes

An object attribute may be referenced in an expression using the form:

Syntax

<instance handle>.<attribute>

where:

<instance handle> is an instance handle.

<attribute> is the name of an attribute.

Notes

- You may read the value of any attribute, including referential attributes.
- <instance handle>.<attribute> is an expression and can be used in any action language 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;
```

5.1.5 Deleting Instances

Syntax

delete object instance <instance handle>;

Notes

- This statement deletes the instance specified by <instance handle>.
- The instance handle cannot be `self`.
- When an instance of an object is deleted, it is no longer available to the domain where the object is defined. However, sophisticated OOA 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 object 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 this_dog in dogs
   delete object instance this_dog;
end for;
```

6. RELATIONSHIPS

This chapter describes creation, deletion and navigation of object relationships.

6.1 RELATIONSHIPS

This section describes statements that support relationships.

6.1.1 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>

where:

<number> the number of the relationship as shown on the information model (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.

Notes

- 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’
```

6.1.2 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>;

where:

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

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

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

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

Notes

- The relationship specification should be framed as if navigating from source object to destination object.
- 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 Model Verifier unless the relationship is (M:M)-M.
- The `using` <associative instance> form must be 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 assigner;
generate ROW2:probe_assigned() to row;
```

6.1.3 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>;

where:

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

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

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

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

Notes

- The relationship specification should be framed as if navigating from the source object to the destination object.
- An attempt to unrelate two instances that are not related by the specified relationship is regarded as a run time error by the BridgePoint Model Verifier.
- The source, destination, and associative instance handles may be `self`.
- 😐 If an associative relationship is unrelated then the associative object instance(s) will not be deleted. The analyst must specify this explicitly.
- 😐 If an unconditional relationship is deleted, instances of participating objects will not automatically be deleted to remain consistent with the information model. It is the responsibility of the analyst to ensure that the IM 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;
```

6.1.4 Relationship Navigation

Relationship navigation is the function whereby relationships specified on the Information Model 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 <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 <handle set> related by <start> -> <relationship link> -> ... <relationship link>
    where <where expression>;
```

where:

<start> is <handle set> or <instance handle>.

<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 object reached by the specified relationship.

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

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

Notes

- Use the `select one` form if at most one instance handle can be returned by navigating the instance chain.
- Use the `select any` or `select many` form if more than one instance handle can be returned by navigating the instance 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 object at the end of the instance chain.
- The `select many` form returns a handle set containing all the instances of the object at the end of the instance chain.
- The `select any ... where` form returns the instance handle of an arbitrary instance of the object at the end of the instance chain that fulfills the <where expression> requirement.
- The `select many ... where` form returns a handle set containing all the instances of the object at the end of the instance chain that fulfill the <where expression> requirement.
- The relationship phrases in the instance chain must be given in the direction of navigation.
- If the starting <instance handle> or <handle set> is empty, then the result will be considered a run time error.
- The returned <instance handle> or <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 meet the criteria of <where expression>. This implies that the instance handle or the handle set may be empty if no instance fulfills the criteria.

Examples

```
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
   where ( 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";
```

7. Events

This chapter explains support in the action language for event creation, generation and data reception.

7.1 EVENTS

This section describes support for events.

7.1.1 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>

where:

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

Notes

- `rcvd_evt`.<supplemental data item> is an <expression> and so can be used in any action language construct specifying an expression.

Example

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

// State 2. "Going"
self.destination = rcvd_evt.destination;
```

7.1.2 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>;

where:

<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 an object, <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> assigner.
- For an event directed at an external entity, <target> is the external entity's keyletters.

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 assigner;

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

7.1.3 Event Pre-creation

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

- 😐 to create an event for an OOA timer.
- 😐 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>;
```

where:

<event label>, <event meaning>, <event parameters> and <target> are as defined in “Event Generation” on page 37.

<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.
- Please refer to the notes in “Event Generation” on page 37.

7.1.4 Sending a Pre-created Event

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

Syntax

```
generate <event instance>;
```

where:

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

Notes

- Sends a pre-created event to its intended recipient.

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

8. EXPRESSIONS

This chapter illustrates the use of arithmetic and logical expressions and their use in variable assignment.

8.1 EXPRESSIONS

This section describes mathematical expressions and variable assignment.

8.1.1 Simple Expressions

Simple expressions are single unary or binary operations. An expression is not a complete action language statement, but is evaluated as part of a full action language 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>

where:

<read value> is a constant, a local variable, the attribute of an object, a supplemental data item received from an event, a transformer invocation, or a bridge invocation

<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 handle and 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 handle and handle set read values, the binary operators are `==` and `!=`. For string read values, the binary operator is `+`.

