MASL Reference Manual

Version 2.0 (Draft)

Synopsis

This document is the definitive language specification and reference for MASL.

Table of Contents

List of Examples

1. Lexical Structure

1.1. Character Set

The valid character set of a MASL program is the set of printable characters in the ISO/IEC 646-US character set (commonly known as ASCII), plus the tab, form feed, line feed and carriage return control characters.

[1] character set = digit | letter | punctuation character | whitespace character;

[2] digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

[3] letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z';

[4] punctuation character = '_' | '{' | '}' | '[' | ']' | '#' | '(' | ')' | '<' | '>' | '%' | ':' | ';' |'.' | '?' | '*' | '+' | '-' | '/' | '^' | '&' | '|' | '~' | '!' | '=' | ',' | '\' | '"' | "'";

[5] whitespace character = inline whitespace character | end of line character;

[6] inline whitespace character = ? ISO 646 SP character ? | ? ISO 646 HT character ? | ? ISO 646 FF character ?;

[7] end of line character = ? ISO 646 CR character ? | ? ISO 646 LF character ?;

1.2. Tokenization

A MASL program is first lexically analysed to break it down into a series of tokens. Once this tokenization is complete, whitepace and comment tokens are discarded before the syntatic analysis is performed. The following tokens types are defined.

1.2.1. Comments

A MASL comment consists of the character sequence "//" followed by any other characters from the character set up to and including the first end of line character sequence

[8] comment = '/' , '/' , { single line character } , end of line;

[9] single line character = digit | letter | punctuation character | inline whitespace character;

[10] end of line = ? ISO 646 CR character ? |
? ISO 646 LF character ? |
? ISO 646 CR character ? , ? ISO 646 LF character ?;

1.2.2. Whitespace

A whitespace token consists of any number of contiguous whitespace characters

[11] whitespace = whitespace character , { whitespace character };

[5] whitespace character = inline whitespace character | end of line character;

[6] inline whitespace character = ? ISO 646 SP character ? |
? ISO 646 HT character ? |
? ISO 646 FF character ?;

[7] end of line character = ? ISO 646 CR character ? |
? ISO 646 LF character ?;

1.2.3. Reserved Words

MASL reserves a number of character sequences to help define the syntactic structure. These reserved words may not be used as identifiers.

[12] reserved word = 'Cannot_Happen' | 'Current_State' | 'Ignore' |
'Non_Existent' | 'anonymous' | 'array' | 'assigner' |
'at' | 'bag' | 'begin' | 'cancel' | 'case' | 'conditionally' |
'console' | 'create' | 'creation' | 'declare' | 'deferred'
| 'delay' | 'delete' | 'delta' | 'digits' | 'domain' | 'else' |
'elsif' | 'end' | 'endl' | 'enum' | 'event' | 'exception' | 'exit'
| 'false' | 'find' | 'find_one' | 'find_only' | 'flush' | 'for' |
'function' | 'generate' | 'identifier' | 'if' | 'in' | 'instance'
| 'is' | 'is_a' | 'link' | 'loop' | 'many' | 'null' | 'object' |
'of' | 'one' | 'ordered_by' | 'others' | 'out' | 'pragma' |
'preferred' | 'private' | 'project' | 'public' | 'raise' | 'range' |
'readonly' | 'referential' | 'relationship' | 'return' | 'reverse'
| 'reverse_ordered_by' | 'schedule' | 'sequence' |
'service' | 'set' | 'start' | 'state' | 'structure' | 'terminal' |
'terminator' | 'then' | 'this' | 'to' | 'transition' | 'true' | 'type'
| 'unconditionally' | 'unique' | 'unlink' | 'using' | 'when' |
'while' | 'with';

1.2.4. Punctuators

MASL uses a number of character sequences to help define the syntactic structure by separating other tokens from each other.

[13] punctuator = '+' | '-' | '&' | '/' | '*' | '**' | '=' | '/=' | '>' | >=' | '<' | '<=' | '>>' |'>>>' | '<<' |'<<<' | ':=' | ':' | ',' | '.' | '<>' | '"' | '..' | '(' | ')' | '[' | ']' | '::' | ';' | '=>' | '->' | '~>' | '|';

1.2.5. Literals

A number of character sequences are used to denote a literal value of an underlying type. For example, the sequence `123` denotes the integer 123, and the sequence `@2010-05-27T16:02:45+01@` denotes the current time of writing.

