

---

---

---

---

---

---

K E N N E D Y C A R T E R



## UML ASL Reference Guide

### ASL Language Level 2.5

Manual Revision D

Ian Wilkie, Adrian King, Mike Clarke, Chas Weaver, Chris Raistrick and Paul Francis

The information in this document is the property of and copyright Kennedy Carter Limited. Permission is granted to copy and distribute this document as long as the content of the document is not altered in any way, this and other copyright notices are retained in the copy and no charge is made for the copy (other than to cover reasonable duplication and distribution costs).

Copyright Kennedy Carter © 2003

---

---

---

---

---

---



# UML ASL Reference Guide

<b>1 Introduction to ASL</b>	<b>page 1</b>
1.1 Manual Revision D . . . . .	1
1.2 Feedback . . . . .	1
1.3 Acknowledgements . . . . .	2
1.4 Footnotes in the ASL Reference Guide . . . . .	2
<b>2 Update History</b>	<b>page 3</b>
<b>3 Basic Concepts</b>	<b>page 5</b>
<b>4 ASL Syntax</b>	<b>page 7</b>
4.1 Overall Structure. . . . .	7
4.2 Comments . . . . .	7
4.3 Names . . . . .	7
4.4 Naming Style . . . . .	8
<b>5 Data Items in ASL Segments</b>	<b>page 9</b>
<i>Data items used and produced within an ASL segment</i>	
5.1 Availability of Data Items . . . . .	9
5.2 Data Item Names . . . . .	10
5.3 Data Types . . . . .	10
5.4 Type Mixing Rules . . . . .	11
5.5 Data Item Multiplicity . . . . .	12
<b>6 Sequential Logic</b>	<b>page 15</b>
6.1 Switch Statement . . . . .	15
6.2 If Statement . . . . .	16
6.3 For Loop . . . . .	16
6.4 Loop Statement . . . . .	18
6.5 Nested Sequential Logic. . . . .	18

<b>7 Class and Object Manipulation</b>	<b>page 19</b>
7.1 Identity and Identifying Attributes . . . . .	19
7.2 Creation of Objects . . . . .	19
7.3 Writing Attributes of Objects . . . . .	21
7.4 Reading Attributes of Objects . . . . .	23
7.5 Deletion of Objects . . . . .	24
7.6 Obtaining Instance Handles . . . . .	25
7.7 Manipulating Single Objects and Sets of Objects . . . . .	26
7.8 Ordering of Instance Handle Sets . . . . .	28
<b>8 Association and Generalisation</b>	<b>page 31</b>
8.1 Association vs. Generalisation . . . . .	31
8.2 Referential Attributes . . . . .	32
8.3 Association Navigation . . . . .	33
8.4 Relationship Specifications . . . . .	35
8.5 Link Creation . . . . .	40
8.6 Link Deletion . . . . .	41
8.7 Generalisation Relationships Revisited . . . . .	42
8.8 Correlated Associative Navigation . . . . .	43
8.9 Multivalued Association Classes . . . . .	44
8.10 Associate and Unassociate . . . . .	45
<b>9 Signal Generation</b>	<b>page 47</b>
<b>10 Arithmetic and Logical Operations</b>	<b>page 49</b>
10.1 Constants and Limits . . . . .	49
10.2 Arithmetic Calculations . . . . .	51
10.3 Local Variable Assignment . . . . .	52
10.4 Logical Conditions . . . . .	53
<b>11 Operations</b>	<b>page 55</b>
11.1 ASL Operations in the Context of a Domain Model . . . . .	55
11.2 Defining and Calling an ASL Operation . . . . .	55
11.3 Domain Scoped Operations . . . . .	56
11.4 Class Scoped Operations . . . . .	58
11.5 Object Scoped Operations . . . . .	60
<b>12 Timer and Time Operations</b>	<b>page 65</b>
12.1 The xUML Timer . . . . .	65
12.2 Current Date and Time . . . . .	70

<b>13 Complex Datatypes and Sets</b>	<b>page 71</b>
13.1 Supported Structures . . . . .	71
13.2 Definition of Structures . . . . .	72
13.3 Instantiation of Structures . . . . .	73
13.4 Assembly of Sets of Structures . . . . .	74
13.5 Use of Loops to Perform Unpacking of Set Structures . . . . .	75
13.6 Ordering of Sets of Structures . . . . .	76
13.7 Subsets of Sets of Structures . . . . .	77
<b>14 Sets, Sequences and Bags</b>	<b>page 79</b>
14.1 Equality . . . . .	79
14.2 The Unique Operation . . . . .	80
14.3 The countof Operation . . . . .	80
14.4 Set Combination Operations . . . . .	80
<b>15 ASL for Bridge Operations</b>	<b>page 83</b>
15.1 Basic Concepts and Terminology . . . . .	83
15.2 Domain Scope Within a Bridge . . . . .	85
15.3 Type Mixing in Bridges . . . . .	85
15.4 Types of Bridge . . . . .	86
15.5 Definition and Invocation of Non-Object Scoped Bridges . . . . .	87
15.6 Definition and Invocation of Object Scoped Bridges . . . . .	89
15.7 Counterpart Relationship Manipulation . . . . .	91
15.8 Definition & Invocation for a Signal Bridge . . . . .	97
<b>16 Native Language Inserts</b>	<b>page 99</b>
<b>17 Appendix A: Requirements for an ASL</b>	<b>page 101</b>
<b>18 Appendix B: The Keywords of ASL</b>	<b>page 103</b>
<b>Index</b>	<b>page 105</b>



# 1 Introduction to ASL

ASL is an implementation independent language for specifying processing within the context of an Executable UML (xUML) model. The language has been placed in the public domain and may be freely used by modellers and developers.

The language is compatible with the emerging “Precise Action Semantics” extension to the UML standard (<http://www.umlactionsemantics.org>)

The aim of the language is to provide an unambiguous, concise and readable definition of the processing to be carried out by an object-oriented system. The requirements and decisions that went into the language design are outlined in Appendix A.

Since 1993 the ASL specification has been extensively distributed to software engineers working in organisations worldwide. The language has been actively used on a large number of projects ranging from small embedded controllers to large distributed database systems. These projects have used a variety of different techniques for mapping the ASL into the chosen software architecture and implementation language. The translation techniques used range from fully automatic generation to manual coding using a defined set of rules. Target languages have included c, c++, Objective c, Ada, Java, Fortran, SQL and proprietary languages.

The ASL definition is independent of any particular implementation. Potential users should therefore check carefully the level of support offered by any particular CASE tool or code generator, before embarking on large-scale use of the language.

## 1.1 Manual Revision D

This revision makes further clarification of terminology and document re-design.

## 1.2 Feedback


If you have any comments or questions regarding ASL or this manual please e-mail them to [method@kc.com](mailto:method@kc.com).

## 1.3 Acknowledgements

The authors would like to thank the many people who have provided constructive criticism and feedback over the years. In particular we would like to thank Colin Carter, Paul Francis, Chris Raistrick, John Wright, Jeff Terrell, David Hawes and Andy Land of Kennedy Carter, Chas Weaver of Rational Software Corporation, Steve Arnott, Phil Reynolds and Don Stewart of Marconi Communications, David Stone and Tim Wilson of Simoco, Dick Taylor and Craig Anderson at MG Rover, Terry Ruthruff and Bary Hogan of Lockheed Martin Tactical Aircraft Systems, Eric Mintz of Bear Stearns, Tony Bloomfield of BAE Systems, and Bryan Hawkins of Domeon Software.

If we have omitted anyone from this list then please accept our apologies.

## 1.4 Footnotes in the ASL Reference Guide

Footnotes are used in the ASL Reference guide to clarify the use of ASL when used with Kennedy Carter's iUML tool. They are highlighted with the iUML logo .



## 2 Update History

Document Version	Date	Comments
1.0 Draft	16/09/93	Stopped with "Navigation Processes".
1.1 Draft	07/10/93	Changes to write accessor syntax - position of "where" clause changed to increase clarity. Addition of relationship operations, transforms signals. More detail on conditions and constants. Details of overall syntax.
1.2	03/11/93	Addition of "ordered by" and "sizeof".
1.3	24/11/93	Specific mention of signal generation with instance handles. Timers.
1.4	05/01/94	Use of association classes in relationship creation and navigation.
1.5	10/01/94	Reference to M-(M:M) associations. Minor Corrections. Extensions to the reserved word list. Document title changed.
2.0	15/02/94	Major Revision of document structure. Additional Relationship Navigation facilities. Revised syntax for one-of, size-of, only directives.
2.1	03/03/94	Minor Corrections.
2.2	04/05/94	Clarification of association class creation/deletion rules. Introduction of "associate" and "unassociate" constructs. Addition of "current_time" and "current_date" keywords. Minor Corrections.
2.3	19/05/94	Correction to procedure parameter type declarations. Inclusion of correlated associative navigation. Table of contents in reference manual.
2.4	11/01/95	Incorporation of complex data types and set manipulation. Withdrawal of sets of single data item. Fix to problem of assigning constant enumeration values. Significant changes to timer use. Withdrawal of "with" clause on "generate". Withdrawal of "single instance specification" idea for "find"
2.4 Rev A	01/05/96	Changes to document content and structure to improve usability.

<b>Document Version</b>	<b>Date</b>	<b>Comments</b>
2.5	06/12/96	Introduction of formalised definition and invocation of synchronous operations and support for counterpart instances in bridges.
2.5 Rev A	24/06/00	Clarification of meaning of equality for sets and structures. Explanation of polymorphic operations.
2.5 Rev B	20/02/01	Adjustment to terminology for UML compatibility and Precise Action Semantics submission.
2.5 Rev C	23/07/01	Further clarification of terminology and document re-design.
2.5 Rev D	16/12/02	Addition of section on Multi-valued Association Classes Addition of “Associate” and Unassociate” statement definitions. Other minor typographical fixes

## 3 Basic Concepts

ASL is a language providing:

- Sequential Logic
- Access to the data described by the Class Diagram
- Access to the data supplied by signals initiating actions
- The ability to generate signals
- Access to timers
- Access to synchronous operations provided by classes and objects
- Access to operations provided by other domains
- Tests and Transformations

Unlike conventional languages, there is no concept of a “main” function or routine where execution starts. Rather, ASL is executed in the context of a number of interacting state machines, all of which are considered to be executing concurrently and in the context of synchronous call stacks invoked from these state machines or directly from outside the system. Any state machine, on receipt of a signal (from another state machine or from outside the system) may respond by changing state. On entry to the new state, a block of processing is performed. This processing can, in principle<sup>1</sup> execute at the same time as processing associated with another state machine.

ASL is organised in to “Segments” each of which is a sequential set of ASL statements with a local data scope. An ASL segment corresponds to a “Procedure” in the proposed “Precise Action Semantics for the UML” currently being submitted to the OMG.

In xUML ASL segments can be used to define:

- The processing to be carried out on entry to a state
- The processing to be carried out by a method behind an operation
- The startup sequence for a system either for test or for target system purposes



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

- The processing to be carried out by test methods for simulation purposes
- The processing to be executed in bridges providing the mapping between domains

The execution rules for a segment are as follows:

- Execution commences at the first ASL statement in the segment and proceeds sequentially through the succeeding lines as directed by the logic structures.
- Execution of the segment terminates when the last ASL statement is completed.
- The only external data available to the ASL segment is:
  - Signal data supplied with the signal or operation call that caused execution of the ASL segment
  - Attributes of classes as defined in the Class Diagram
  - There are no “global” data other than those detailed in the Class Diagram
  - Local variables created within the segment go out of scope when execution of the segment ends

This section describes the syntax of ASL, providing explanation and examples of ASL usage.

## 4.1 Overall Structure

An ASL segment consists of a number of ASL statements. Each statement can be either a simple statement (such as an access to the attributes of a class) or a sequential logic structure (such as a loop).

ASL statements are terminated by the use of a new line. A “\” character can be placed at the end of the line to indicate that the following line is to be included as part of the same ASL statement.

## 4.2 Comments

Comments may be inserted by use of the “#” character at any point in the line. When this character is detected, the rest of the line is considered to be a comment and ignored.

Multi-line comments may be created by surrounding the lines by “#{” and “}#”. In this case there should be no characters on a line before the “#{” or after the “}#”. Multiline comments do not nest. Care should be taken not to inadvertently start a multi-line comment when commenting out ASL statements that start with a set definition “{ ...”.

## 4.3 Names

ASL statements are composed from:

- ASL Keywords
- Logical and Arithmetic Operators
- Names of xUML Elements (Data items, Classes, Associations, Generalisations, Signals and Enumeration Values).

ASL keywords may be represented in upper and lower case in any combination. In this document ASL keywords and symbols are shown in **courier bold** font.

The names of xUML elements must conform to the following rules:

- Names are case sensitive
- Names may contain only the characters [a-z][A-Z][0-9][\_]<sup>2</sup>
- Names must not start with an underscore character [\_]
- Names must not begin or end with a numeric character [0-9]
- Names must not conflict with the keywords of ASL<sup>3</sup>
- Class or terminator “Key Letters” must not contain any numeric character [0-9]

## 4.4 Naming Style

Since, in general, explicit type statements are not required in ASL (see the discussion of Data Types in Section 6), great care must be taken with the use of names of data items. Specifically, within the scope of one action or operation, the same name **must not be used** for different data items or data items of a different type. Names are statically typed by first appearance.

For example:

```
Dog = find Dog where name = "Fido"
```

is invalid, since “Dog” is used both as an instance handle (of class Dog) and as a class name.

Thus, local variables must not have the same name as:

- A class
- A terminator
- An attribute
- An association (e.g. R7)
- A Generalisation (e.g. R8)
- A counterpart association (e.g. CPR1)
- A signal parameter

Additionally, the following is very unlikely to be valid:

```
if my_dog.name = "Fido" then
    sex_or_colour = my_dog.colour
else
    sex_or_colour = my_dog.sex
endif
```

since `sex_or_colour` is probably of more than one data type (and also poor modelling!).



2. Specifically, spaces are not permitted. Since spaces are allowed in xUML model element names language implementations must provide some suitable mapping. In iUML all spaces are replaced by underscore characters when building a simulation or target code. ASL that references these names must therefore use the underscore character.

3. See Appendix B for a list of the keywords of ASL.

## 5 Data Items in ASL Segments

### Data items used and produced within an ASL segment

The ASL for a segment has access to, and can produce certain data items.

#### 5.1 Availability of Data Items

Data items available at the start of and during execution of a segment:

- Signal parameters of the signal that caused entry to the state (in the case of a segment attached to a state)
- Input parameters to the operation defined by this segment (in the case of a segment for an operation)
- The instance handle “**this**” (in an object scoped segment)
- Parameters returned by operation invocations made by this segment
- Values of attributes of classes in the Class Diagram
- Local variables (created by ASL statements within the segment)

Data items produced during action:

- Local variables
- Values of attributes of classes in the Class Diagram
- Signal parameters supplied to signal generation statements
- Operation parameters supplied to operation invocations

Data items have three properties:

- Name
- Data type
- Multiplicity

Taking these in turn:

## 5.2 Data Item Names

These must conform to the rules of ASL names defined in a previous section. Data item names must not conflict with the name of any class or terminator.<sup>4</sup>

## 5.3 Data Types

In order to support ASL as a rigorous process modelling formalism, the data types of class attributes in the xUML model must be specified rigorously.

Note that this is not a question of “design” polluting the analysis model. Rather that the choice of “Integer”, for example, as the data type of “Shipment ID” is simply reflecting the fact that in the problem domain, the “Shipment ID” can be 1, 2, 3... etc. How such types are implemented is a software design issue.

In ASL every data item has a type that is statically determined by first appearance. ASL allows the following “base types” for attributes of classes:

```
Real      Integer      Text
Date      Time_of_Day
Boolean
```

In addition analysts can define constrained versions of these (“User Defined Types”).

Enumeration types are permitted; they will always be user-defined and must always be provided with user-specified valid values. For example `traffic_light_colour_type` having the values `'Red'`, `'Amber'`, `'Green'`.

In addition to these types, local variables, signal parameters and operation parameters can be of one of the following types:

- **instance handle**  
This is a reference to an object

and:

- **sets of structures**  
Which is a complex data type covered in more detail in a later section

Finally, any data item can also be of “deferred” type. This means that the ASL in this domain does not implement the type. Instead all operations on that type are passed through to another domain<sup>5</sup>.