Examples

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

8.1.2 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>

where:

<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 an object, a supplemental data item received from an event, a transformer invocation, or a bridge 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 action language 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 action language statements using
// compound expressions:
if ((i == 1) AND (name == "Doug"))
   assign x = 0.5 * (y + z);
end if;

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

8.1.3 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>

where:

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

<unary arithmetic operator> is + or -.

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

Notes

- 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())
```

8.1.4 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>

where:

<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 a compound expression that evaluates to a boolean value.

<boolean operator> is a logical operator (refer to Table 8-1 on page 46).

<time value> a date or a timestamp variable (or a transformer or bridge invocation that returns a date or a timestamp).

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

<unique id> a variable of type unique ID (or a transformer or bridge invocation that returns a unique ID).

Notes

- 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
`!=`	not-equals	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

Table 8-1: The logical operators defined in the action language. Note that certain operators are valid for certain data types only

`and`	logical and	boolean
`or`	inclusive logical or	boolean
`not`	logical negation	boolean

Table 8-1: The logical operators defined in the action language. Note that certain operators are valid for certain data types only

8.1.5 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. `Selected` 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.

Notes

- 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;
```

8.1.6 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>;

where:

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

Examples

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

8.1.7 Assignment of Variables

Calculations analogous to Shlaer-Mellor transformation processes 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>;

where:

<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 an object instance.

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

<string var> is a string variable or a string attribute of an object 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.
- `Assign` is an optional keyword in the assignment statement.

Examples

```
assign x = 1;
```

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

8.1.8 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 OOA models require minimal use of constants since such data is more commonly stored as attributes of specification objects.

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

Table 8-2: Syntax of Constant Data

Notes

- Constants may be defined for the above data types only.
- A constant may be used in any construct requiring an expression.

8.1.9 Additional Unary Operators

Three set operators have been provided to allow the analyst to determine the size of a 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>

where:

<handle> is an <instance handle> or <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;
```

9. TRANSFORMERS/BRIDGES

This chapter explains the use of transformer and bridge processes.

9.1 TRANSFORMERS/BRIDGES

This section describes analyst-defined transformers and bridges.

9.1.1 Transformer Invocation

The analyst may define data transformers as desired using the Transformer Data Editor of BridgePoint Model Builder. A transformer invocation can be used as a stand-alone statement or it can be used in an expression.

Syntax

as a transformer expression:

```
[transform] <keyletter>::<transformer name> (<data item>:<expression>, . . .)
```

as a stand-alone transformer assignment statement:

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

where:

<keyletter> is the keyletter of an object.

<transformer name> is a transformer defined for the object specified by <keyletter>.

<data item> is the name of a data item defined as input for <transformer 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 an attribute or a local variable.

Notes

- `Transform` is an optional keyword in all transformer invocations.
- A transformer expression may be used anywhere an expression is valid (e.g., assignment statement read values, control logic expressions, etc.).
- The stand-alone transformer assignment statement is provided for compatibility with action language written for previous versions of BridgePoint. This form of transformer invocation must always be a stand-alone statement and cannot be used as an expression within another statement.
- Since a transformer expression may be used anywhere an expression may be used, stand-alone transformer assignment statements are not necessary. The user may simply use a transformer 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 <transformer name>.

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

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

Example

```
// transformer expression without assigning the return value
DD::open(wait:20);

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

// transformer expression within a while loop
while (MOD::status() != 1)
   // ...
end while;

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

9.1.2 Bridge Invocation

Syntax

as a bridge expression:

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

as a stand-alone bridge assignment statement:

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

where:

<eekeyletter> is the keyletter(s) 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 an attribute or a local variable.

Notes

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

- `Bridge` is an optional keyword in the bridge statement.

- 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 is provided for compatibility with action language written for previous versions of BridgePoint. This form of bridge invocation 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 an object or a local variable that has previously been assigned), then either

  - <variable> must be of the same data type as the output value of <bridge name>, or
  - the output value of <bridge name> 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 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 );
```

9.1.3 Avoiding Ambiguity in Invocations

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

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

In such a situation, all invocations of the ambiguous operations should be clarified by using the appropriate keyword (`bridge` or `transform`).

9.2 ACTIVE DESCRIPTIONS

Action language can be stored in transformer and bridge operation descriptions. The syntax rules applied to action language stored in these descriptions differ slightly from those applied to action language stored in state-actions (state descriptions). These differences are described in detail below.

9.2.1 Return Statement

Since transformers and bridge operations can return a value, the `return` statement is accepted within transformer and bridge operation descriptions.

Syntax

return <expression>;

return;

where:

<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 transformer or bridge operation.

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 transformer or bridge operation is `void`, then <expression> must be omitted.

Example