1.2.5.1. Numeric Literals

A numeric literal represents a number. If the literal contains a decimal point or an exponent, then it is interpreted as a real number, otherwise it is interpreted as an integer.

By default, a literal is defined in base 10, but a different base (up to 36) may also be specified with a prefix. In this case, digits with value (decimal) 10 up to the new base are represented by the letters (A-Z or, equivalently, a-z) as appropriate. An exponent for a real number is always specified in decimal, regardless of the base for the mantissa, and the resultant number is interpreted as $\text{mantissa} \times \text{base}^{\text{exponent}}$

Example 1.1. Numeric Literal Examples

The following examples are some of the many different ways of representing the decimal value 251

- `251`
- `2#11111011`
- `16#FB`
- `36#6Z`

The following examples are some of the many different ways of representing the decimal value 10.25

- `10.25`
- `1025e-2`

- `1.025e1`
- `2#1010.01`
- `2#1.01001#+3`
- `16#A.4`
- `16#A4#-1`
- `36#A.9`
- `36#0.0A9#2`

Note

Note that negative values cannot be specified lexically (other than in the exponent), but can be represented syntactically with a unary expression.

[14]	integer literal	=	digits | base , based digits;
[15]	real literal	=	[digits] , '.' , digits , [exponent] | digits , ['.' , digits] , exponent | base , [based digits] , '.' , based digits , [based exponent] | base , based digits , ['.' , based digits] , based exponent;
[16]	digits	=	digit , { digit };
[17]	exponent	=	('e' | 'E') , ['+' | '-'] , digits;
[18]	base	=	digits , [digits], '#';
[19]	based digit	=	digit | letter;
[20]	based digits	=	based digit , { based digit };
[21]	based exponent	=	'#' ['+' | '-'] , digits;

1.2.5.2. Date and Time Literals

Date and time literals can represent either an instant in time (known as a timestamp), or a period of time (known as a duration). These literals are represented in MASL in a subset of ISO 8601:2004 format, surrounded by '@' characters.

1.2.5.2.1. Timestamp Literals

Timestamp literals are specified in ISO 8601:2004 Date or Date and Time format, as detailed in section 4.1 and 4.3 of the standard. A summary is provided below, but the standard should be considered the definitive guide. Reduced accuracy representations are allowed, and will be interpreted as the earliest instant within the interval covered by the accuracy loss. If no time is specified, then it will be interpreted as midnight UTC. e.g. `@2010-05@` is the same instant as `@2010-05-10T00:00:00Z@`. Expanded representations (which cover years after 9999AD) are not permitted. Handling of dates prior to 1582 is undefined.

All the complete representations consist of a date followed by a 'T' character followed by the time of day. Representations with reduced accuracy may drop the least significant fields. If any time fields are present, then both the 'T' separator and timezone must be present. Both extended (separator characters between fields) and extended (no separators) formats are accepted.

Dates may be specified in calendar date (YYYY-MM-DD), ordinal date (YYYY-DDD) or week date (YYYY-Www-D) formats. In calendar date format, January is represented as month 01, and so on. In ordinal date format, January 1st is represented as day 001, and so on. In week date format, week W01 represents the week that contains the first Thursday of the year, day 1 represents the Monday of that week, and day 7 the Sunday.

Times are specified in hh:mm:ss.s+hh:mm format. The least significant field supplied may be an arbitrary precison decimal. Timezones are interpreted as the number of hours ahead of UTC. UTC may be specified as either `+00` (and variations thereof) or `Z` .

Example 1.2. Timestamp Literal Examples

The following examples are some of the many different ways of representing 1am BST on 28th May 2010.

- `@2010-05-28T01:00:00+01@`
- `@20100528T010000+01@`
- `@2010-05-28T00:00:00Z@`
- `@2010-05-28T00:00@`
- `@2010-05-27T24:00@`
- `@2010-05-28@`
- `@2010-148@`

- `@2010-W21-5T00Z@`
- `@2010W215T00Z@`

[22] timestamp literal = '@' , ? ISO 8601:2004 Date Format (non-expanded) (Section 4.1) ? , '@' |
'@' , ? ISO 8601:2004 Date and Time of Day Format (non-expanded) (Section 4.3) ? , '@';

1.2.5.2.2. Duration Literals

Duration literals are represented in ISO 8601:2004 Duration format with designators and context information, as detailed in section 4.4.3.2 of the standard. Note in particular that the alternative format (from section 4.4.3.3) is not supported. A summary is provided below, but the standard should be considered the definitive guide. MASL does not support the use of year and month designators, as they cannot give a determinate date without context information.