Notes:

- Enumerations, Instance handles and Sets of Structures are not considered to have a “base type”. This is important when interpreting the type mixing rules (see below).
- Attributes of classes *cannot* be of type “instance handle” or “set of structures”.



4.iUML Simulator requires all names to be unique within a domain, with the exception that the same attribute name can be used in multiple classes. Note also that no item may have the same name as a domain key letter. (See the section on “Bridges” for a discussion of domain key letters).

5.Currently, iUML Simulator supports the implementation of deferred types in implementation domains only.



- Strictly speaking an item is not of type “instance handle”, but rather of type “instance handle for class <x>”, where <x> is the name of a class. Any attempt to use an instance handle for one class in the context of another is therefore regarded as a compile time error.
- In any non-creation action in any state machine, a special instance handle called “**this**” is always available. This is always set to the object that is executing the current action. The value of “**this**” cannot be changed by an action. “**this**” is not available in creation states, nor in some operations and bridges (see sections “14 Sets, Sequences and Bags” on page 79 and “15 ASL for Bridge Operations” on page 83 for more details).
- Typed instance handles can exist as sets or single values. A set of handles for a particular class is considered to be different to a single handle for that class. This has an impact that is considered in the section on multiplicities below.
- Structures cannot exist as single items but must always be sets.

## 5.4 Type Mixing Rules

1. The following rules define the permitted level of type mixing in ASL:
2. With few exceptions<sup>6</sup>, there are no explicit type declarations in ASL. All data items are implicitly typed by the value assigned to them on their first use within a segment.
3. Except as allowed for in other rules, a data item of a particular type must not be used in a context where a different type is expected. To do so is considered to be a compile time error. There is no explicit type casting available.
  - A context where a particular type is expected can be:
    - any use of a typed instance handle
    - actual operation call parameter
    - actual signal generation parameter
    - actual bridge call parameter
    - values supplied to “with” clause on “create”
    - values supplied to “where” clause on “find”
    - assignment to an attribute
    - assignment to a local data item the type of which has been previously determined
4. Rule 2 is relaxed in that a *base type* can always be used in a context where a constrained version of *that type* is expected<sup>7</sup>.
5. If, at run time, the constraint on a type is violated by use of Rule 4 then this is considered to be a run time error.<sup>8</sup>
6. Types may be mixed in an arithmetic expression, provided that *all* the base types involved are either **Integer** or **Real**.
7. The result of an arithmetic expression is always **Integer** base type unless



6. The exceptions to this are procedure and bridge definitions, assignment of constants to enumerated types and assembly of sets of structures.

7.iUML Simulator actually allows use of a User Defined Type in a context where another type is expected as long as the base types agree. This will be tightened up in future releases. Note that the Configurable Code Generator (CCG) has much stricter type checking than iUML Simulator.

8.iUML Simulator does not perform run-time constraint checking.

any of the operands are **Real** base type, division is involved or raising to a non-integer or negative power is involved. In this case the result will be a **Real** base type<sup>9</sup>.

8. Constants are considered to be of the corresponding base type.
9. Type mixing of operands is not permitted in a binary logical expression.
10. Any data items in scope when a \$USE directive is encountered automatically revert to base type. This allows implicit casting in bridges by virtue of rule 4. See “15 ASL for Bridge Operations” on page 83.
11. Types that are not considered to have a base type (enumerations, typed instance handles and sets of structures) cannot ever be used in a context where another type is expected.<sup>10</sup>

## 5.5 Data Item Multiplicity

Data items can be single valued, or can be sets. A set is simply a collection of values and is analogous to an array, vector or tuple in other languages.

In ASL, only instance handles and structures can be the basis of sets. Structures are described in “Complex Datatypes and Sets” on page 71. Note that a set of instance handles for class <x> is of a different data type to a set of structures composed of a single member of type instance handle for class <x>.

Single valued data items are written simply using their name, for example an assignment to a local variable is written:

```
<local variable> = <ASL statement that returns a single value>
```

Set valued items must be explicitly noted by using the brace characters “{}”, thus:

```
{<local variable>} = <ASL statement that returns a set>
```

In ASL, sets cannot be used in a context where a single value is expected and vice-versa. Throughout this manual, it will be made clear whether a statement returns a set or a single value.

Various functions may be used to manipulate sets:

```
<local variable> = find-one {<set of instance handles>}11
```

Chooses an arbitrary object from the set and assigns it to the single valued local variable.

```
<local variable> = countof {<set of instance handles>}
```

Finds the number of elements in the set and assigns it to the local variable

```
<local variable> = find-only {<set of instance handles>}12
```

At run time, if the {<set of instance handles>} contains one and only one value, then this will be assigned to the single valued local variable. If more than one value exists in the set, a run time error will have occurred.




9.iUML Simulator does not support mixing of Integer and Real types in an arithmetic expression.

10.Currently, iUML Simulator allows assignment of enumerations of different types. This will work correctly if the enumerations have been defined in the correct way to support the required mapping.

11.“one-of” is equivalent to “find-one” and is due to be removed from subsequent language level versions.

12.“only” is equivalent to “find-only” and is due to be removed from subsequent language level versions.

Note that at run time:

- A single valued data item may contain the value “**UNDEFINED**”. This would happen, for example, when accessing an attribute whose value has not yet been set<sup>13</sup> .
- A set may be empty. The “countof” function will return the value 0. Any attempt to access a member of the set (e.g. using “find-one”) will result in the value UNDEFINED.



13. Whether such a value is actually implemented and testable at run time is architecture dependent. iUML Simulator supports UNDEFINED only for instance handles.



## 6 Sequential Logic

### 6.1 Switch Statement

Depending on the outcome of a test, at most one of a number of groups of ASL statements (referenced by a switch clause) will be executed.

Syntax:

```
switch <switch variable>
  case <value 1>
    # Executed if <switch variable> = <value 1>
    <ASL statements>
  case <value 2>
    # Executed if <switch variable> = <value 2>
    <ASL statements>
  ...
  default
    # Executed if no other clause has executed
    <ASL statements>
endswitch
```

Where:

<switch variable> is a data item that can be:

single valued local variable	of a type for which “=” is defined <sup>a</sup>
single valued signal parameter	
single valued <instance handle>.<attribute>	(See 10.4 “Logical Conditions” on page 53)
single valued operation input parameter	

a.iUML Simulator supports only the use of integer or enumeration as a loop local variable.

- Notes:
1. The switch execution terminates at the end of the executed case clause. (i.e. Execution does not continue through to the next clause, as is the case in “C”).
  2. There must be at least one **case** clause.
  3. The single **default** clause is optional. (Note that if omitted, there may be no clause that is executed at run time.)

## 6.2 If Statement

Depending on the result of a test, one of up to two groups of statements will be executed.

Syntax:

```
if <condition> then
    # Executed if <condition> is TRUE
    <ASL statements>
endif
```

or:

```
if <condition> then
    # Executed if <condition> is TRUE
    <ASL statements>
else
    # Executed if <condition> is FALSE
    <ASL statements>
endif
```

Where <condition> is a Boolean value or logical condition<sup>14</sup>, as described in “Logical Conditions” on page 53.

## 6.3 For Loop

There are two forms of **for** loop defined in ASL. The first applies to sets of instance handles where one or more ASL statements will be executed a number of times, with a local variable being assigned to each value in the instance handle set in turn. The second applies to sets of structures, and is used to unpack such sets. The former is explained below, the latter is explained in “Sets, Sequences and Bags” on page 79.

Syntax:

```
for <loop local variable> in {<set of instance handles>} do
    <ASL statements using loop local variable>
endfor
```

or:

```
for <loop local variable> in {<set of instance handles>} do
    <ASL statements using loop local variable>
    <break statement>
endfor
```

Where:

<pre>&lt;break statement&gt; is <b>breakif</b> &lt;condition&gt;<sup>a</sup> or: <b>break</b></pre>
---

a. iUML Simulator does not support the comparison of instance handles in a “breakif” statement.



14. iUML Simulator does not support the comparison of instance handles in an “if” statement.

- Notes:
1. Each iteration round the loop will assign <loop local variable> to a successive value in the <set of instance handles> until all the elements have been used.
  2. The order of obtaining the members of the set will be indeterminate unless the set has been constructed with explicit ordering.
  3. The <break statement> will cause termination of the loop as follows:
  4. The “**breakif**” statement will cause termination of the loop if <condition> is true
  5. The “**break**” statement will always cause termination of the loop
  6. Following the execution of a break, execution will continue with the statement immediately following the “**endfor**”.
  7. The <break statement> is optional
  8. The <break statement> may appear anywhere in the loop (for brevity, this possibility has been omitted from the syntax above).
  9. The <break statement> may appear multiple times in the loop.
  10. The value of <loop local variable> will remain valid after termination of the loop (either by execution of a <break> or by exhaustion of the {<set of instance handles>}).
  11. No <asl statement> within the loop may change the value of the <loop local variable>.
  12. No <asl statement> within the loop may change the contents of the {<set of instance handles>} that is the subject of the loop.
  13. If {<set of instance handles>} is empty, then the ASL within the loop will not be executed, and the value of <loop local variable> will be **UNDEFINED**.
  14. The type of the members of {<set of instance handles>} determines the type of <loop local variable>. Thus, by virtue of the typing rules defined earlier, <loop local variable> may not be used later in a context expecting or defining a different type (for example, in a subsequent loop with a different type).

## 6.4 Loop Statement

A general purpose loop that will repeatedly execute a group of ASL statements until a break is explicitly executed.

Syntax:

```
loop
  <ASL statements>
  <break statement>
endloop
```

Where:

<pre>&lt;break statement&gt; is <b>breakif</b> &lt;condition&gt;<sup>a</sup> or: <b>break</b></pre>
---

a. iUML Simulator does not support the comparison of instance handles in a "breakif" statement.

- Notes:
1. This will loop indefinitely, until <break statement> causes a break, at which point execution will continue at the statement immediately following the "endloop" statement.
  2. The <break statement> will cause termination of the loop as follows:
  3. The "breakif" statement will cause termination of the loop if <condition> is true
  4. The "break" statement will always cause termination of the loop
  5. The <break statement> is mandatory
  6. The <break statement> may appear anywhere in the loop.
  7. The <break statement> may appear multiple times in the loop.
  8. To conform to the xUML rules for segment execution, the analyst must guarantee that a break is executed at some finite time after the start of loop execution.

## 6.5 Nested Sequential Logic

Sequential logic may be nested to any depth. For example:

```
for <local variable> in {<set of instance handles>} do
  if <condition 1> then
    <ASL statements>
    loop
      <ASL statements>
      breakif <condition 2>
    endloop
    <ASL statements following break>
  else
    <ASL statements>
  endif
endfor
```

- Notes:
1. If <condition 2> is true, the "breakif" statement will cause execution to continue at the line immediately following the *inner* endloop statement, i.e. "<ASL statements following break>".



## 7 Class and Object Manipulation

### 7.1 Identity and Identifying Attributes

In executable UML there is a very strong emphasis on modelling the identity of concepts in the problem domain. This identity is captured in the form of (possibly multiple alternate) identifiers each composed of one or more identifying attributes. ASL uses this concept of identity in several ways as a cross-check or constraint check. For example, if a Class has had an identifier defined then the values for it must be specified when an object of the class is created.

It is recognised, however, that in many problem domains identity is often arbitrary and there is less benefit in explicitly modelling it. ASL supports this in two ways. First the “create unique” operation will automatically assign values to explicitly modelled arbitrary identifying attributes and secondly, if there are no identifying attributes modelled then the issues of identity can be ignored by the analyst when writing ASL.

### 7.2 Creation of Objects<sup>15</sup>

Creation of an object is achieved by use of the “**create**” statement. Execution of this statement will cause the xUML architecture to create the object, which will then be visible to other ASL statements such as “**find**”.

Syntax:

```
<instance handle> = create <class> with <attribute assignments>
```

or:

```
<instance handle> = create unique <class>
```

or:

```
<instance handle> = create unique <class> with <attribute assignments>
```



15. iUML Simulator requires that the name of the state attribute of an active class is always “Current\_State” (which is the default name provided by the core iUML tool). All ASL which accesses this attribute must use this name.



## 7.3 Writing Attributes of Objects

Attributes of objects may be set to specific values by use of the write accessor construct.

Syntax:

```
<class>.<attribute list> = <value list> where <class condition>
```

or:

```
<instance handle>.<attribute list> = <value list>
```

or:

```
{<instance handle set>}.<attribute list> = <value list>
```

This takes the values in the <value list> and assigns them to the attributes in the <attribute list> for the objects specified by <instance handle> or by <class> **where** <class condition>. Where there is {<instance handle set>} then assignment is made for every member of the set.<sup>22</sup>

Where:

<class>	is a logical condition based on attributes of <class>
<class conditions>	is a logical condition based on attributes of <class>
<attribute list>	is <name of attribute> or [<name of attribute 1>,<name of attribute 2>...]
<value list>	is <value> or [<value 1>,<value 2>,...]
<value>	is <data item> available to the segment

- Notes:**
1. If the attribute list contains more than one value (the “[...],...” form) then matching with the value list will be on the basis of position within the list. If the two lists have a different number of elements, or there is a type mismatch then this will be regarded as a compile time error.
  2. It is explicitly disallowed to make the <value list> a read accessor; for example, the following is invalid:  

```
new_dog.owner_name = old_dog.owner_name
```
  3. Changing the value of any attribute that is part of any identifier of the class is not allowed since this is equivalent to deletion of one object and creation of another. An attempt to change such an attribute value is a compile time error.<sup>23</sup>
  4. Setting the value of the status attribute of an active class should be avoided except when executing a “**create**” operation.



22. Assignment to members of a set of handles is not supported in iUML Simulator.

23. This is not checked by iUML Simulator.

Example1:

```
# "Track" is a class with attributes "track_id", "position"  
# and "threat"  
# (Assume new_id, threat_level and detected_position are signal data  
# or local variables)  
# First, create and new object ...  
new_track = create Track with track_id = new_id  
  
# Now set an attribute using the returned instance handle  
new_track.position = detected_position  
  
# Now set an attribute using class specification  
Track.position = detected_position where track_id = new_id
```

Example 2:

```
# Example with multiple attributes  
  
Track.[position,threat] = [detected_position, threat_level] where track_id = new_id  
# The above example is equivalent to the following two statements:  
  
Track.position = detected_position where track_id = new_id  
Track.threat = threat_level where track_id = new_id
```

## 7.4 Reading Attributes of Objects

Attribute values for objects of a class may be read and assigned to local variables as follows:

Syntax:

```
<local variable list> = <class>.<attribute list> where <class condition>
```

or:

```
<local variable list> = <instance handle>.<attribute list>
```

Where:

<class>	is the name of a class
<class conditions>	is a logical condition based on attributes of <class>
<attribute list>	is <name of attribute> or [<name of attribute 1>,<name of attribute 2> ...]
<local variable list>	is <local variable> or [<local variable 1>,<local variable 2> ...] <sup>a</sup>

a. iUML Simulator does not support the [...] form.

- Notes:**
1. If class specification (<class> ... **where** <condition>) returns a set, then this is regarded as a run time error. If sets of data are required while reading attribute values then this must be explicitly managed using sets of structures (see later).
  2. If the attribute list contains more than one value (the “[...,...]” form) then matching with the local variable list will be on the basis of position within the list. If the two lists have a different number of elements then this will be regarded as a compile time error.

**Examples:**

```
# Read a single attribute value (NB "John" must be unique).
johns_age = Person.age where name = "John"

# Read two attributes at a time
[johns_age,johns_height] = Person.[age,Height] where name = "John"

# Example using an instance handle
# Create new object and get handle ...
new_person = create Person with name = new_name

# Later ... Use the instance handle to access an attribute
new_age = new_person.age
```

## 7.5 Deletion of Objects

Objects may be deleted by means of the following:

Syntax:

```
delete <instance handle>
```

or:

```
delete {<set of instance handles>}
```

or:

```
delete <class> where <class condition>
```

Will delete the object or objects specified by the <instance handle>, the {<set of instance handles>} or by the class specification (“<class> **where** <class condition>”).

- Notes:**
1. When an object is deleted, it is no longer available to the domain where the class is defined.
  2. If the instance handle set is used or the class specification specifies more than one object, then all the objects specified will be deleted.
  3. Deletion of an object is not sufficient to specify deletion of attached relationships. Certain types of architectures may fail if such “dangling” relationships are used at run time. The analyst must explicitly delete relationships before deleting the participating objects, otherwise this is considered to be a run time error ([See 8 “Association and Generalisation” on page 31](#)).
  4. Deletion of “this” is only allowed in ASL segments where the scope is object-based - i.e. in object-based operations and in state entry actions (in fact strictly only for those states declared in the xUML model to be a terminal state). An attempt to do otherwise is regarded as a compile time error.