```
// ACTION_SPECIFICATION: TRUE
select any dog from instances of DOG;
return dog.weight;

// ACTION_SPECIFICATION: TRUE
// 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";
```

9.2.2 Parameters

Transformers and bridge operations can accept parameters. These parameters can be accessed as <read values> by using the `param` keyword.

Syntax

param.<parameter>

where:

<parameter> is the name of a parameter for the transformer or bridge operation.

Notes

- `param`.<parameter> is an <expression> and so can be used in any action language construct specifying an expression.

Example

```
// For a bridge or transformer invocation like MATH::SQR(x:3)

// ACTION_SPECIFICATION: TRUE
// Return x**2
return param.x * param.x;
```

9.2.3 Other Differences

The action language within a description may not contain references to `rcvd_evt` or `self`.

Notes

- Transformers and bridge operations do not receive events, so `rcvd_evt` is not allowed within a description.

- Transformers and bridge operations are not associated with instances, so `self` is not allowed within a description.

10. DATE AND TIME

This chapter illustrates the built-in support for accessing date and time values.

10.1 EXTERNAL AND INTERNAL TIME

This section describes statements that support date and time. The action language 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.*

10.1.1 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>

[bridge] <integer variable> = TIM::get_day (date:<date variable>);

[bridge] <integer variable> = TIM::get_month (date:<date variable>);

[bridge] <integer variable> = TIM::get_year (date:<date variable>);

[bridge] <integer variable> = TIM::get_second (date:<date variable>);

[bridge] <integer variable> = TIM::get_minute (date:<date variable>);

[bridge] <integer variable> = TIM::get_hour (date:<date variable>);

where:

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

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

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

Notes

- External time is represented by a 24-hour clock.

10.1.2 Internal Time

Syntax

To read the internal system clock, write

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

where:

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

Notes

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

11. Timers

This chapter explains the syntax for starting, stopping and manipulating OOA timers.

11.1 TIMERS

This section describes statements that support timers. The action language supports the concept of OOA timers that are manipulated through the use of bridge operations associated with the TIM external entity.

11.1.1 Starting a Timer

Syntax

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

Starts a timer set to expire in <arithmetic expression> microseconds, generating the event <event instance> upon expiration. Returns the instance handle of the timer.

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

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.

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);
```

11.1.2 Querying a Timer

Syntax

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

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

11.1.3 Manipulating a Timer

Syntax

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

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>);
```

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.

11.1.4 Canceling a Timer

Syntax

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

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.

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.

APPENDIX A: REFERENCES

This appendix lists the various published works used in compiling this manual.

A.1 REFERENCES

Shl92Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*, Prentice Hall, Englewood Cliffs, 1992

Shl95Sally Shlaer and Neil Lang, *Shlaer-Mellor Method: The OOA96 Report*, Project Technology, Inc., Berkeley, California, 1995

Wil95Ian Wilkie, Adrian King, and Mike Clarke, *The Action Specification Language (ASL) Reference Guide*, version 2.4, Kennedy-Carter, London, 1995

*A*PPENDIX *B*: *K*EYWORDS

This appendix contains a list of the reserved words.

B.1 KEYWORDS

across	and	any	assign
assigner	**break**	bridge	by
cardinality	**continue**	**control**	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	**stop**
to	transform	true	unrelate
using	**where**	**while**	

Table B-1: BRIDGEPOINT Action Language Keywords

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.

Keywords shown in **bold** type are new as of version 3.3 of BridgePoint.

APPENDIX C: SYNTAX SUMMARY

This appendix contains a summary of the syntax of the language.

C.1 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 <handle set>
    <statements>
end for;

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

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 <handle set> from instances of <keyletter> [ where <where expression> ];
```

Writing Attributes

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

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> -> <instance chain> [where <where expression>];

select any <instance handle> related by <start> -> <instance chain> [where <where expression>];

select many <handle set> related by <start> -> <instance chain> [where <where expression>];

Reading Event Data

rcvd_evt.<supplemental data item>

Generating Events

generate <event label>[:<event meaning>] [(<event parameters>)] to <target>;

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

generate <event instance>;

Arithmetic, Logical, and String Assignment

[assign] <boolean var> = <boolean expression>;

[assign] <arithmetic var> = <arithmetic expression>;

[assign] <string var> = <string expression>;

Size of a Set

empty <handle>

not_empty <handle>

cardinality <handle>

Transformers

[transform] <keyletter>::<transformer name> (<data item>:<expression>, . . .)