The representations consist of the character 'P' (standing for 'period') followed by a series of fields with following identifiers. All fields must be in descending order of significance, and if any fields shorter than a day are present they must be preceded by the character 'T' (standing for 'time'). The least significant field present may be specified as an arbitrary precision decimal. If the week field is present, it must be the only field. At least one field must be present, but may be zero. Additionally a zero duration may be represented as the integer literal `0`.

The field identifiers are: 'W' = week, 'D' = day, 'H' = hours, 'M' = minutes and 'S' = seconds.

Example 1.3. Duration Literal Examples

The following examples are some of the many different ways of representing 1 day, 2 hours, 3 minutes and 4.5 seconds

- `@P1DT2H3M4.50000S@`
- `@PT26H3M4.5S@`
- `@PT93784.5S@`
- `@PT93784.5S@`
- `@P1DT2.05125H@`
- `@PT26.05125H@`

- `@P1.08546875D@`
- `@P0.155066964W@`

[23] duration literal = '@' , ? ISO 8601:2004 Duration General Format with Designators (Section 4.4.3.2) ? , '@';

1.2.5.3. Text Literals

A character literal consists of a single character or escape sequence surrounded by single quotes. A string literal consists of a sequence of characters or escape sequences surrounded by double quotes. Any character in the character set other than an end of line character , the escape character, a double quote character in a string literal or single quote character in a characer literal may be specified using its raw character. Any character not in this set must be specified using an escape sequence.

Example 1.4. Text Literal Examples

The following examples are some of the different ways of representing the string 'Hello World!'

- `"Hello World!"`
- `"Hello World\041"`
- `"Hello World\u0021"`

The following examples are some of the different ways of representing the backslash character

- `'\\'`
- `'\134'`
- `'\u005c'`

[24] character literal = single quote character , (character set - (escape character | single quote character | end of line character) | escape sequence) , single quote character;

[25] string literal = double quote character , (character set - (escape character | double quote character | end of line

			character) | escape sequence) , double quote character;
[26]	escape character	=	'\';
[27]	escape sequence	=	escape character , ('b' | 't' | 'n' | 'f' | 'r' | '"' | "'" | '\') | unicode escape | octal escape;
[28]	unicode escape	=	escape character , 'u' , hex digit , hex digit , hex digit , hex digit;
[29]	octal escape	=	escape character , [octal first digit] , [octal digit] , octal digit;
[30]	hex digit	=	digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';
[31]	octal first digit	=	'0' | '1' | '2' | '3';
[32]	octal digit	=	'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7';
[33]	double quote character	=	'"';
[34]	single quote character	=	"'";

1.2.5.4. Identifiers and Relationship Numbers

Identifiers consist of a letter followed by any number of letters, digits or underscores, where the resultant word is not a reserved word or a relationship number. A relationship number consists of the letter 'R' followed by a number.

Example 1.5. Identifier Examples

The following examples are all valid and distinct identifiers.

- `i`
- `fred`
- `Fred`
- `a_very_long_identifier_2`
- `AVeryLongIdentifer2`

The following examples are all valid relationship identifiers

- `R1`
- `R12`
- `R12345678`

[35] identifier = ((letter | '_') , { letter | digit | '_' }) - reserved word - relationship name;

[36] relationship name = 'R' , (digit - '0') , { digit };

2. Type System

MASL is a strongly typed language, which means that every variable, attribute, parameter and expression has a type that is known at compile-time. Types limit the values that a variable, attributed and parameter can hold, or that an expression can produce, limiting the operations supported on those values and determine the meaning of the operations. A type is characterized by a set of values and a set of operations, which implements the fundamental aspects of its semantics. An entity of a given type is a run-time entity that contains a value of the type.

2.1. Type Conversion

To specify rules regarding type conversion in MASL we first need to introduce some concepts which are used to determine which conversions are possible

2.1.1. Primitive Type

The primitive type of any type can be found by moving up through the heirarchy of type declarations until a built in type is reached, ignoring any constraints introduced by type definitions along the way. Once a built in type is reached the following table can be used to find the primitive type. If a type does not appear in the table below, then it is a primitive type itself.

Type	Primitive
`byte`	`long_integer`
`integer`	`long_integer`
`string`	`sequence of character`
`set of T`	`sequence of T`
`bag of T`	`sequence of T`
`array of T`	`sequence of T`

In addition, the primitive type of a structure is a structure with anonymous components of the same type as the components of the original structure. Any type defined to be an enumeration is its own primitive type.