**Examples:**

```
# Delete a single fully specified instance of "Dog"
# (Dog.name is the entire identifier of "Dog")
delete Dog where name = "fido"

# This line will cause deletion of every object
# referenced in the set {expiredLicences}
delete {expiredLicences}
```

## 7.6 Obtaining Instance Handles

Instance handles can be explicitly obtained by using the find statement:

Syntax:

```
{<instance handle set>} = find <class> where <class condition>
```

or:

```
{<instance handle set>} = find {<set of existing instance handles>} where <class condition>
```

or:

```
{<instance handle set>} = {<set of existing instance handles>} where <class condition>
```

All of the above will return a set of handles to instances of <class> whose attribute values match the conditions specified by <class condition>.

or:

```
{<instance handle set>} = find-all <class>
```

Returns a set of handles to all instances of <class>.

Where:

{<instance handle set>}	is a set of handles for those instances of <class> that match the criteria specified by <class condition>
<class>	is the name of the class
<class condition>	is a logical condition that specifies a set of instances of <class>
{<set of existing instance handles>}	is a set of instance handles to <class> that is used to form a sub-set of instance handles

- Notes:**
1. The <class condition> is specified as a logical condition using attributes of the class being found.
  2. The above forms of “**find**” always return a set of instance handles, even if there is only one member at run time.

**Examples:**

```
# Find employees who are close to retirement
{retiring_employee} = find Employee where age > 63

# Get all company cars
{car} = find-all Company_Car

# A more complex condition, used to get a set of handles
{retiring_male} = find Employee where age > 63 & sex = 'Male'

# Now use find again to further subset the handles
{local_retiring_male} = find {retiring_male} \
    where location = "London"
```

## 7.7 Manipulating Single Objects and Sets of Objects

The “**find**” statement always returns a set of instance handles. However, operations are provided to reduce this to a single value:

- **find-one**
- **find-only**

### Use of “find-one”

Syntax:

```
<instance handle> = find-one <class>
```

or:

```
<instance handle> = find-one <class> where <condition>
```

or:

```
<instance handle> = find-one {<set of instance handles>}
```

or:

```
<instance handle> = find-one {<set of instance handles>} where <condition>
```

Each of the above will return a single arbitrary object from the set specified by “<class>” or “<class> **where** <condition>” or “{<set of instance handles>}”

**Notes:** 1. If there are no objects in the set specified by “<class> **where** <condition>”, then <instance handle> will be set with the value **UNDEFINED**.

**Example:** # Get an arbitrary disk transfer with the correct status  
 jobToDo = **find-one** Disk\_Transfer **where** status = 'ReadyForRobot'



## Use of “find-only”

Syntax:

```
<instance handle> = find-only <class>
```

or:

```
<instance handle> = find-only <class> where <condition>
```

or:

```
<instance handle> = find-only {<set of instance handles>}
```

or:

```
<instance handle> = find-only {<set of instance handles>} where <condition>
```

Each of the above will return the only object in the set specified by “<class> **where** <condition>” or “{<set of instance handles>} **where** <condition>”. Omitting the **where** clause asserts either that there is only one object for the <class> or that there is only one object in the {<set of instance handles>}.

- Notes:**
1. If at run time there is not exactly one object in the set specified by the **find-only** statement then this is considered to be a run time error.
  2. This construct can be used for two purposes:
    - to indicate that the <condition> specifies the entire identifier of “<class>”
    - to allow the analyst to assert that the run time dynamics of the system are such that there will be exactly one object that satisfies the condition.

## Other Ways to Reduce a Set of Objects to a Single Object

Another way to reduce a set to a single object is to use the “**for**” loop - see the example below:

**Example:**

```
# Example of use of a 'for' loop to access single values from a set
{ localPersons } = Person where location = "Edinburgh"
for thePerson in {localPersons} do
  if thePerson.proximity = 'Local' then
    selectedPerson = thePerson
  break
endfor
if selectedPerson != UNDEFINED then
  <ASL statements relating to selectedPerson>
endif
```

## 7.8 Ordering of Instance Handle Sets

In the absence of any other specification, the “**find**” statement will return a set of instance handles with an arbitrary order. Use of the “**for**” loop on such unordered sets will access each object in an order that cannot be assumed or predicted by the analyst. If ordering is required, the following constructs can be used:

Syntax:

```
{<instance handle set>} = find <class> where <condition> ordered by <attribute 1> \
                                & <attribute 2> ...
```

OR:

```
{<instance handle set>} = find <class> where <condition> reverse ordered by <attribute 1> \
                                & <attribute 2> ...
```

OR:

```
{<instance handle set>} = find-all <class> ordered by <attribute 1> & <attribute 2> ...
```

OR:

```
{<instance handle set>} = find-all <class> reverse ordered by <attribute 1> & <attribute 2> ...
```

OR:

```
{<instance handle>} = find {<set of instance handles>} where <condition> ordered by <attribute 1> \
                                & <attribute 2> ...
```

OR:

```
{<instance handle>} = find {<set of instance handles>} where <condition> \
                                reverse ordered by <attribute 1> & <attribute 2> ...
```

OR:

```
{<instance handle>} = find {<set of instance handles>} ordered by <attribute 1> & <attribute 2> ...
```

OR:

```
{<instance handle>} = find {<set of instance handles>} reverse ordered by <attribute 1> \
                                & <attribute 2> ...
```

These will return sets of instance handles that are explicitly ordered by the referenced attribute(s). Any use of the “**for**” loop on such ordered sets will execute in the order specified when the set was created.

- Notes: 1. The meaning of the ordering will be defined for the standard attribute base types as follows:

Real	Normal Arithmetic Meaning	(e.g. 1.0 is less than 2.0)
Integer	Normal Arithmetic Meaning	(e.g. 1 is less than 2)
Date	Normal Time Ordered Meaning	(e.g. 2002.01.01 is less than 2002.01.02)
Time	Normal Time Ordered Meaning	(e.g. 11:00:00 is less than 13:57:00.)
Text	UNDEFINED	Although the meaning of these is undefined in the ASL <i>language</i> , we do not exclude the possibility that any particular implementation of the language may allow the provision of “user defined” comparisons in order to facilitate ordering on these attribute types.
Enum	UNDEFINED	
Boolean	UNDEFINED	

2. It is considered to be a compile time error to attempt to order by attributes of a type for which ordering is undefined.
3. In the case of “**ordered by**”, the ordering will be such that the first iteration of a “**for**” loop using the set of instance handles will return the object with the lowest value of <attribute> first.
4. In the case of “**reverse ordered by**”, the ordering will be such that the first iteration of a “**for**” loop using the set of instance handles will return the object with the highest value of <attribute> first. The “**reverse**” clause applies to all the attributes in the list and so a mixture of forward and reverse ordering cannot be achieved in one operation.
5. In the case where two objects have the same value of <attribute>, the order of objects will be indeterminate.
6. In the case where the find operation is working on an existing {<set of instance handles>}, then the **find** keyword is optional<sup>24</sup>.
7. Where ordering on multiple attributes is specified, the set will be sorted by <attribute 1> and then within each value of <attribute 1>, by <attribute 2> and so on<sup>25</sup>.



24.iUML Simulator does not allow the **find** to be omitted when both a **where** and (**reverse**) **ordered by** clause are present.  
 25.iUML Simulator supports ordering on a single attribute only.



In executable UML the Class Diagram is used to capture two different kinds of UML relationship:

- Association
- Generalisation

This section describes how ASL is used to manipulate these.

## 8.1 Association vs. Generalisation

Within the model, the ASL describes how and when the Association and Generalisation relationships are used to support the operation of the system. In particular, the ASL specifies how and when instances of the relationships are:

- Created
- Deleted
- Navigated (Read)

For these operations, ASL provides three relationship primitives:

- link
- unlink
- -> (navigate)

For associations these correspond to the CreateLinkAction, DestroyLinkAction and ReadAssociationAction of UML.

For generalisations, the mapping is slightly different. In UML, generalisation is not the same concept as inheritance although the two are often confused. UML defines generalisations simply as a taxonomic relationship between elements. By contrast inheritance is an implementation mechanism that operates on a generalisation hierarchy. In the inheritance view, a language creates instances of the most specialised class which then inherit attributes and operations from parent classes in the generalisation hierarchy.

Unfortunately, this approach has problems when trying to deal with two of the most common and powerful uses for generalisation in analysis modelling:

- Multiple Classification
- Dynamic Classification

As a result, ASL takes the approach of treating a populated superclass/subclass hierarchy as if there are separate objects which are related through the generalisation relationship. Models must create objects in both the superclass and subclass and relate (link) them explicitly through the generalisation. Thus the link and unlink operations map on to classification and reclassification actions and the navigate replaces the automatic inheritance of attributes.

Note that some of the features of inheritance are still present in ASL. For example, polymorphic operations are supported.

From the analysts point of view however, the situation is very simple. Generalisations are treated just like Associations at the ASL level. In this chapter, therefore we will discuss the manipulation primitives using the language of Associations and revisit the question of Generalisations later.

## 8.2 Referential Attributes

As discussed in Section Identity and Identifying Attributes executable UML emphasises the modelling of identity through identifying attributes. When such attributes are modelled, any relationships automatically result in the existence of referential attributes<sup>26</sup> to model the identity of an association.

ASL allows the possibility<sup>27</sup> of using manipulation of referential attributes in place of the relationship primitives to manipulate relationships. For example, one could create an instance of an association by setting the value of a referential attribute. If such an approach is used, it must not be mixed with use of the relationship primitives within a given domain model.

If the relationship *primitives* are used:

- Referential attributes will have their values set automatically by the implementation as the link and unlink operations are executed.
- Referential attributes must not be written to by the ASL, except when the referential is also part of an identifier of the class (See Section Creation of Objects).
- Referential attribute values should be read only in order to find out their *value*. They should not be read in order to find out facts about the related object. Use the navigation primitive instead.

Over the next few pages we detail the various relationship primitives supplied by ASL.



<sup>26</sup>.Tools such as iUML manage the creation and maintenance of such referential attributes in an almost entirely automatic fashion.

<sup>27</sup>.We know of no implementation of ASL which actually takes this approach.

## 8.3 Association Navigation

Association navigation is the function whereby the associations specified on the Class Diagram are read in order to determine the set of objects that are linked to an object or set of objects of interest.

Syntax:

```
{<instance handle set>} = <starting handle> -> <multi-valued relationship specification>
```

will return a set of handles of objects that are related to the object specified by the starting handle via the association specified. A set of instance handles is returned when the relationship specification is multivalued in the direction of navigation.

or:

```
{<instance handle set>} = {<starting handle set>} -> <relationship specification>
```

will return a set of handles of objects that are related to the set of objects specified by the starting handle set via the association specified. A set of instance handles is returned irrespective of the multiplicity of the relationship specification in the direction of navigation.

or:

```
{<instance handle set>} = {<starting handle set>} -> <relationship specification> \  
                                     -> <relationship specification> \  
                                     -> <relationship specification> \  
                                     -> ...
```

will return a set of instance handles of objects in the class at the final destination of the chain of associations. A set is returned irrespective of the association multiplicities in the chain, since the start of the navigation is from a set.

or:

```
{<instance handle set>} = <starting handle> -> <relationship specification> \  
                                     -> <relationship specification> \  
                                     -> <relationship specification> \  
                                     -> ...
```

will return a set of instance handles of objects in the class at the final destination of the chain of associations. A set is returned if *at least one* relationship specification in the chain is multi-valued in the direction of navigation.

or:

```
<instance handle> = <starting handle> -> <single-valued relationship specification>
```

will return the *single* instance handle that references the *single* object specified by the starting handle via the association specified. This will be valid only if the association specified is *single valued* in the direction of navigation.

or:

```
<instance handle> = <starting handle> -> <single-valued relationship specification> \
                    -> <single-valued relationship specification> \
                    -> <single-valued relationship specification> \
                    -> ...
```

will return the *single* instance handle that references the object at the final destination of the chain of associations. This will be valid only if *all* the associations in the chain are *single valued* in the direction of navigation.

- Notes:
1. In summary, the result of these navigations is always a set unless the starting handle is single valued and all the associations involved are single valued in the direction of navigation.
  2. It is quite possible that a particular relationship specification or chain of specifications could return a set of handles where the same object appears multiple times. This issue is discussed further in a later section on sets sequences and bags.
  3. If the {<starting handle set>} is empty (or is **UNDEFINED** if a single value), then the result will be considered as a run time error.
  4. The returned {<instance handle>} set can be empty (or be **UNDEFINED** if a single value) if any of the associations in the chain are conditional in the direction of navigation.
  5. The instance handle set {<starting handle>} can be replaced by a class specification thus:<sup>28</sup>

Syntax:

```
{<instance handle set>} = <class> where <class condition> -> <relationship specification>
```

which is equivalent to the following:

```
{<starting handle set>} = find <class> where <class condition>
{<instance handle set>} = {<starting handle set>} -> <relationship specification>
```

or:

```
{<instance handle>} = <class> where <class condition> -> <relationship specification> \
                    -> <relationship specification> \
                    -> <relationship specification> \
                    -> ...
```

which is equivalent to the following:

```
{<starting handle set>} = find <class> where <class condition>
{<instance handle set>} = {<starting handle set>} -> <relationship specification> \
                    -> <relationship specification> \
                    -> <relationship specification> \
                    -> ...
```



<sup>28</sup>The forms of navigation involving <class> **where** <class condition> are not supported in iUML Simulator. We are considering withdrawing these constructs from the language.



## 8.4 Relationship Specifications

A relationship specification specifies exactly which association is required to be created, navigated or deleted. There are four forms of an relationship specification:

**Relationship Number:** Syntax: **R**<number>

This is the number of the association as shown on the Class Diagram (e.g. R1). This is sufficient to specify only the association required.

Example **R3**

**Relationship Role:** Syntax: **R**<number>."**<role name>**"<sup>29</sup>

The text that appears on the Class Diagram at the destination end of the association (e.g. "is owned by"). This form specifies not only the association, but also the *direction* that is required. This is of importance when navigating "reflexive" associations where classes are related to themselves.

Example: **R3**."**is\_owned\_by**"

**Qualified Number:** Syntax: **R**<number>.<class name>

This identifies not only a specific association, but the class that is the target of the association navigation. This is required, for example in order to distinguish which class is required when an associative class is present on the association.

Example: **R3**.Owner

**Qualified Role:** Syntax: **R**<number>."**<role name>**".<class name>

This identifies the association, the direction and which class is required.

Example: **R3**."**is owned by**".Owner

The meaning and uses of all these forms is perhaps best illustrated by use of an example.



<sup>29</sup>iUML Simulator does not allow white space in role names. Note that iUML Modeller will automatically replace spaces with underscores when performing a code generation dump.