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

Bridges

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

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

Action Language in Descriptions

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

Date and Time

See “External and Internal Time” on page 61.

Timers

See “Timers” on page 65.

C.2 STATEMENT COMPONENTS

<arithmetic expression>	expression evaluating to a real or an integer value
<arithmetic operator>	`+`, `-`, `*`, `/`, or `%` (remainder from arithmetic division)
<attribute>	name of an attribute
<binary operator>	and, or, `+`, `-`, `*`, `/`, or `%`
<boolean expression>	expression evaluating to `TRUE` or `FALSE`
<bridge name>	is the name of a bridge defined for the external entity specified by <eekeyletter>.
<eekeyletter>	keyletter(s) of an external entity
<event instance>	local variable of type event instance
<event label>	<keyletter><event number>
<event meaning>	is 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>	is the supplemental data items (if any) to be carried by the event. Each data item is given in the form <supplemental data item>: <value>.
<transformer name>	is the name of a transformer defined for the object specified by <keyletter>.
<handle set>	local variable referring to a set of instance handles
<handle>	is an <instance handle>, <handle set>, or a timer handle (or a bridge or transformer invocation that returns a timer handle).
<instance chain>	<relationship link> or <relationship link> -> . . . -> <relationship link>
<instance handle>	local variable referring to a single instance
<keyletter>	key letters of an OOA object
<read value>	is a readable value: a constant, local variable, <instance handle>.<attribute>, `rcvd_evt`.<supplemental data item>, `param`.<parameter>, or an invocation of a transformer or bridge operation.
<relationship link>	is a <keyletter>[<relationship specification>], where the square brackets are literal and do not indicate optional text.
<relationship phrase>	text description of the relationship enclosed in tick marks
<relationship specification>	R<number> or R<number>.<relationship phrase>

<simple string>	is any of the following that evaluates to a string value: string constant, local variable, attribute of an object, transformer invocation, bridge invocation, or a supplemental data item received from an event.
<string expression>	expression evaluating to a string value
<supplemental data item>	name of a supplemental data item
<unary operator>	is a unary operator such as not for boolean expressions and + and - for arithmetic expressions.
<unique id>	is a variable of type unique ID (or a bridge or transformer invocation that returns a unique ID).
<where expression>	is a special boolean expression at the end of a select statement; it must contain the selected keyword.

C.3 PRODUCTION RULES

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

statement ::= (assignment_statement | control_statement | break_statement | bridge_statement | continue_statement | transform_statement | bridge_or_transform_statement | create_event_statement | create_object_statement | delete_statement | for_statement | generate_statement | if_statement | relate_statement | unrelate_statement | select_statement | while_statement | return_statement | empty_statement) SEMI ;

assignment_statement ::= [ASSIGN] assignment_expr ;

break_statement ::= BREAKTOKEN ;

bridge_statement ::= BRIDGE [(local_variable | attribute_access) EQUAL] bridge_invocation ;

bridge_or_transform_statement ::= bridge_or_transform_invocation ;

control_statement ::= CONTROL STOP ;

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 (statement)* (END FOR | Eof) ;

generate_statement ::= GENERATE (event_spec | local_variable) ;

if_statement ::= IF expr (statement)* [(ELIF expr (statement)*)+] [ELSE (statement)*] (END IF | Eof) ;

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 [(local_variable | attribute_access) EQUAL] transform_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 (statement)* (END WHILE | Eof) ;

assignment_expr ::= (local_variable EQUAL)? local_variable EQUAL expr | (attribute_access EQUAL)? attribute_access EQUAL expr | event_data_access EQUAL expr ;

attribute_access ::= inst_ref_var DOT attribute ;

bridge_invocation ::= ee_keyletters DOUBLECOLON bridge_function LPAREN [bridge_or_transform_parameters] RPAREN ;

bridge_or_transform_invocation ::= obj_or_ee_keyletters DOUBLECOLON function_name LPAREN [bridge_or_transform_parameters] RPAREN ;

bridge_or_transform_expr ::= BRIDGE bridge_invocation | TRANSFORM transform_invocation | bridge_or_transform_invocation ;

bridge_or_transform_parameters ::= bridge_or_transform_data_item COLON expr (COMMA bridge_or_transform_data_item COLON expr)* ;

event_data_access ::= RCVD_EVT DOT supp_data_item ;

event_spec ::= event_label [COLON event_meaning] [LPAREN [supp_data] RPAREN] TO (((object_keyletters ASSIGNER)? object_keyletters ASSIGNER | (object_keyletters CREATOR)? object_keyletters CREATOR) | (inst_ref_var_or_ee_keyletters)) ;

inst_ref_var_or_ee_keyletters ::= (local_variable | GENERAL_NAME | kw_as_id3) ;

instance_chain ::= local_variable (ARROW object_keyletters LSQBR relationship [DOT phrase] RSQBR)+ ;

object_spec ::= (RELATED BY instance_chain | FROM INSTANCES OF object_keyletters) [WHERE expr] ;

param_data_access ::= PARAM DOT bridge_or_transform_data_item ;