2.1.2. Anonymous Types

All variables, parameters, object attributes and structure components have a known type which can be deduced from their definition. There are a number of expressions, however, which have a type which is never explicity specified, for example literals, navigates and finds. These are known as anonymous types, and they have slightly relaxed rules for assignability. It is also possible to explicity declare a variable,

parameter etc as having an anonymous type, which can sometimes be useful in specifiying generic interfaces.

2.1.3. Assignable

- A named type is assigable to the same named type
- An anonymous type is assignable to a named type if they have the same primitive type
- An named type is assignable to an anonymous type if they have the same primitive type
- An anonymous integer type is assignable to any type with a primitive type of real
- An anonymous structure is assignable to a type with a primitive type of a structure if the individual components are assignable. If the primitive type of the expression to be asssigned is not a structure, it should be promoted to an anonymous structure with a single component of the original expression before trying this test.
- An anonymous collection (set, array, bag or sequence) is assigable to a type with a primitive type of sequence if the contained elements are assignable. If the primitive type of the expression to be asssigned is not a sequence, it should be promoted to an anonymous sequence with a single element of the original expression before trying this test.

2.1.4. Convertible

to do

2.2. Type Declaration

to do

[37] type declaration = [type visibility] , 'type' , type name , 'is' , type definition , ';' , pragma list;

[38] type visibility = 'private' | 'public';

[39] type name = identifier ;

2.3. Type References

to do

[40] type ref = named type ref | instance type ref | collection type ref;

[41] named type ref = builtin type ref | user defined type ref;

2.4. Type Definition

to do

[42] type definition = structure type definition | enumeration type definition | constrained type definition | type ref;

[40] type ref = named type ref | instance type ref | collection type ref;

[41] named type ref = builtin type ref | user defined type ref;

2.4.1. Builtin Types

to do

[43] builtin type ref = ['anonymous'] ('character' | 'string' | 'boolean' | 'byte' | 'integer' | 'long_integer' | 'real' | 'device' | 'duration' | 'timestamp' | 'timer');

2.4.1.1. Boolean Type

to do

2.4.1.2. Integer Types

to do

2.4.1.3. Real Type

to do

2.4.1.4. Timestamp Type

to do

2.4.1.5. Duration Type

to do

2.4.1.6. String Type

to do

2.4.1.7. Character Type

to do

2.4.1.8. Timer Type

to do

2.4.1.9. Device Type

to do

2.4.2. User Defined Types

to do

[44]	user defined type ref	=	[domain name , '::'] , type name;

2.4.3. Enumeration Types

to do

[45]	enumeration type definition	=	'enum' , '(' , enumerator , { ',' , enumerator } , ')';
[46]	enumerator	=	enumerator name , [':=' , enumerator value];
[47]	enumerator name	=	identifier ;
[48]	enumerator value	=	const expression ;

2.4.4. Structure Types

to do

[49]	structure type definition	=	'structure' , structure component definition , { structure component definition } , 'end' , ['structure'];
[50]	structure component definition	=	component name , ':' , type ref , [':=' , component default value] , ';' , pragma list;
[51]	component name	=	identifier ;

[52]	component default value	=	const expression ;
[40]	type ref	=	named type ref | instance type ref | collection type ref;

2.4.5. Instance Types

to do

[53]	instance type ref	=	['anonymous'] , 'instance' , 'of' , object name;
[101]	object name	=	identifier ;

2.4.6. Collection Types

to do

[54]	collection type ref	=	['anonymous'] , 'sequence' , ['(' , expression , ')'] , 'of' , type ref | ['anonymous'] , 'array' , array bounds , 'of' , type ref | ['anonymous'] , 'set' , 'of' , type ref | ['anonymous'] , 'bag' , 'of' , type ref | ['anonymous'] , 'dictionary' , [[dictionary key type] , 'of' , type ref];
[55]	dictionary key type	=	named type ref | instance type ref;
[56]	array bounds	=	'(' , expression , ')';
[41]	named type ref	=	builtin type ref | user defined type ref;
[40]	type ref	=	named type ref | instance type ref | collection type ref;

2.4.7. Constrained Types

to do

[57]	constrained type definition	=	named type ref , type constraint;
[58]	type constraint	=	range constraint | delta constraint | digits constraint;
[59]	range constraint	=	'range' , expression;
[60]	delta constraint	=	'delta' , const expression , range constraint;
[61]	digits constraint	=	'digits' , const expression , range constraint;

3. Domains

In building large software systems, the analyst has to deal with a number of distinctly different subject matters or domains. A domain is a separate world inhabited by a distinct set of objects that behave according to the rules and policies characteristic of the domain.