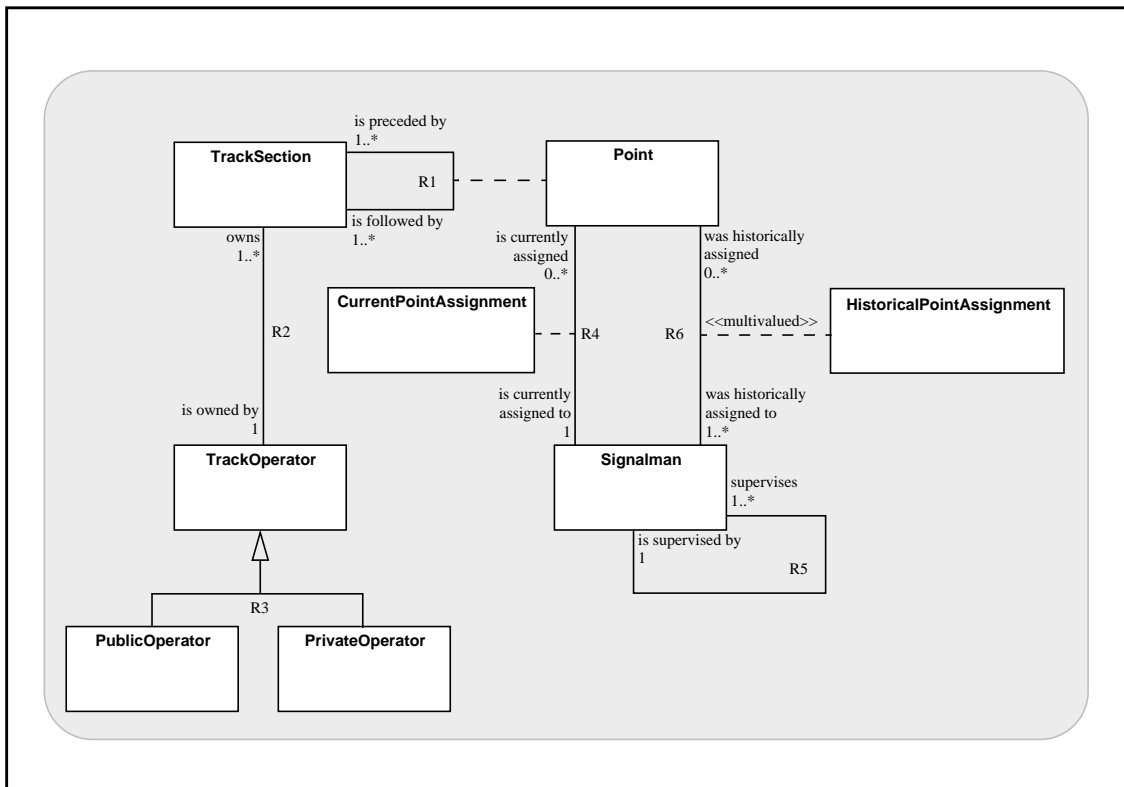


Figure 1: Class Diagram

For the Class Diagram shown, we could specify the following navigations<sup>30</sup>, each by use of the four different relationship specification types discussed. Not all forms are valid, however.

**From “TrackSection” to “TrackOperator”**

<code>operator = this -&gt; R2</code>	is valid
<code>operator = this -&gt; R2."is_owned_by"</code>	is valid
<code>operator = this -&gt; R2.TrackOperator</code>	is valid
<code>operator = this -&gt; R2."is_owned_by".TrackOperator</code>	is valid

**From “TrackOperator” to “PrivateOperator”**

<code>privateOperator = this -&gt; R3</code>	is ambiguous (which subclass?)
<code>privateOperator = this -&gt; R3.PrivateOperator</code>	is valid



30. In each of the ASL navigation statements “this” refers to an instance of the ‘source’ class - for example, in the first table “this” refers to an instance of “TrackSection”.

### From “PrivateOperator” to “TrackOperator”

---

<code>trackOperator = this -&gt; R3</code>	is valid
<code>trackOperator = this -&gt; R3.TrackOperator</code>	is valid

---

### From “Signalman” to “Signalman”

---

<code>supervisor = this -&gt; R5</code>	is ambiguous (which direction?)
<code>supervisor = this -&gt; R5."is_supervised_by"</code>	is valid
<code>supervisor = this -&gt; R5.Signalman</code>	is ambiguous (which direction?)
<code>supervisor = this -&gt; R5."is_supervised_by".Signalman</code>	is valid

---

### From “Point” to “Signalman” (through R4)

---

<code>controller = this -&gt; R4</code>	is ambiguous (which class?)
<code>controller = this -&gt; R4."is_currently_assigned_to"</code>	is ambiguous (which class?)
<code>controller = this -&gt; R4.Signalman</code>	is valid
<code>controller = this -&gt; R4."is_currently_assigned_to".Signalman</code>	is valid

---

### From “Point” to “CurrentPointAssignment”

---

<code>assignment = this -&gt; R4</code>	is ambiguous (which class?)
<code>assignment = this -&gt; R4."is_currently_assigned_to"</code>	is ambiguous (which class?)
<code>assignment = this -&gt; R4.PointAssignment</code>	is valid
<code>assignment = this -&gt; R4."is_currently_assigned_to".PointAssignment</code>	is valid

---

### From “Point” to “Signalman” (through R6)

---

<code>{oldControllers} = this -&gt; R6</code>	is ambiguous (which class?)
<code>{oldControllers} = this -&gt; R6."was_historically_assigned_to"</code>	is ambiguous (which class?)
<code>{oldControllers} = this -&gt; R6.Signalman</code>	is valid
<code>{oldControllers} = this -&gt; R6."was_historically_assigned_to".Signalman</code>	is valid

---

### From “HistoricalPointAssignment” to “Signalman”

oldController = <b>this</b> -> R6	is ambiguous (which class?)
oldController = <b>this</b> -> R6."was_historically_assigned_to"	is ambiguous (which class?)
oldController = <b>this</b> -> R6.Signalman	is valid
oldController = <b>this</b> -> R6."was_historically_assigned_to".Signalman	is valid

### From “TrackSection” to “TrackSection”

{previousSections} = <b>this</b> -> R1	is ambiguous (which direction?)
{previousSections} = <b>this</b> -> R1."follows"	is ambiguous (which class?)
{previousSections} = <b>this</b> -> R1.TrackSection	is ambiguous (which direction?)
{previousSections} = <b>this</b> -> R1."follows".TrackSection	is valid

In summary the analyst must ensure that the following are **unambiguously** specified:

- the **relationship** (whether it be an association or a generalisation);
- the **direction** of the navigation;
- the **destination class**.

**The Relationship:** The relationship can be specified by:

- relationship number
- relationship role<sup>31</sup> (if unambiguous and not a generalisation)

**Direction of Navigation:** The direction of navigation can be specified by:

- default (in non-reflexive associations)
- relationship role (if unambiguous and not a generalisation)
- class name (in non-reflexive associations)

**Destination Class:** The destination class can be specified by:

- default (only for non-reflexive, non-associative associations, and for navigations from an object of a subclass to the related object of the superclass in a generalisation)
- relationship role (in non-associative associations)
- class name



<sup>31</sup> Whilst it is feasible to identify the relationship solely from the relationship role, the legal forms of relationship specification in ASL do not allow this - it should be always combined with the relationship number.

It is the responsibility of the analyst to provide *sufficient* information to identify all the necessary parameters. Note however, that it is quite reasonable for the analyst to specify *more* information than is required in order to make the ASL clearer and more readable, although this may require more model maintenance.

**Note:** In the case of a reflexive association with the *same* role at each end, it is indeterminate what result will be returned by a navigation of that association.

## 8.5 Link Creation

Instances of associations (UML “links”) between objects may be created using the “**link**” statement:

Syntax:

```
link <source instance handle> <relationship specification> <destination instance handle>
```

or:

```
link <source instance handle> <relationship specification> <destination instance handle> \
    using <association class instance handle>
```

This causes the specified objects to be linked together via the specified association, such that the new association instance may subsequently be navigated using the “->” primitive.

Where:

<source instance handle>	is the single instance handle of the first object to be linked.
<destination instance handle>	is the single instance handle of the second object to be linked.
<relationship specification>	is the specification of the association from the source class to the destination class. This can be of any of the forms described in the previous section. Note that this specification should be framed as if navigating from “source class” to “destination class”.
<association class instance handle>	is the handle of an existing object that is to be used as the association class instance for this association instance. This can only be used if the specified association has been defined as being associative in the xUML model.

- Notes:**
1. When a “link” is instantiated via this mechanism, the values of any non-identifying referential attributes will be set by the architecture automatically. The values of *identifying* referential attributes (i.e. those part of any identifier) will not be set since the analyst is required to set them as part of the creation statement for the class (See “Object Creation”). In this latter case, if at run time the value of any identifying attribute in one object is not the same as the value of the corresponding (identifying) referential attribute in the object being linked, this is considered to be a run time error.<sup>32</sup>
  2. Although we talk here of “source” and “destination” class, it should be emphasised that all associations in an xUML model are “two-way”. Creation of an instance of an association implies the subsequent ability to navigate the association from either end. The concept of “source” and “destination” here is used to ensure that there is no ambiguity with regard to the relationship specification.
  3. Both the source and destination handles must be single instance handles (not sets).
  4. If an attempt is made to link two objects together via the same association twice, then this is regarded as a run time error<sup>33</sup>.
  5. The “**using** <association class instance>” form must be used when an association that involves an association class is being instantiated. Therefore, the association class instance must have already been created before the association is instantiated.



<sup>32</sup>.iUML Simulator does not check this.

<sup>33</sup>.iUML Simulator does not check for this condition.

## 8.6 Link Deletion

Instances of associations can be destroyed using the unlink statement.

Syntax:

```
unlink <source instance handle> <relationship specification> <destination instance handle>
```

Where:

<source instance handle>	is the single instance handle of the first object to be unlinked.
<destination instance handle>	is the single instance handle of the second object to be unlinked.
<relationship specification>	is the specification of the association from the source class to the destination class. This can be of any of the forms described in the previous sections. Note that this specification should be framed as if navigating from “source class” to “destination class”.

- Notes:**
1. Although we talk here of “source” and “destination” class, it should be emphasised that all associations in xUML models are “two-way”. Deletion of an instance of an association implies the subsequent inability to navigate the association from either end. The concept of “source” and “destination” here is used to ensure that there is no ambiguity in relation to the relationship specification.
  2. Both the source and destination handles must be single instance handles (not sets).
  3. If an attempt is made to unlink two objects which are not linked by the specified association, then this is regarded as a run time error.
  4. If an association with an association class is unlinked then the association class instance will not be deleted. The analyst must specify this explicitly.
  5. If a mandatory association is deleted, participating objects 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/subclass associations.

## 8.7 Generalisation Relationships Revisited

As we discussed earlier in Section 8.1, “Association vs. Generalisation,” on page 31, ASL regards superclass and subclass objects as separate, but linked, instances. That is, creating a subclass object does not automatically create a superclass object. So referring to the sample Class Diagram we used in previous sections, if we were to create a new object of the PrivateOperator subclass, we would write ASL that looked something like this:

```
# Create the superclass object
newTrackOperator = create TrackOperator with ...

# Create the subclass object
newPrivateOperator = create PrivateOperator with ...

# Create the generalisation relationship instance
link newTrackOperator R3 newPrivateOperator
```



## 8.8 Correlated Associative Navigation<sup>34</sup>

In Section 8.4, “Relationship Specifications,” on page 35 we presented an example of a navigation from an object through an association to the related association object. Referring to the sample Class Diagram in Figure 1 on page 36 we had:

```
assignment = this -> R4.PointAssignment
```

when navigating from a “Point” object (referred to by “this”), through the association R4 to the related association object “PointAssignment”. In this particular case, since R4 has multiplicity of 1 in the direction of navigation, the handle returned is a single value (as opposed to a set).

If, however, we had chosen to navigate from a “Signalman” object (referred to by the instance handle “theSignalman”) through R4 to the related association objects we would have to have write:

```
{assignments} = theSignalman -> R4.PointAssignment
```

This is because for a given object of class “Signalman”, there will be many related objects of class “Point” and hence many related objects of the class “PointAssignment” - one for each association instance.

In practice, it will often be the case that we wish to find the object of the association class that is related to a particular **pair** of related objects of the other classes. To do this we can use a correlated associative navigation thus:

```
<association class instance handle> = <starting handle a> and \  
    <starting handle b> -> <associative relationship spec>
```

Where:

<association class instance handle>	is the returned handle for the object of the association class that correlates <starting handle a> and <starting handle b>.
<starting handle a>	is the instance handle for one of the pair of related objects.
<starting handle b>	is the instance handle for the other of the pair of related objects.
<associative association spec>	must be an relationship specification that navigates to an association class for an association that links the two classes. The specification should be framed as if navigating from <starting handle a> to <starting handle b>

**Example:** # If “thePoint” and “theSignalman” are handles on instances of the  
# corresponding classes then:  
ourAssignment = thePoint **and** theSignalman -> R4.PointAssignment

- Notes:**
1. A correlated associative navigation cannot be placed as a part of a chain of navigations;
  2. If the link between the association class and the association is stereotyped <<multivalued>> (as it is for association R6 in Figure 1), then a correlated associative navigation will return a set of instance handles (See 8.9 “Multivalued Association Classes” on page 44).

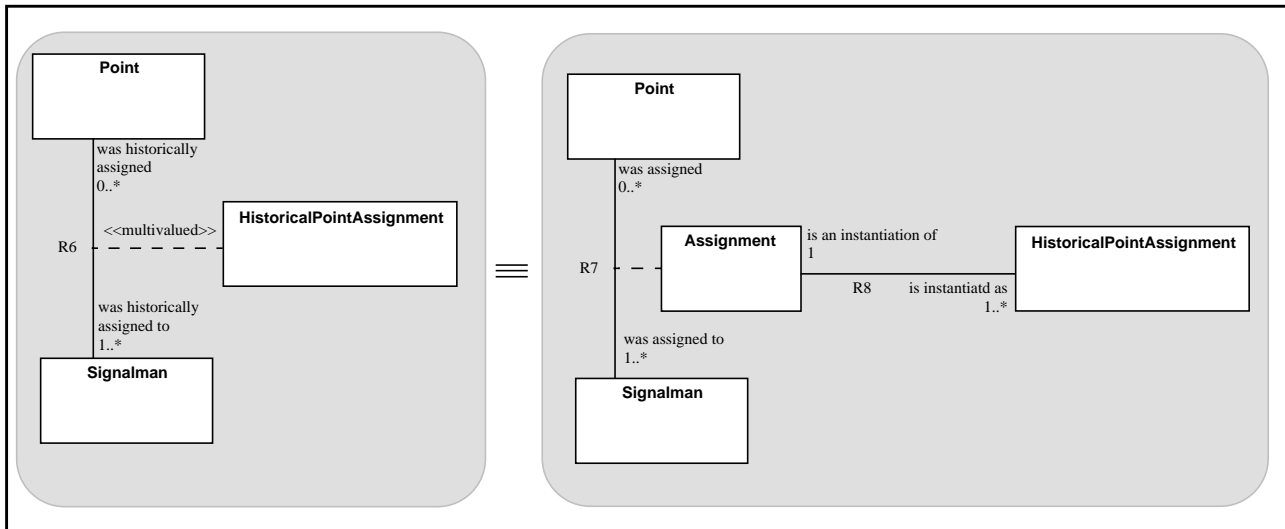


34. iUML Simulator fully supports correlated associative navigation, however the compile time error checking is not as complete as for the other navigation forms. See the release notes for details.

## 8.9 Multivalued Association Classes

In associations where there can be many instances of the associative class per instance of the association, special treatment is required. Such associations appear on a Class Diagram with the stereotype `<<multivalued>>` attached to the link between the association class and the association - as is the case for R6 in Figure 1 on page 36.

Associations that are stereotyped in this way have a semantic equivalent that is illustrated in Figure 2 below.



**Figure 2: Semantic Equivalence for `<<multivalued>>` Association Classes**

Note now the difference between the abstractions of the two association classes “HistoricalPointAssignment” on the left and “Assignment” on the right.

Each instance of “HistoricalPointAssignment” on the left is an abstraction of an assignment of an instance of “Point” to an instance of “Signalman” - and if a particular “Point” is assigned to the same “Signalman” n times there will be n instances of “HistoricalPointAssignment”. Hence there are multiple instances of “HistoricalPointAssignment” for a given instance of association R6.

Each instance of “Assignment” on the right is an abstraction of the fact that one or more assignments have been made between an instance of “Point” and an instance of “Signalman”. The details of each instantiation of such an assignment in this equivalent model are captured via the association R8 and the respective instances of “HistoricalPointAssignment”.

If desired, a standard modelling pattern may be applied in order to convert models containing such `<<multivalued>>` association classes into their semantic equivalent representation.<sup>35</sup>



35.iUML fully supports `<<multivalued>>` association classes, so there is no need to apply such a modelling pattern.

## 8.10 Associate and Unassociate

In the case of a multivalued association class, the association itself must be created and deleted using the “link using” and “unlink” constructs respectively. However, two additional constructs are used to specify the addition or removal of an association object to or from an **existing** association instance (or UML “link”).

### Addition of a new association object:

Syntax:

```
associate <source instance handle> <relationship specification> \  
<destination instance handle> using <association class instance handle>
```

Where:

<source instance handle>	is the handle of the first linked object (the “source” object)
<destination instance handle>	is the handle of the second linked object (the “destination” object)
<relationship specification>	is the specification of the relationship from the “source object” to the “destination object”.  This can be of any of the forms described in previous sections, providing that it is unambiguous. Note that this specification should be framed as if navigating from “source object” to “destination object”.
<association class instance handle>	is the handle of an existing object instance that is to be added as an association object to the respective (multivalued) association instance.