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

transform_invocation ::= object_keyletters DOUBLECOLON transformer_function LPAREN [bridge_or_transform_parameters] RPAREN ;

where_spec ::= expr ;

assoc_obj_inst_ref_var ::= inst_ref_var ;

attribute ::= general_name ;

bridge_function ::= function_name ;

bridge_or_transform_data_item ::= data_item_name ;

data_item_name ::= general_name ;

keyletters ::= general_name ;

ee_keyletters ::= keyletters ;

event_label ::= general_name ;

event_meaning ::= (phrase | general_name) ;

general_name ::= (limited_name | GENERAL_NAME | kw_as_id2 | kw_as_id4) ;

limited_name ::= ID | RELID ;

inst_ref_set_var ::= local_variable ;

inst_ref_var ::= local_variable ;

kw_as_id1 ::= ACROSS .. USING;

kw_as_id2 ::= ACROSS .. TRUETOKEN;

kw_as_id3 ::= BRIDGE .. TRUETOKEN;

kw_as_id4 ::= PARAM .. SELF;

local_variable ::= (limited_name | kw_as_id1 | SELECTED | SELF | GARBAGE) ;

function_name ::= general_name ;

obj_or_ee_keyletters ::= keyletters ;

object_keyletters ::= keyletters ;

phrase ::= (PHRASE | BADPHRASE_NL | Eof) ;

relationship ::= RELID ;

supp_data_item ::= data_item_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 | MINUS] term ;

boolean_negation ::= NOT term ;

term ::= (CARDINALITY | EMPTY | NOTEMPTY) local_variable | rval | LPAREN ((assignment_expr)? assignment_expr | expr) RPAREN ;

rval ::= constant_value | variable | attribute_access | event_data_access | bridge_or_transform_expr | param_data_access | QMARK ;

variable ::= local_variable ;

constant_value ::= (FRACTION | NUMBER | TRUETOKEN | FALSETOKEN) | quoted_string ;

quoted_string ::= QUOTE (STRING | BADSTRING_NL | Eof) ;

APPENDIX D: CUSTOMER SUPPORT

This appendix contains information on obtaining customer support.

D.1 SUPPORT CHANNELS

BridgePoint Customer Support is available to all BridgePoint users that have current maintenance and support. Support can be obtained in several ways:

1. By WWW 24 hours a day at http://www.projtech.com/support/bpcsa.html
2. By e-mail 24 hours a day at support@projtech.com.
3. By facsimile 24 hours a day at (520) 544-2912.
4. By telephone 9:00 AM to 5:00 PM Mountain Standard Time at (800) 482-3853 or (520) 544-0808.

Your Web Support Access

Your project team has a special user-id and password to allow your entire team access to web-based customer support. We encourage you to record your ID and password below for easy reference:

User ID ________________________ Password ______________________

Web Support Features

We encourage all BRIDGEPOINT support customers to visit our web support site. We are working to grow the amount of self-help support features available on this site, so you can help yourself through problems 24 hours a day, 7 days a week. Some of the features already on the site include:

Problem Report Submission - Use a form to submit a problem report and be assured you are providing enough information for our support technicians to help you.

Enhancement Request - Submit your enhancement ideas directly from your workstation as you think of them. This information will be immediately entered in our enhancements database for consideration in future releases. Your ideas drive our product development ONLY when we hear them.

Application Notes - Read technical papers that treat topics of interest in depth. Many are written by PT developers, instructors, and consultants, and we encourage you to submit your own contributions.

BridgePoint Users Mailing List - We have extended our popular users' mailing list service by adding a list for BridgePoint users. This mailing list allows you to communicate with a community of BridgePoint users, who are solving many of the same problems you encounter each day. You'll have to sign-up for this service.

BridgePoint Documentation - Download, and print locally, additional copies of BridgePoint manuals, such as the BridgePoint *Action Language Manual*.

Future Release Schedule - See the feature list for the next release of BridgePoint and other related products.

INDEX