- Notes:**
1. The source and destination objects must be linked through the association specified at the time this statement is executed. Any attempt to associate an association object to an unlinked source & destination object pair is regarded as a run time error.
  2. There is no distinction between the association object linked when the association was created (by “link using”) and an association object linked subsequently (using “associate”).

### Removal of an existing association object:

Syntax:

```
unassociate <source instance handle> <relationship specification> \  
  <destination instance handle> from <association class instance handle>
```

Where:

<source instance handle>	is the handle of the first linked object (the “source” object)
<destination instance handle>	is the handle of the second linked object (the “destination” object)
<relationship specification>	is the specification of the relationship from the “source object” to the “destination object”.  This can be of any of the forms described in previous sections, providing that it is unambiguous. Note that this specification should be framed as if navigating from “source object” to “destination object”.
<association class instance handle>	is the handle of the object that is to be removed as an association object for the respective association.

- Notes:
1. The source and destination instances must be linked through the association specified at the time this statement is executed.
  2. The association object must have been associated with the association instance (using either “**link using**” or “**associate**”) prior to this statement being executed.
  3. It is the responsibility of the analyst to ensure that an association that involves an association class has at least one association object associated with it. If the last association object is deleted, the analyst must ensure that the corresponding association is also deleted using “**unlink**”.

## 9 Signal Generation

Signals may be generated and sent to objects, classes and to non-counterpart terminators using the “**generate**” statement:

Syntax:

```
generate <signal specification>( <signal parameters> )
```

in the case of a creation signal, a signal destined for an assigner state machine or a non-counterpart terminator.

or:

```
generate <signal specification>( <signal parameters> ) to <destination instance handle>
```

in the case of a signal destined for an existing state machine.

Where:

<code>&lt;signal specification&gt;</code>	<p>consists of the signal label and the signal name in the following syntax:</p> <pre>&lt;key letter&gt;&lt;signal number&gt;:&lt;signal name&gt;</pre> <p>The key letter can be that of a class or a non-counterpart terminator. In the case of a signal destined for an assigner state machine, the key letter will be of the form:</p> <pre>&lt;association class key letter&gt;-A</pre>
<code>&lt;signal parameters&gt;</code>	<p>is a comma separated list of parameters for the signal. If there are no parameters then the empty signal parameter list must be used - for example:</p> <pre><b>generate</b> NOZ8:triggerDepressd() <b>to</b> theNozzle</pre> <p>Signal parameters may be of any valid ASL type.</p>
<code>&lt;destination instance handle&gt;</code>	<p>is the handle of the destination object to which the signal is directed.</p> <p>This handle must be for a class of the correct type - i.e. the object that the handle references must be for the same class as that indicated by the &lt;key letter&gt; in the signal label.</p>

- Notes:
1. All the signal parameters must be supplied in the *same order* as that in the definition in the xUML model.
  2. Use of the “to” clause is invalid for signals to assigner state machines, non-counterpart terminators and the creation states of state machines.
  3. If a signal is polymorphic (i.e. it is directed at a superclass), then the <destination instance handle> must be a handle for an instance of the *superclass*.

Examples:

```
# Example 1: Send a creation request to Disk Request
#
# In this example, the signal causes entry to an initial
# (creation) state and so does not require a "to" clause
generate DR5:Create_Disk_Request()
```

```
# Example 2: Send a signal to an existing object of class Robot
#
# Assume that "this" is a handle on an object of a class which
# is associated with the Robot class through the association R1,
# and "x" and "y" are local variables that define the position we
# want the "requiredRobot" to move to when reset.

requiredRobot = this -> R1.Robot

# "requiredRobot" is thus a handle on an object of
# the "Robot" class

generate ROB1:Reset(x,y) to requiredRobot
```

## 10 Arithmetic and Logical Operations

### 10.1 Constants and Limits

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 xUML models should require minimal use of constants since such data could be stored as attributes of specification classes. Nevertheless, so as not to be restrictive ASL defines the following syntax for constants:

Syntax:

The syntax depends on the base datatype:

<b>Integer:</b>	1	42	-127	etc.
<b>Real:</b>	1.0	4.5	-56.0	1E27 etc.
<b>Date:</b>	yyyy . mm . dd			
<b>Time of Day:</b>	hh : mm : ss			
<b>Text:</b>	"text"			
<b>Enumeration:</b>	'enumeration_value'			

A full definition of the syntax can be found in the syntax summary at the end of this reference manual.

For the above types the following bounds are defined:

<b>Integer:</b>	-Infinity	+Infinity
<b>Real:</b>	-Infinity	+Infinity
<b>Date:</b>	-Infinity	+Infinity
<b>Time of Day</b>	00:00:00	23:59:59
<b>Text:</b>	""	Unlimited length string
<b>Enumeration</b>	Bounded by the definition of the type	

Of course no practical implementation can achieve all of these. Analysts should therefore make use of User Defined Types (based on the appropriate underlying

type) with acceptable ranges specified. Architectures must then achieve an effective implementation of these types. For the date types, the calendar system in use will depend on the software architecture. If more sophistication is required, this should be made the subject of a calendar domain.

Special Note: Throughout ASL only plain single and double quotes are used. Unfortunately many word processors use “smart quotes” that automatically change text to use balanced and symmetric quotes. While we have made every effort to ensure that has not happened in the production of this document we state the ASL policy here for the avoidance of doubt.

For the remaining types, the following constant values are defined:

Boolean:	<b>TRUE</b>	<b>FALSE</b>	
Instance handle:	<b>UNDEFINED</b>	<b>ALREADY_DEFINED<sup>a</sup></b>	<b>ERROR<sup>a</sup></b>

a.iUML Simulator does not support **ALREADY\_DEFINED** or **ERROR**

- Notes:
1. Constants may be used in place of <local variable> in any ASL construct where the <local variable> appears on the right hand side of the assignment operator (the “=” symbol).
  2. When it is required to assign a constant enumeration value to a local variable, it is necessary to explicitly state the enumeration type. (This must be so, since many enumerations may share the same enumeration values. It is therefore impossible to unambiguously determine the enumeration type from a single value assignment).

Thus such assignments must be written as follows:

```
<local variable> of <enumeration type> = '<enumeration value>'
```

Example: `bad_status of status_type = 'failed'`



## 10.2 Arithmetic Calculations

Arithmetic calculations may be performed during an assignment to a local variable.

Syntax:

```
<local variable> = <arithmetic expression>
```

Where:

<code>&lt;local variable&gt;</code>	is the variable to which the results of the expression are assigned.
<code>&lt;arithmetic expression&gt;</code>	is an arithmetic expression making use of the normal arithmetic operations on local data items (local variable, constant or <code>&lt;instance handle&gt;.&lt;attribute name&gt;</code> ).

- Notes:**
1. Components of arithmetic expressions cannot be complex (for example embedded ASL operation invocations).
  2. Arithmetic operations are defined for numeric base types only (**Integer** and **Real**) and for user defined types based upon them.
  3. The actual precision and truncation rules for arithmetic calculation depends on the software architecture and implementation domains in use. Clearly, the software architecture must attempt to provide the best possible match to the constraints specified in a user defined type or to unconstrained base types.
  4. The following arithmetic operations are defined<sup>36</sup>: + - / \* ^
  5. There are no precedence rules except for those defined by use of parentheses ( ).
  6. The discussion regarding data type rules describes more about arithmetic expressions.



<sup>36</sup>iUML Simulator does not support the use of the power operator (^).

## 10.3 Local Variable Assignment

Most of the statements that we have considered so far result in assignment to a local variable. In addition to those mentioned, simple copying of local variables is permitted:

Syntax:

```
<local variable> = <local variable>
```

or:

```
{ <local variable set> } = { <local variable set> }
```

These will result in the copying of the local variable.

**Notes:** 1. Assignment of instance handles is permitted; e.g.

```
new_object = existing_object
```

2. Assignment of instance handle sets is permitted, and results in a copy of the set being made; e.g.

```
{new object set} = {existing object set}
```

3. Whether an actual copy of the set is made is architecture dependant. The ASL rules are such that it must appear *as if* a copy is made and thus if {existing object set} is changed subsequent to the execution of this statement, {new object set} will not change.

4. Assignment of sets of structures is permitted, and results in a copy of the set being made; e.g.

```
{new structure set} = {existing structure set}
```

The same considerations apply as for copying of instance handle sets. Sets of structures are discussed more fully in a later section.

## 10.4 Logical Conditions

Many ASL statements use logical conditions. Such a condition performs a series of logical tests on data item values.

Syntax:

```
<component> <binary logical operator> <component>
```

or:

```
! <component>
```

This defines a Boolean data type that receives the value **TRUE** or **FALSE**. Where:

```
<component> is either another <condition> or a <data item>
```

**<data item> can be:**

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

```
<local variable name>
```

```
<constant>
```

```
countof { <instance handle set> }
```

```
countof <class>
```

The following table shows the logical operations that are defined in ASL:

Symbol	Keyword	Meaning	Valid Data Types
!	<b>not</b>	Logical Negation	Boolean
=	<b>equals</b>	Equality	Integer, Date, Time, Text, Enum, Boolean, Instance Handle <sup>a</sup> , Structure <sup>b</sup>
!=	<b>not-equals</b>	Inequality	Integer, Date, Time, Text, Enum, Boolean, Instance Handle
<	<b>less-than</b>	Inequality	Real, Integer, Date, Time
>	<b>greater-than</b>	Inequality	Real, Integer, Date, Time
<=	<b>less-than-or-equal-to</b>	Inequality	Integer, Date, Time
>=	<b>greater-than-or-equal-to</b>	Inequality	Integer, Date, Time
&	<b>and</b>	Logical And	Boolean
	<b>or</b>	Inclusive Logical Or	Boolean

a. iUML Simulator does not support comparison of instance handles, although comparison of a handle with UNDEFINED is supported.

b. Equality is defined only if it is defined (recursively) for the members of the structure. See section Equality for more details.

- Notes:
1. These will be evaluated with a precedence according to the order shown (highest precedence first), although parentheses () may be used to alter this if desired. Certain operations are valid for certain data types only.
  2. The last column shows the data types for which the operation is valid.
  3. Either the Symbol or the Keyword may be used in ASL

# 11 Operations

## 11.1 ASL Operations in the Context of a Domain Model

ASL provides for operation definitions and operation calls. Formally, in ASL, this is achieved through the mechanism of an ASL function.

The following sections cover the definition and calling of ASL functions and their specific uses as synchronous operations provided by elements of an executable UML model.

## 11.2 Defining and Calling an ASL Operation

To use operations, a formal definition of the interface to the operation and the implementation of the operation must be made. This is done in a “function definition”.

Such definitions are associated with the operations provided by the following model element types within a UML model:

- domains (See 11.3 “Domain Scoped Operations” on page 56)
- classes (See 11.4 “Class Scoped Operations” on page 58)
- objects (See 11.5 “Object Scoped Operations” on page 60)

## 11.3 Domain Scoped Operations

A domain scoped operation is one that is associated with the domain that it is declared in, but not with any specific class or object within that domain. This provides a partially anonymous interface to the domain that can be called from outside the domain<sup>37</sup>. Thus users of the domain (i.e. bridges) are not coupled to the internal structure of the domain.

### Operation Definition Syntax (for domain scoped operations):

```

define function <operation specification>
input <parameter 1 name>:<parameter 1 type>,<parameter 2 name>: ...
output <result 1 name > : <result 1 type>,<result 2 name> : ...
    <ASL statements>
enddefine
    
```

Where:

<operation specification>	identifies the domain scoped operation and conforms to the following syntax:  <domain key letter><operation number>::<operation name>
<parameter 1 name>:<parameter 1 type> <parameter 2 name>:<parameter 2 type> ...	are comma separated pairs of formal parameter names and their types that are passed into the operation <sup>a</sup>
<result 1 name> : <result 1 type> <result 2 name> : <result 2 type> ...	are comma separated pairs of formal parameter names and their types that are returned by the operation <sup>b</sup>
<ASL Statements>	are the ASL statements that define the method for the respective operation.

a. There are limitations in iUML Simulator with respect to the way input parameters can be used within the function. See release notes for details.

b. In iUML Simulator any output parameter from a function cannot be placed on the right hand side of a statement within a function. Such parameters should be placed on the left hand side of assignment statements only. This will require the use of temporary local variables if necessary.

- Notes:
1. See “Notes Common To All Operation Definitions:” on page 62.
  2. The handle “**this**” is not available in the method (the <ASL Statements>) of a domain scoped operation.



37. Domain scoped operations may also be called from within the domain in which they are defined.

### Operation Invocation Syntax (for domain scoped operations):

```
[<result 1> , <result 2> ,...] = <operation specification>[ <parameter 1> , \
<parameter 2> ,...]
```

Where:

<result 1> <result 2> ...	are the comma separated names of the actual parameters returned by the invocation.
---------------------------------	--

<operation name>	identifies the operation, which conforms to the following syntax:
------------------	---

<domain key letter><operation number>::<operation name>

<parameter 1> <parameter 2> ...	are the comma separated names of the actual input parameters passed with the invocation.
---------------------------------------	--

Notes: 1. See "Notes Common To All Operation Invocations:" on page 63.

## 11.4 Class Scoped Operations

A class scoped operation is one that is associated with and provided by a specific class in a domain. For example, this might be used to provide a creation operation for the class. Such an operation would create a new object of the class and set up any initial conditions required. If the class was active, it could then generate a signal to the class to drive it through its lifecycle.

Class scoped operations have exactly the same definition and call syntax as any ASL operation except that a special syntax is defined for the name

### Operation Definition Syntax (for class scoped operations):

```

define function <operation specification>
input <parameter 1 name>:<parameter 1 type>,<parameter 2 name>: ...
output <result 1 name > : <result 1 type>,<result 2 name> : ...
    <ASL statements>
enddefine
    
```

Where:

<operation specification>	identifies the class scoped operation and conforms to the following syntax:  <class key letter><operation number>:<operation name>
<parameter 1 name>:<parameter 1 type> <parameter 2 name>:<parameter 2 type> ...	are comma separated pairs of formal parameter names and their types that are passed into the operation <sup>a</sup>
<result 1 name> : <result 1 type> <result 2 name> : <result 2 type> ...	are comma separated pairs of formal parameter names and their types that are returned by the operation <sup>b</sup>
<ASL Statements>	are the ASL statements that define the method for the respective operation.

a. There are limitations in iUML Simulator with respect to the way input parameters can be used within the function. See release notes for details.

b. In iUML Simulator any output parameter from a function cannot be placed on the right hand side of a statement within a function. Such parameters should be placed on the left hand side of assignment statements only. This will require the use of temporary local variables if necessary.

- Notes:
1. See “Notes Common To All Operation Definitions:” on page 62.
  2. The handle “**this**” is not available in the method of a class scoped operation.
  3. The <operation number> name space is disjoint from the <signal number> name space employed for signals directed at the class (See 9 “Signal Generation” on page 47).  
Thus a class scoped operation and a signal may share the same number for a given class.
  4. The <operation name> should not be the same as the name of any signal directed at the same class.



### Operation Invocation Syntax (for class scoped operations):

```
[<result 1> , <result 2> ,...] = <operation specification>[ <parameter 1> , \
<parameter 2> ,...]
```

Where:

<result 1> <result 2> ...	are the comma separated names of the actual parameters returned by the invocation.
---------------------------------	--

<operation name>	identifies the operation, which conforms to the following syntax:
------------------	---

<class key letter><operation number>:<operation name>

<parameter 1> <parameter 2> ...	are the comma separated names of the actual input parameters passed with the invocation.
---------------------------------------	--

Notes: 1. See "Notes Common To All Operation Invocations:" on page 63.

## 11.5 Object Scoped Operations

An object scoped operation provided by a class is one that is applied to a specific object at run time in an analogous way to “existing instance signals”.

Such operations require a specific syntax both for their definition and for their call.

### Operation Definition Syntax (for object scoped operations):

```

define instance function <operation specification>
instance this : <class name>
input <parameter 1 name>:<parameter 1 type>,<parameter 2 name>: ...
output <result 1 name > : <result 1 type>,<result 2 name> : ...
    <ASL statements>
enddefine
    
```

Where:

<operation specification>	identifies the class scoped operation and conforms to the following syntax:  <class key letter><operation number>:<operation name>
<class name>	is the name of the class that provides the object scoped operation.
<parameter 1 name>:<parameter 1 type> <parameter 2 name>:<parameter 2 type> ...	are comma separated pairs of formal parameter names and their types that are passed into the operation <sup>a</sup>
<result 1 name > : <result 1 type> <result 2 name> : <result 2 type> ...	are comma separated pairs of formal parameter names and their types that are returned by the operation <sup>b</sup>
<ASL Statements>	are the ASL statements that define the method for the respective operation.

a. There are limitations in iUML Simulator with respect to the way input parameters can be used within the function. See release notes for details.

b. In iUML Simulator any output parameter from a function cannot be placed on the right hand side of a statement within a function. Such parameters should be placed on the left hand side of assignment statements only. This will require the use of temporary local variables if necessary.

- Notes:
1. See “Notes Common To All Operation Definitions:” on page 62.
  2. The instance handle “**this**” is available for use within the method for an object scoped operation.
  3. The <operation number> name space is disjoint from the <signal number> name space employed for signals directed at the class (See 9 “Signal Generation” on page 47).  
Thus an object scoped operation and a signal may share the same number for a given class.
  4. The <operation name> should not be the same as the name of any signal directed at the same class.

## Operation Invocation Syntax (for object scoped operations):

```
[<result 1> , <result 2> ,...] = <operation specification>[ <parameter 1>, \
<parameter 2>,...] on <handle>
```

Where:

<code>&lt;result 1&gt;</code> <code>&lt;result 2&gt;</code> ...	are the comma separated names of the actual parameters returned by the invocation.
<code>&lt;operation name&gt;</code>	identifies the operation, which conforms to the following syntax:  <code>&lt;class key letter&gt;&lt;operation number&gt;:&lt;operation name&gt;</code>
<code>&lt;parameter 1&gt;</code> <code>&lt;parameter 2&gt;</code> ...	are the comma separated names of the actual input parameters passed with the invocation.
<code>&lt;handle&gt;</code>	is an instance handle that refers to the object which provides the operation.

- Notes:
1. See “Notes Common To All Operation Invocations:” on page 63.
  2. If, at compile time, the object scoped operation is found to have been called with the `<handle>` being of a type different from that indicated by the **instance** declaration in the respective operation definition then this is regarded as a compile time error<sup>38</sup>.
  3. If an object scoped operation is declared for a superclass, and the implementation of the operation is declared<sup>39</sup> to be deferred down a specified subclass relationship then there must be an operation with the **same name** declared for each subclass of the association. At run time, the user ASL must invoke the superclass operation on a superclass object. The implementation will then automatically invoke the appropriate operation on the correct subclass.



38. The **instance** declaration is necessary to support polymorphic operations. iUML Modeller automatically generates all the appropriate instance definitions and so the analyst need not be concerned with this issue.

39. There is no actual syntax within ASL for making this declaration. The information is held directly within a CASE tool repository.

### Notes Common To All Operation Definitions:

1. In CASE tools such as iUML, it is not necessary for the analyst to write the specify the definition of the operation using the “define function ... input ... output ...enddefine” syntax. The analyst need only write the method for the operation (i.e. the <ASL Statements>).
2. The <operation specification> must be unique within a domain.
3. Parameters can be of any valid ASL type, and can be either single valued or sets of instance handles or structures.
4. If there are no input and/or output parameters the <parameter name>:<parameter type> clauses should be omitted from the respective parts of the operation definition.
5. If an input or an output parameter is an instance handle, then the name of the class to which the handle refers should be used for the type.

For example, suppose that the object scoped operation “AC1:getAccountOwner” provided by the “Account” class returns the parameter “theOwner” of type “Customer” instance handle, the operation might be defined as:

```
define instance function AC1:getAccountOwner
instance this : Account
input
output theOwner : Customer
    theOwner = this -> R1
enddefine
```

6. If the input or output parameters are sets (of instance handles or structures), then the parameter name must be surrounded by braces {}.

For example, suppose that the class scoped operation “ROB7:findEveryIdleRobot” provided by the “Robot” class returns the parameter {idleRobots} of type “Robot” instance handle, the operation might be defined as:

```
define function ROB7:findEveryIdleRobot
input
output {idleRobots} : Robot
    {idleRobots} = find Robot where status = 'Idle'
enddefine
```

Such a definition indicates that a set of “Robot” handles is expected to be returned.

7. The ASL language definition does not specify whether input or output parameters for operations are to be passed by value or by reference. Implementations may do either or both. However, ASL in an operation must not attempt to modify input parameters<sup>40</sup>.



40.For backwards compatibility reasons, iUML Simulator allows modification of input parameters that are sets of structures. Such modifications remain visible in the calling block of ASL. Analysts producing new ASL should not use this feature.

Notes Common To All  
Operation Invocations:

1. The input and return parameter names used in the definition may differ from those used in the invocation. Matching is achieved by the ordering of items within the brackets “[ ]” and within the input and output parts of the respective operation definition.
2. If there are no input and/or output parameters empty brackets “[ ]” must be used in the respective part of the invocation.
3. Recursive operation calls are permitted.



## 12 Timer and Time Operations

### 12.1 The xUML Timer

The xUML formalism provides a built-in timer mechanism modelled closely on that of Shlaer Mellor<sup>41</sup>. The timer is used to request that a specified signal is sent when either a relative or absolute time has expired.

Analysts can make use of the functionality provided by the xUML Timer from within their own models by:

- invoking operations on the Timer terminator, or;
- sending signals to the Timer terminator.

Such operations and signals are “bridged” to operations supplied by another domain<sup>42</sup> that implements the timer functionality.

The following sections therefore illustrate how, using ASL, the interface to the xUML Timer is in order to obtain the required behaviour.



41. “Object Lifecycles - Modeling the World in States” p53

42. In iUML and iUML Simulator the operation of instantiating a timer terminator in a domain automatically makes available the various functions signals and data types required by the terminator, and provides the implementation of the domain that provides the xUML Timer together with the bridge implementations for the various operations and signals defined for the timer terminator.

### Invoking the Operations Provided By the xUML Timer Interface

The 3 operations provided by the xUML Timer interface are:

- Create\_Timer
- Get\_Time\_Remaining
- Delete\_Timer

The invocation syntax for each of these operations now follows.

**Create\_Timer** [ <My\_Timer\_ID> ] = **Create\_Timer**[ ]

This creates a timer object which can then be set, reset or deleted by using the returned timer ID as a reference.

Where:

<My_Timer_ID>	is the id of the created timer (of type Timer_ID).
---------------	--

**Get\_Time\_Remaining** [ <still\_to\_go> ] = **Get\_Time\_Remaining**[ <My\_Timer\_ID> ]

This reads the amount of time remaining before the specified timer is due to expire. The result will be zero if the timer specified is not running.

Where:

<still_to_go>	is an integer indicating the time remaining (in units specified when TIM1 was sent).
<My_Timer_Id>	is the id of the timer who remaining time that is being queried.

**Delete\_Timer** [ ] = **Delete\_Timer**[ <My\_Timer\_ID> ]

This deletes the specified timer. This should only be performed after the timer has been reset.

Where:

<My_Timer_ID>	is the id of the timer that is to be deleted.
---------------	---



## Signals Consumed By the xUML Timer

There are 3 signals that can be directed at the xUML Timer each of which carry one or more signal parameters:

- TIM1:Set\_Timer
- TIM2:Reset\_Timer
- TIM10:Set\_Absolute\_Timer

The syntax for generating such signals is defined below:

```
TIM1: generate TIM1:Set_Timer( <Timer_ID>, \
                               <Time_Remaining>, \
                               <Granularity>, \
                               <Signal>, \
                               <Return Instance Handle> )
```

This puts the timer into a state where it is counting down until the supplied Time\_Remaining has expired. At that point the timer will send the specified signal to the object specified by <Return Instance Handle>.

Where:

<Timer_ID>	is the id of the timer that is to be set.
<Time_Remaining>	is an integer indicating the time-out time in units indicated by Granularity
<Granularity>	is an enumeration specifying the units for this timer. The enumeration values defined in ASL are 'MICROSECOND', 'MILLISECOND', 'SECOND', 'MINUTE', 'HOUR' and 'DAY'.
<Signal>	is the return signal to be sent when the timer expires. A directive “ <b>signal</b> ” is supplied for the purpose and is described in “ <a href="#">Signal Directive:</a> ” on page 68.
<Return Instance Handle>	is a handle on the object to which the signal must be returned. This is of type “Instance Handle”, which is a special “untyped” instance handle. This type cannot be used for any other purpose in a domain.

```
TIM2: generate TIM2:Reset_Timer( <Timer_ID> )
```

Causes the timer to enter a reset state where it is no longer counting down. If the timer was previously counting down, the requested return signal will not be sent when the timer is reset.

Where:

<Timer_ID>	is the id of the timer that is to be reset.
------------	---

```
TIM10: generate TIM10:Set_Absolute_Timer(<Timer_ID>, \
                                         <Date>, \
                                         <Time>, \
                                         <Signal>, \
                                         <Return Instance Handle>)
```

This puts the timer into a state where it is counting down until the supplied Date and Time is reached. At that point the timer will send the specified signal to the object specified by <Return Instance Handle>.

Where:

<Timer_ID	is the id of the timer that is to be set.
<Date>	is the date on which the timer is to expire (type Date)
<Time>	is the time at which the timer is to expire (type Time_of_Day)
<Signal>	is the return signal to be sent when the timer expires. A directive “ <b>signal</b> ” is supplied for the purpose and is described in “ <b>Signal Directive:</b> ” below.
<Return Instance Handle>	is a handle on the object to which the signal must be returned. This is of type “Instance handle”, which is a special “untyped” instance handle. This type cannot be used for any other purpose in a domain.

Note: An absolute timer will return time remaining in SECOND granularity.

```
Signal Directive: event("<signal label>")
```

Where:

<signal label>	defines the signal that is to be generated when the respective timer expires, and is of the form:  <class keyletter><signal number>
----------------	---

Note: This construct turns a string literal containing a signal label and returns an ASL meta-type, namely Signal Event. The construct can only be used in these timer signals.

## Examples of ASL that make use of the xUML Timer

The following examples provide an illustration of how the various xUML Timer operations and signals may be used within an executable UML model.

### Example 1:

```
# Create a new timer
[timer_id] = Create_Timer[]
```

### Example 2:

```
# Set a timer to return the signal "D7:Time to do something" to
# ourselves in 5 seconds from now
timeout = 5
generate TIM1:Set_Timer(timer_id,timeout,'SECOND',event("D7"),this)
```

### Example 3:

```
# Set a timer to return the signal "D8:Time to Wake Up" to ourselves
Actual_Date = 1994.10.19
Actual_Time = 07:00:00
generate TIM10:Set_Absolute_Timer( timer_id, \
                                   Actual_Date, \
                                   Actual_Time, \
                                   signal("D8"),\
                                   this)
```

### Example 4:

```
# To reset the timer
generate TIM2:Reset_Timer(timer_id)
```

### Example 5:

```
# To find the time remaining
[still_to_go] = Get_Time_Remaining[timer_id]
# To delete the timer
[] = Delete_Timer[timer_id]
```

## 12.2 Current Date and Time

Access to the current date and time may be achieved as follows:

Syntax:

```
<local variable> = current-date
```

```
<local variable> = current-time
```

- Notes:
1. The data types returned are "Date" and "Time\_of\_Day" respectively.
  2. "**current-date**" and "**current-time**" are ASL keywords.
  3. There are no ASL defined operations on time other than the logical operators comparison operators.
  4. "**current-date**" and "**current-time**" cannot be used directly in expressions.

## 13 Complex Datatypes and Sets

Until now we have considered rather simple data items<sup>43</sup> within a ASL. These cover the majority of requirements. However, in some situations it may be necessary to transmit signal parameters that are complex structures. This typically happens in cross-domain signals where the pattern of information in one domain may be radically different to that in another.

For example, we may wish to assemble a message for the operator in the form of a small tabular report that contains information gathered from many objects of many classes in the application domain.

### 13.1 Supported Structures


ASL supports hierarchical data structures. In principle these can be nested to any depth, but in practice one would expect limited depth within an xUML model.

These structures can then be used to support complex signal parameters and local variables in ASL. As we have already stated the use of structures as the data type of an attribute of a class is *specifically excluded*.



<sup>43</sup>.We have dealt with only single values of any type and sets of instance handles.

## 13.2 Definition of Structures

Data structures are defined as follows<sup>44</sup> :

```

define structure <structure type name>
    <member name> <member type>
    <member name> <member type>
    ...
    ...
enddefine
    
```

Where:

<structure type name>	is the name that will be used when defining instances of the structure using the “is” construct (see later)
<member name>	is the name of a component or member of the structure
<member type>	is any valid ASL type, including structure types

- Notes:**
1. Any member of the structure can be a set, by use of the { } notation. For example:

```

define structure my_type
    number_of_robots Integer
    {robots} Robot
enddefine
    
```

This defines a structure that contains two components, a single integer and a set of handles for the class “Robot”.

2. These structure definitions are not shown with the ASL associated with an action. Rather they are (in language terms) defined as part of a global type definition for a domain.
3. Recursive data structures are not permitted. (i.e. a structure cannot be defined to contain a member, at any level of nesting, of the same type as the structure being defined).

**Examples:** # Two structures (one nested in the other) necessary for the example # given in the previous section.  
 # The “inner” structure

```

define structure disk_request_record
    qp_id Integer
    time_of_request Time_of_Day
enddefine
    
```

# The “outer” structure

```

define structure request_list
    disk_id Integer
    {requests} disk_request_record
enddefine
    
```



44. In iUML, structure definitions are held in the database and manipulated by menu operation, and the analyst is never required to write out the syntax described here.

## 13.3 Instantiation of Structures

So far in most of ASL there have not been any explicit type declaration for local data items. This is possible because the types on all data items “external” to an action (attributes and signal parameters) are explicitly defined in the xUML model. This allows the type of data items within an action to be determined from their first use, and this relieves the analysts from the burden of specifying the type of every item.

However, with sets of complex structures, explicit typing is necessary. This is indicated by use of the “**is**” statement:

```
{<local structure>} is <data type>
```

Where:

{<local structure>}	is the local instance of a set of structures being declared
<data type>	is any valid structure type

This creates an empty set of the appropriate type.

- Notes:**
1. The “**is**” statement must appear in an ASL segment before any use of the set of structures (for example in an **append** statement).
  2. A single “**is**” statement (for example: “x **is** request\_list”) may be executed many times within an action. The “**is**” statement is not simply a declaration but also *initialises* the set to be empty. Any data in the set before the “**is**” is executed will be lost.
  3. ASL does not allow instantiation of a single structure. Structures must always exist in sets.

**Example:** # Declare a set of request\_lists (as defined in  
# the previous section).

```
{requests} is request_list
```

## 13.4 Assembly of Sets of Structures

Having declared a set of structures, members can be added using the `append` statement:

```
append [<value list>] to {<local structure>}
```

Where:

{<local structure>}	is a previously declared set of structures
[<value list>]	is a comma separated list of local variables to be assigned to members

- Notes:**
1. Matching of structure members to values is achieved on the basis of *position* within the <value list> against position within the original structure definition.
  2. When one set is appended as a member of another (as in the example below), then the contents of the set being appended are effectively *copied* into the set being appended to. This means that any subsequent change to the set that was appended (the next iteration in the outer loop below), will not affect the member of the outer set to which it had just been appended.

**Examples:**

```
# Set up a disk request list and send to operator for display.
# First, declare the list to be sent.
{request_table} is request_list

# Then, loop over all the disks, building up outer structure
{disks} = find-all Disk
for disk in {disks} do
  {requests} = disk -> R1.Disk_Request
  {disk_table} is disk_request_record
  # Loop round requests for this disk, building up the inner structure
  for request in {requests} do
    append [ request.QP_Id,request.Time_Request_Made ] to {disk_table}
  endfor
  append [ disk.Disk_Id ,{disk_table} ] to {request_table}
endfor

generate EU11:displayRequestTable( {request_table} )
```



## 13.5 Use of Loops to Perform Unpacking of Set Structures

Typically, such sets of structures will also be received by actions in the form of signal parameters or input parameters to operations. To unpack these the loop construct must be used.

For example, an action receiving the {request\_table} created above could perform:

```
for [ disk_id , {disk_table} ] in {request_table} do  
  x = disk_id  
  ...  
  ...  
  for [ qp_id , time ] in {disk_table} do  
    y = qp_id  
    z = time  
    ...  
    ...  
    w = disk_id  
    # (i.e. disk_id is still in scope from the outer loop)  
  endfor  
endfor
```

## 13.6 Ordering of Sets of Structures

By default, sets of structures have no ordering. However, they can be explicitly ordered by the following techniques:

```
append [<value list>] to {<set>} ordered by <member>
```

```
{<new set>} = {<old set>} ordered by <member>
```

Where:


<member>	is the <i>name</i> of the member of the structure
----------	---

- Notes:**
1. The meaning of ordering is as defined in the earlier section on “Ordering of Instance handles”.
  2. Reverse ordering may be achieved by using “**reverse ordered by**”
  3. Ordering may be achieved on multiple members<sup>45</sup> by use of “&”, for example:
  4. {new\_list} = {old\_list} ordered by name & address
  5. The resulting set will be primarily sorted by name, and within each name by address.
  6. Appending to the same set with different ordering in **append** different statements will cause unpredictable results.



45.iUML Simulator does not support ordering by multiple structure members.

## 13.7 Subsets of Sets of Structures

The contents of sets can be reduced by making subsets<sup>46</sup> :

`{<new set>} = {<old set>} where <condition>`

Where:

<code>{&lt;new set&gt;}</code>	is a set of the same type as <code>{&lt;old set&gt;}</code> , each member of which is a member of <code>{&lt;old set&gt;}</code> and satisfies <code>&lt;condition&gt;</code>
<code>{&lt;old set&gt;}</code>	is set of structures.
<code>&lt;condition&gt;</code>	is a logical condition based on members of <code>{&lt;old set&gt;}</code>

- Notes:**
1. This operation effectively makes a copy of the elements of `{<old set>}` so that any subsequent change to `{<old set>}` will not affect the contents of `{<new set>}`.
  2. Whether or not the data is physically copied by this operation is architecture dependent. It is easy to envisage run time data structures that would achieve these effects simply by maintaining masks and indices on the original set.
  3. Note that this subsetting of sets of structures is similar to one form of the “find” operation for sets of instance handles.

**Example:** `{old_requests} = {disk_table} where time_request_made < 12:00:00`



<sup>46</sup>iUML Simulator does not support subsetting of structures.



## 14 Sets, Sequences and Bags

Until now we have talked rather loosely about “sets”, without regard to their precise mathematical definition. In fact, “sets” in ASL are capable of being sets, sequences or bags. A newly created ASL set is likely to be a “bag” or a “sequence”. That is it can have duplicates and can be created with a defined order. (Duplicates are likely to appear for example in a multiple navigation involving two “many” association ends, back-to-back on one class in a chain.)

In what follows, the “sets” referred to can equally well be sets of handles or sets of structures<sup>47</sup>.

### 14.1 Equality

Implicit in all of the set operations is the concept of “Equality” of members of a set. For example the Unique operation removes duplicate members from the set. Duplicates are members that are, in some sense, equal.

ASL defines equality of members as follows:

- For instance handle sets, two members are identical if they refer to the same object.
- For structure sets, two element are identical if all of the members of the first element are identical to the corresponding element in the other member. The correspondence between members is determined by the order of the members in the definition of the structure. This means that equality must be defined for all the member types.

For structures containing structures, the definition of equality applies recursively.



<sup>47</sup>.iUML Simulator does not support the set operations in this section for sets of structures. However, the operations are fully supported for sets of instance handles.

## 14.2 The Unique Operation

Duplicates may be removed from sets using the “unique” construct:

```
{<new set>} = unique {<old set>}
```

This has the effect of removing any identical elements in {<old set>}.

- Notes:**
1. See Section 14.1 on page 79 for the definition of the meaning of elements being “identical”.

## 14.3 The countof Operation

The **countof** operation is available for all set types.

- Notes:**
1. in the case of a set containing a set the operation will return the number of elements in the outer set.
  2. if the set contains duplicates (is a bag for example), then the countof operation will count each duplication of the member.
  3. the countof operation cannot be applied to the name of a Class in order to determine the number of objects belonging to the Class. Instead the following example shows the ASL that is required:

```
{all_robots} = find-all Robot
number_of_robots = countof {all_robots}
```

## 14.4 Set Combination Operations

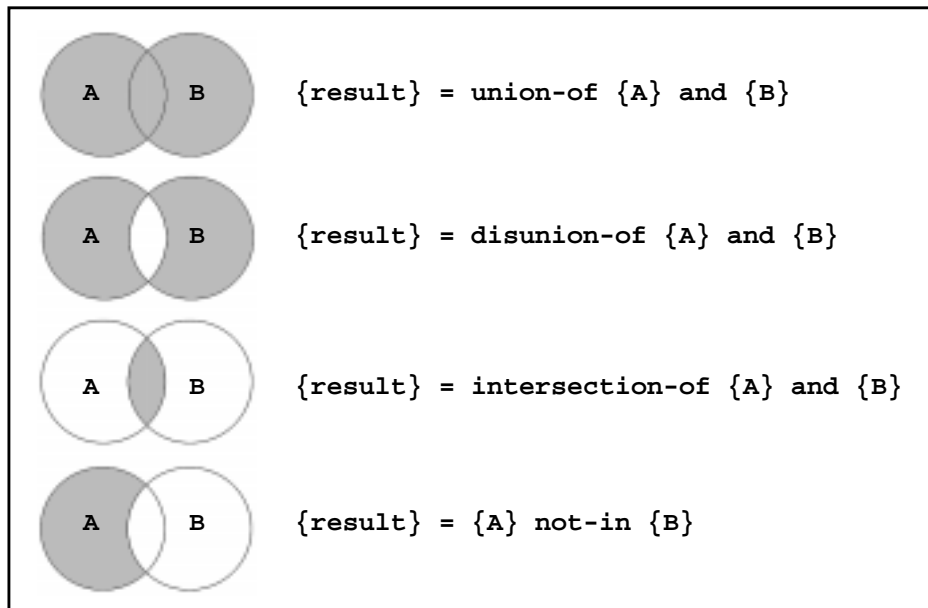
A new set can be created from two old sets, by various manipulations:<sup>48</sup>

```
{<new set>} = union-of {<old set a>} and {<old set b>}
{<new set>} = disunion-of {<old set a>} and {<old set b>}
{<new set>} = intersection-of {<old set a>} and {<old set b>}
{<new set>} = {<old set a>} not-in {<old set a>}
```

These are best explained by use of the Venn diagrams in Figure 3 on page 81:



48.iUML Simulator only supports the set operations described in this section for sets of instance handles.



**Figure 3: Illustration of Set Combination Operations**

- Notes:**
1. The results of these operations are true sets in that there will be no duplicates and no defined ordering in the resulting set, irrespective of the condition of the original sets.
  2. All sets being combined must be of the same type, and the resultant set will also be of that type.





## 15 ASL for Bridge Operations

So far we have discussed ASL in the context of states or operations provided by domains, classes or objects. In this section we discuss the use of ASL to express bridge mappings between domains.

The capabilities and features described here apply to any section of bridge ASL. The intention in this document is to provide a reference for the ASL syntax and semantics only<sup>49</sup>. For information on strategies for using bridges please refer to documentation on the Executable UML method.

### 15.1 Basic Concepts and Terminology

In the Executable UML method, within a given domain, “Terminators” represent abstractions of external entities expressed in the subject matter of the domain. Since the actual physical connection to the external entity will be through another domain, a “Terminator” represent the connection to other domains.

For example, imagine a domain concerned with the control of an industrial robot. Such a domain will have a number of classes capturing concepts in in the problem domain of robot control.

At certain points, the ASL in the robot control domain will require to invoke actual operations on the external robot hardware such as “extend\_arm”. In the robot control domain model, we would see such an operation being invoked on the terminator called, for example, “Robot Hardware”.

Of course, in order to implement this operation, an appropriate electrical signal must be generated in the appropriate piece of hardware. In our imagined system this would be the job of the “Process I/O” (PIO) domain. The PIO domain would understand the subject matter of boards, registers and control words. The “Robot Hardware” terminator thus also represents the interface to the PIO domain as seen in the robot control domain.

In order to make the system work, there will be a segment of ASL that implements the “extend\_arm” operation. That segment of ASL will specify that the operations



<sup>49</sup>Note that in iUML the case tool requires user to write only some of the ASL required for a typical bridge. The tool generates the surrounding ASL automatically. Please refer to the iUML documentation for more details.

is implemented by invoking a particular operation provided by the PIO domain. We call such a segment of ASL a “bridge” since it forms the connection between two or more domains.

In summary then, we have:

Terminator	an abstraction of an external entity and, therefore, an interface to another domain. Technically, a UML Interface Class.
Terminator Operation	an operation provided by a terminator
Bridge	a segment of ASL that implements a terminator operation

## 15.2 Domain Scope Within a Bridge

Since a bridge provides a mapping between two (or more) domains, the scope of the ASL used to describe it is no longer that of a single domain. This creates the possibility of ambiguity in the ASL. For example, an operation name or signal specification can be the same in two different domains in the same system so there must be some way of resolving any ambiguity. This is achieved by the “\$USE directive:

```
$USE <domain_1_keyletter>
  <ASL statements requiring visibility of domain_1>
$ENDUSE
$USE <domain_2_keyletter>
  <ASL statements requiring visibility of domain_2>
$ENDUSE
```

- Notes:
1. the “\$USE” directives are not really ASL statements as such. Rather they direct a translator and indicate which domain is to be used to interpret the ASL statements that follow. As a result, “\$USE” statements cannot be nested.
  2. when no “\$USE” is currently in effect, the context is that of the calling or invoking domain.
  3. the “\$USE” is required to scope items where there is no other means of knowing the scope of a name. Thus ...

```
$USE DOM_B
  slot = my_robot -> R1
$ENDUSE
```

... is OK even if R1 is not in the scope of DOM\_B. This is because, the type of “my\_robot” is already known and so the scope of R1 is inferred. However ...

```
my_robot = find-one Robot
```

... requires an explicit \$USE for the domain containing the class “Robot” since there is no other way of knowing to which domain “Robot” belongs.

## 15.3 Type Mixing in Bridges

As indicated in section Type Mixing Rules (“Type Mixing Rules”), the normal rules of ASL are relaxed in bridges. The effect of this is to allow assignments and arithmetic expressions that mix types from different domains as long as the base types involved are commensurate.

This feature supports the “semantic shift” that appears in bridges.

## 15.4 Types of Bridge

In ASL 2.4 and earlier there were only two types of bridge that were supported. These were:

- Signal to a terminator
- Operation invoked on a terminator

The distinction between the two is unhelpful and our intention is, ultimately, to remove support for signals to terminators and leave a single explicit bridge invocation mechanism:

- The terminator operation call

In addition in ASL 2.5 there are two flavours of terminator operation:

- Non-object scoped terminator operation
- Object scoped terminator operation

The former of these is exactly equivalent to the ASL 2.4 terminator operation. The object scoped terminator is used for interactions between counterpart objects in different domains, and the corresponding bridge ASL has access to the keyword “this”.

The following sections describe the characteristics of the bridges that implement these terminator operations as well as showing how actual counterpart manipulation is achieved.

## 15.5 Definition and Invocation of Non-Object Scoped Bridges

The non-object scoped bridge definition and invocation syntax is:

### Definition Syntax:

```
define bridge <invoking domain kl>:<terminator kl><operation number>_<operation name>
input <in param 1>:<in param 1 type>, ...
output <out param 1>:<out param 1 type>, ...
  <asl statements in the context of invoking domain>
  $USE <domain_1_keyletter>
    <ASL statements requiring visibility of domain_1>
  $ENDUSE
  $USE <domain_2_keyletter>
    <ASL statements requiring visibility of domain_2>
  $ENDUSE
  ...
enddefine
```

### Invocation Syntax:

```
[<out param 1>, ...] = <terminator kl><operation number>:<operation name>[<in param 1>, ...]
```

Where:

<invoking domain kl>	is a convenient short form of the name of the calling domain. This name must be defined within the context of the system into which this domain is assembled, and must be unique within that context.
<terminator kl>	is the key letter of the terminator which provides the bridge operation in the calling domain.
<operation number>	is the number of the operation within the terminator that provides it in the calling domain.
<operation name>	is the name of the operation provided by the terminator in the calling domain.
<in param 1> etc.	is an input parameter for the bridge operation. Input parameters for bridge operations have exactly the same syntax and rules as for a regular ASL operation definition <sup>a</sup> .
<out param 1> etc.	is an output parameter for the bridge operation. Output parameters for bridge operations have exactly the same syntax and rules as for a regular ASL operation definition <sup>a</sup> .
<domain_1_keyletter> etc.	are the short form names for the domains on which operations are to be invoked.

a. In iUML Simulator there are limitations as to the use that can be made of a bridge input parameter within the bridge. See release notes for details.

- Notes:**
1. Any ASL statement within the bridge may set the values of the output parameters.
  2. The operation number must be unique within the context of the terminator providing the operation in the calling domain
  3. The operation name must be unique across all the operations provided by the terminator in the calling domain.

```
Example: # State ASL invokes bridge operation provided by the
# Robot Hardware (RHW) terminator
[arm_status] = RHW1:getArmStatus[robot_id]

# Bridge mapping definition made in the context of a build that
# uses the PIO domain to implement the operation
define bridge ODMS:RHW1_getArmStatus
input robot_id:Integer
output status:Arm_Status
$USE PIO
    # Calculate mapping to correct register
    registerNumber = (robot_id * 7) + 3

    # Invoke domain scoped operation provided by the PIO domain
    [registerValue] = PIO3::getRegisterValue[registerNumber]

    # Map result back
    if registerValue = 0 then
        status = 'Retracted'
    else
        status = 'Extended'
    endif
$ENDUSE

enddefine
```

## 15.6 Definition and Invocation of Object Scoped Bridges

An object scoped bridge is one that implements an operation provided by a terminator that represents the counterpart of a class in the calling domain. Such terminators are thus always associated with a specific class in the calling domain.

The object scoped bridge definition and invocation syntax is:

### Definition Syntax:

```
define instance bridge <invoking domain kl>:<terminator kl><operation number>_<operation name>
instance this:<class name>
input <in param 1>:<in param 1 type>, <in param 2>:<in param 2 type>, ...
output <out param 1>:<out param 1 type>, ...
  <asl statements in the context of invoking domain>
  $USE <domain_1_keyletter>
    <ASL statements requiring visibility of domain_1>
  $ENDUSE
  $USE <domain_2_keyletter >
    <ASL statements requiring visibility of domain_2>
  $ENDUSE
enddefine
```

### Invocation Syntax:

```
[<out param 1>, ...] = <terminator kl><operation number>:<operation name>[<in param 1>, \
                                                                <in param 2> \
                                                                ... ] on counterpart
```

Where:

<invoking domain KL>	is a convenient short form of the name of the calling domain. This name must be defined within the context of the system into which this domain is assembled, and must be unique within that context.
<terminator KL>	is the key letter of the terminator which provides the bridge operation in the calling domain.
<operation number>	is the number of the operation within the terminator that provides it in the calling domain.
<operation name>	is the name of the terminator operation as seen by the calling domain.
<class name>	is the name of the class with which the counterpart terminator is associated in the calling domain
<in param 1> etc.	is an input parameter for the bridge operation. Input parameters for bridge operations have exactly the same syntax and rules as for a regular ASL operation definition <sup>a</sup> .
<out param 1> etc.	is an output parameter for the bridge operation. Output parameters for bridge operations have exactly the same syntax and rules as for a regular ASL operation definition <sup>a</sup> .
<domain_1_name> etc.	are the short form names for the domains on which operations are to be invoked.

<sup>a</sup>In iUML Simulator there are limitations as to the use that can be made of a bridge input parameter within the bridge. See release notes for details.

- Notes:
1. Any ASL statement within the bridge may use the handle **this**.
  2. "**counterpart**" is an ASL keyword. The interpretation of the line of as invoking the bridge is that it is being invoked on "the counterpart of **this**". It is the use of the "**on counterpart**" in the invocation that provides the value of "**this**" in the bridge.
  3. The "**on counterpart**" clause can only be used in a context where **this** is valid and where this refers to a class which partakes in a counterpart association.
  4. The "**on counterpart**" can be replaced by "**on <instance handle>**" in the creation state of the class which is to have the counterpart. This allows the counterpart operation that is called to set up the counterpart association. In this case "<instance handle>" must be of the correct type.
  5. The operation number must be unique within the context of the terminator providing the operation in the calling domain.
  6. The operation name must be unique across all the operations provided by the terminator in the calling domain.
  7. Any ASL statement within the bridge may set the values of the output parameters<sup>50</sup>.

In the previous example in Section 15.5 on page 87, the correspondence between the object of class robot (as indicated by "robot\_id" and the appropriate register was captured in the line:

```
registerNumber = (robot_id * 7) + 3
```

In the following example, this bridge has been enhanced to use a counterpart association (CPR1) to provide this mapping. Counterpart associations are covered in more detail in a later section.

**Example:**

```
# State action invokes bridge operation provided by the
# Robot Hardware (RHW) terminator
[arm_status] = RHW1:get_arm_status[robot_id] on counterpart

# Bridge mapping definition made in the context of a build that
# uses the PIO domain to implement the operation
define instance bridge ODMS:RHW1_get_arm_status
instancethis:Robot
input
output status:Arm_Status
$USE PIO
  # Navigate to correct register using the counterpart association
  register = this -> CPR1
  # Invoke object scoped operation provided by Register in PIO domain.
  [register_value] = REG3:get_register_value[] on register
  # Map result back
  if register_value = 0 then
    status = 'Retracted'
  else
    status = 'Extended'
  endif
$ENDUSE
enddefine
```



50. In iUML Simulator the results of all operations and calculations in the bridge should be assigned to local variables. Such variables should then be assigned to the return parameters.



## 15.7 Counterpart Relationship Manipulation

Object scoped bridges may create, delete and navigate instances of counterpart relationships. There are two types of counterpart relationship:

- Counterpart Generalisations
- Counterpart Associations

The context that these are used in is described more fully in Kennedy Carter's xUML method documentation. In particular it should be noted that, particularly in the context of counterpart generalisations, the analyst will not actually need to write much of the relationship manipulation since this can be generated automatically by a CASE tool that understands the formalism.

However, this section fully describes the syntax and direct use of the relationship manipulation.

- Notes:**
1. All counterpart relationship manipulations take place within a defined domain context of an object scoped bridge. This context is either (by default) that of the calling domain, or it is an explicit context set by a `$USE` clause.
  2. Counterpart relationship navigations cannot be placed in a navigation chain.
  3. Since counterpart relationships are much more restricted than normal associations and generalisations (no association classes, reflexive associations or navigation chains), the following sections show each case individually, rather than generically as was done in Section 8, "Association and Generalisation," on page 31.

## Counterpart Generalisations

### Navigating a Counterpart

**Generalisation:** The syntax for navigation of a counterpart generalisation is as follows:

```
<destination handle> = <starting handle> -> CPR<n>.<class name>
```

Where:

<destination handle>	is the instance handle of the object obtained as a result of the navigation
<starting handle>	is the instance handle of the object that is the source of the navigation
CPR<n>	is the counterpart generalisation that is being navigated (from <starting handle> resulting in <destination handle>)
<class name>	is the name of the class that is the destination of the navigation

- Notes:**
1. <class name> is mandatory when navigating from a “generic” object to the related “specific” object, even if there is only one specific class defined for the counterpart generalisation.
  2. <class name> is optional when navigating from a “specific” object to the its related “generic” object.
  3. If <starting handle> is undefined when the navigation is executed then this is considered as a run time error.

### Creating an Instance of a Counterpart

**Generalisation:** The syntax for creating an instance of a counterpart generalisation is as follows:

```
link-counterpart <source handle> CPR<n>.<class name> <destination handle>
```

Where:

<source handle>	is the single instance handle of the first object to be linked via the specified counterpart generalisation.
<destination handle>	is the single instance handle of the second object to be linked via the specified counterpart generalisation.
CPR<n>	is the counterpart generalisation that is to be created (linked) between the <source handle> and the <destination handle>.
<class name>	is the name of the destination class and is optional. <sup>a</sup>

a. iUML Simulator does not accept this optional clause

- Notes:**
1. The terms “source” and “destination” are used for the purposes of explanation only. As with all associations, counterpart associations are “two way” and can be navigated from either end.
  2. the <class name> clause is optional.
  3. If either instance handle is undefined when the statement is executed, then this is considered to be a run time error.
  4. If an attempt is made to link the same two objects together via the same relationship twice then this is regarded as a run time error.

## Deleting an Instance of a Counterpart Generalisation:

The syntax for deleting an instance of a counterpart generalisation is as follows:

```
unlink-counterpart <source handle> CPR<n>.<class name> <destination handle>
```

Where:

<source handle>	is the single instance handle of the first object to be unlinked via the specified counterpart generalisation.
<destination handle>	is the single instance handle of the second object to be unlinked via the specified counterpart generalisation.
<b>CPR</b> <n>	is the counterpart generalisation that is to be deleted (unlinked) between the <source handle> and the <destination handle>.
<class name>	is the name of the destination class and is optional <sup>a</sup>

a. iUML Simulator does not accept this optional clause

- Notes:**
1. The terms “source” and “destination” are used for the purposes of explanation only. As with all relationships in xUML, counterpart relationships are “two way” and can be navigated from either end.
  2. The <class name> clause is optional.
  3. If the two objects are not linked, or if either instance handle is undefined when the statement is executed, then this is considered to be a run time error.

## Counterpart Associations

### Navigating a Counterpart

**Association:** The syntax for navigation of a counterpart association is as follows:

```
<destination handle> = <starting handle> -> CPR<n>."<role phrase>".<class name>
```

or:

```
{<destination handle set>} = <starting handle> -> CPR<n>."<role phrase>".<class name>
```

Where:

<destination handle>	is the instance handle of the object obtained as a result of the navigation (See Note 3 below).
{<destination handle set>}	is the instance handle set for the objects obtained as a result of the navigation (See Note 3 below).
<starting handle>	is the instance handle of the object that is the source of the navigation
CPR<n>	is the counterpart association that is being navigated (from <starting handle> resulting in <destination handle> or {<destination handle set>}).
<class name>	is the name of the class that is the destination of the navigation and is optional
<role phrase>	is the role phrase appropriate to the destination class and is optional

- Notes:**
1. <class name> and <role phrase> are optional<sup>51</sup>
  2. If the <starting handle> is undefined when the navigation is executed then this is considered as a run time error.
  3. The navigation will return a set if the multiplicity of the counterpart association is many valued in the direction of navigation and is single valued otherwise.



51.iUML Simulator does not accept the presence of role phrases in counterpart association manipulations.

## Creating an Instance of a Counterpart Association:

The syntax for creating an instance of a counterpart association is as follows:

```
link-counterpart <source handle> CPR<n>."<role phrase>".<class name> <destination handle>
```

Where:

<source handle>	is the single instance handle of the first object to be linked via the specified counterpart association.
<destination handle>	is the single instance handle of the second object to be linked via the specified counterpart association.
<b>CPR</b> <n>	is the counterpart association that is to be created (linked) between the <source handle> and the <destination handle>.
<class name>	is the name of the destination class and is optional <sup>a</sup>
<role phrase>	is the role phrase appropriate to navigating from the source to the destination class and is optional

a. iUML Simulator does not accept this optional clause

- Notes:**
1. The terms “source” and “destination” are used for the purposes of explanation only. As with all associations, counterpart associations are “two way” and can be navigated from either end.
  2. the <class name> and <role phrase> clauses are optional.
  3. If either instance handle is undefined when the statement is executed, then this is considered to be a run time error.
  4. If an attempt is made to link the same two objects together via the same association twice then this is regarded as a run time error.

### Deleting an Instance of a Counterpart Association:

The syntax for deleting an instance of a counterpart association is as follows:

**unlink-counterpart** <source handle> **CPR**<n>."**<rolePhrase>**".<class name> <destination handle>

Where:

<source handle>	is the single instance handle of the first object to be unlinked via the specified counterpart association.
<destination handle>	is the single instance handle of the second object to be unlinked via the specified counterpart association.
<b>CPR</b> <n>	is the counterpart association that is to be deleted (unlinked) between the <source handle> and the <destination handle>.
<class name>	is the name of the destination class and is optional <sup>a</sup>
<rolePhrase>	is the role phrase appropriate to navigating from the source to the destination class and is optional

a. iUML Simulator does not accept this optional clause

- Notes:**
1. The terms "source" and "destination" are used for the purposes of explanation only. As with all associations, counterpart associations are "two way" and can be navigated from either end.
  2. The <class name> and <role phrase> clauses are optional.
  3. If the two objects are not linked, or if either instance handle is undefined when the statement is executed, then this is considered to be a run time error.

## 15.8 Definition & Invocation for a Signal Bridge

As indicated at the start of this section on bridges, signals to terminators should be avoided in favour of terminator operations. However, ASL 2.5 still supports the older construct. A bridge that implements a signal received by a terminator in the invoking domain is defined in ASL as follows:

### Definition Syntax:

```
define bridge <invoking domain keyletter>:<terminator signal name>
input <in param1>:<in param 1 type>, ....
output
<asl statements in the context of invoking domain>
$USE <domain_1_keyletter>
    <ASL statements requiring visibility of domain_1>
$ENDUSE
$USE <domain_2_keyletter>
    <ASL statements requiring visibility of domain_2>
$ENDUSE
...
enddefine
```

Where:

<invoking domain keyletter>	is a convenient short form of the name of the calling domain. This name must be defined within the context of the system into which this domain is assembled, and must be unique within that context.
<terminator signal name>	is the name of the signal as seen by the generating domain. Note that the “:” in the signal label becomes a “_” character (see example below).
<in param 1> etc.	refer to the signal parameters and have exactly the same syntax and rules as for a regular ASL operation definition.
<domain_1_keyletter> etc.	are the short form names for the domains on which operations are to be invoked.

**Note:** 1. such a bridge may not return output parameters, but the **output** statement must be present.

**Example:**

```
# State action generates terminator signal
generate ROBHW7:extendArm(robot_id)

# Bridge definition
define bridge ODMS:ROBHW7_extendArm
input robot_id:Integer
output
    $USE PIO
        # Calculate mapping to correct register
        registerNumber = (robot_id * 7) + 1

        # Invoke domain scoped operation provided by PIO domain
        [] = PIO4::setRegisterValue[registerNumber , 1]
    $ENDUSE
enddefine
```





## 16 Native Language Inserts

### The use of \$INLINE

ASL allows analysts to insert sections of native language code into the body of the ASL.

Syntax:

```
$INLINE  
    <native language statements>  
$ENDINLINE
```

The precise effect of this is architecture dependant, but will involve the ASL translator simply dropping the native language statements into the generated code without modification. Clearly the nature of the translation mapping must be understood before such native statements can be written. As a result the statements will be highly unstable against changes in the architecture and so such sections should be used with great caution.



## 17 Appendix A: Requirements for an ASL

The purpose of analysis is to understand a problem domain and to specify behaviour that will meet a set of requirements. To this end the analyst describes a system in terms of Classes and Interacting State machines with well defined processing actions. The xUML models produced should be sufficiently detailed and precise that they are capable of being “tested” against external criteria (such as external reality or desired system behaviour) and as such these models should, in principle, be executable.

The outline requirements that have guided us in the definition of ASL are:

- The ASL must be detailed and precise enough that the resulting models can be executed without any ambiguity or use of assumptions. Of necessity, this will result in an ASL that has the appearance of a programming language, although not necessarily an algorithmic one<sup>52</sup>.
- The ASL must be rich enough to specify all the processing that will be required. If the language is not sufficiently versatile, then analysts will be forced to resort to other, perhaps vague, forms of expression.
- The ASL must be readily readable. It is one thing to create a language that is precise, it is another to produce one that can be quickly and easily scanned by the human reader. There must always be a place for reviews and other such procedures, and the ASL must not hinder that activity.
- The ASL must be simple, and rapid to create. Although a similar problem to the previous requirement, different issues come into play, such as the need to avoid long complex keywords, or a wealth of different special characters.
- The ASL must be sufficiently rich that automated execution of architectural mappings becomes feasible. For example, in order to support “instance handle” based architectures, it must be possible to recognise association manipulations clearly and unambiguously.

These requirements have implicit contradictions, for example:

- Short, easy to type keywords can produce cryptic names



<sup>52</sup>It should be noted, however, that we are not advocating the inclusion of “design” ideas in an analysis model. The ASL must allow the analyst to specify processing without assuming any particular computational or other solution. Such detail will be the subject of another domain.

- Many analysts prefer to think algorithmically. However, while an algorithmic specification may be correct, there may be other, more appropriate algorithms that can be used in the implementation.

In defining this ASL, we have attempted to form the best possible compromise between the competing requirements.

## 18 Appendix B: The Keywords of ASL

The following keywords are reserved in ASL and must not be used as the name of any Class, terminator, Domain or Attribute, or data item.

ALREADY_DEFINED	delete	link-counterpart
Boolean	disunion-of	loop
Date	do	not
ERROR	else	not-equals
FALSE	enddefine	not-in
Integer	endfor	of
Real	endif	one-of
TRUE	endloop	only
Text	endswitch	or
Time_of_Day	enduse	ordered
UNDEFINED	equals	output
and	event	reverse
and	find	structure
append	find-all	switch
associate	find-one	then
break	find-only	this
breakif	for	to
bridge	function	unassociate
by	generate	union-of
case	greater-than	unique
countof	greater-than-or-equal-to	unlink
counterpart	if	unlink-counterpart
create	in	use
current-date	input	using
current-time	instance	where
default	intersection-of	with
define	is	



# Index

## Symbols

# comment 7  
\$ENDUSE 85  
\$USE 85  
& 76  
-> 33  
\  
ASL newline 7  
{ } 12  
} 7

## A

ALREADY\_DEFINED 20  
and 43  
append 74

## B

base type 10  
break 16,18  
breakif 16,18  
bridges  
  events to terminators 97  
  instance based 89  
  scope in 85  
  type mixing in 85  
  use of "on counterpart" 89

## C

case 15  
comments 7  
complex structures 71  
counterpart relationships  
  counterpart associations 94  
  linking 95  
  navigation of 94  
  unlinking 96  
  counterpart generalisations  
  linking 92  
  navigation of 92  
  unlinking 93  
  CPR 92,93,94,95,96  
countof 12,53,80  
CPR 92,93,94,95,96  
create 19

Create\_Timer 66  
current\_date 70  
current\_time 70

## D

data items 9  
default 15  
define function 56,58,60,62  
define structure 72  
delete 24  
Delete\_Timer 66  
disunion-of 80  
do 16

## E

else 16  
enddefine 56,58,60,72  
endfor 16  
endif 16  
endloop 18  
endswitch 15  
enduse 85,87,89,97  
ERROR 20  
event 67,68  
execution rules 6  
expressions  
  arithmetic 51  
  logical 53  
external data 6

## F

find 25  
find-all 25  
find-one 26  
find-only 27  
for 16,75  
functions  
  definition 55,56,58,60  
  invocation 57,59,61

## G

generate 47  
Get\_Time\_Remaining 66

## I

if 16  
in 16  
input 56,58,60  
instance  
  creation 19  
intersection-of 80  
is 73

## K

keywords 7  
  summary 103

## L

link 40  
link-counterpart 92,95  
logical conditions 53  
loop 18

## N

names 7  
newline 7  
not-in 80

## O

of 50  
only 12  
Operations 55  
  Class Scoped Operations 58  
  Create\_Timer 66  
  Domain Scoped Operations 56  
  Object Scoped Operations 60  
ordered by 28,76  
output 56,58,60

## R

- relationship specifications 35
  - Qualified Number 35
  - Qualified Role 35
  - Relationship Number 35
  - Relationship Role 35
- reverse ordered by 28,76

## S

- scope 8
- set, definition of 12
- sets 71
- signal generation 47
- structure 5,7
  - statement termination 7
- switch 15
- switch statement 15

## T

- then 16
- this 9,11
- TIM1:Set\_Timer 67
- TIM10:Set\_Absolute\_Timer 68
- TIM2:Reset\_Timer 67
- Time 65
- Timer Operations 65
- timer terminator 65
- to 74
- typing, implicit 11

## U

- UNDEFINED 13
- union-of 80
- unique 19,80
- unlink 41
- unlink-counterpart 93,96
- use 85,87,89,97
- using 40

## W

- where 21,77
- with 19

## X

- xUML Timer 65,66
  - Operations 66
    - Delete\_Timer 66
    - Get\_Time\_Remaining 66
  - Signals 67
    - TIM1:Set\_Timer 67
    - TIM10:Set\_Absolute\_Timer 68
    - TIM2:Reset\_Timer 67