Domains allow the specification of groups of logically related entities. Typically, a domain contains the declaration of a number of services, which can be called from outside the domain, whilst the inner workings of the domain remain hidden from outside users.

3.1. Domain Definition

A domain definition defines a new domain. The definition specifies the name of the domain together with declarations and definitions of its members,that is, type declarations, exception declarations, object declarations, and definitions, service and function declarations, terminator definitions, and relationship definitions.

[62] domain definition = 'domain' , domain name , 'is' , { domain member } , 'end' , ['domain'] , ';' , pragma list;

[63] domain name = identifier ;

[64] domain member = object declaration | domain service declaration | domain function declaration | terminator definition | relationship definition | object definition | type declaration | exception declaration;

3.2. Domain Services and Functions

A domain service provides an interface to the domain which may be invoked by means of a domain service call. The domain service will typically cause the domain to perform a sequence of actions.

A domain function provides an interface to retrieve information from the domain which may be invoked by means of a domain function call. The domain function will typically cause the domain to perform a sequence of actions to calculate a value, and return that value.

3.2.1. Domain Service and Function Declaration

A domain service declaration defines the name and parameters of a domain service. A domain function declaration defines the name, parameters and return type of a domain function. If the visibility of the declaration is `public`, then the declaration is available to be used by actions which have visibility of the domain itself. If the

visibility is `private`, then the declaration may only be used by actions within the domain in which it is declared.

The parameters on a declaration determine the arguments which must be passed to any call of a service or function, and the names by which they are referenced in the action definition. If a parameter has a mode of `in`, then its value will be passed into the action and will not be changed by the action. If a parameter is marked as `out`, then its value will be passed into the action and may be changed by the action. The return type of a function declaration is the type of the value that is returned by the function.

[65]	domain service declaration	=	[service visibility] , 'service' , service name , parameter list , ';' , pragma list;
[66]	domain function declaration	=	[service visibility] , 'function' , service name , parameter list , 'return' , return type , ';' , pragma list;
[69]	service visibility	=	'private' | 'public';
[70]	service name	=	identifier ;
[73]	parameter list	=	'(' , [parameter definition] , { ',' , parameter definition } , ')';
[74]	parameter definition	=	parameter name , ':' , parameter mode , parameter type;
[75]	parameter name	=	identifier ;
[76]	parameter mode	=	'in' | 'out';
[77]	parameter type	=	type ref ;
[78]	return type	=	type ref ;

3.2.2. Domain Service and Function Action Definition

A domain service or function action definition defines the action to be taken by the domain when the service or function is invoked. The signature of the definition must match exactly the signature of the corresponding declaration.

[67]	domain service definition	=	[service visibility] , 'service' , domain name , '::' , service name , parameter list , 'is' , code block , 'end' , ['service'] , ';' , pragma list;
[68]	domain function definition	=	[service visibility] , 'function' , domain name , '::' , service name , parameter list , 'return' , return type , 'is' , code block , 'end' , ['function'] , ';' , pragma list;
[63]	domain name	=	identifier ;

[70] service name = identifier ;

[79] code block = { variable declaration } , 'begin' , statement sequence ,
['exception' , { exception handler } , [other handler]];

4. Services, Functions and State Actions

4.1. Service, Function and State Declarations

to do

[65]	domain service declaration	=	[service visibility] , 'service' , service name , parameter list , ';' , pragma list;
[66]	domain function declaration	=	[service visibility] , 'function' , service name , parameter list , 'return' , return type , ';' , pragma list;
[96]	terminator service declaration	=	[service visibility] , 'service' , service name , parameter list , ';' , pragma list;
[97]	terminator function declaration	=	[service visibility] , 'function' , service name , parameter list , 'return' , return type , ';' , pragma list;
[111]	object service declaration	=	[service visibility] , [service type] , 'service' , service name , parameter list , ';' , pragma list;
[112]	object function declaration	=	[service visibility] , [service type] , 'function' , service name , parameter list , 'return' , return type , ';' , pragma list;
[116]	state declaration	=	[state type] , 'state' , state name , parameter list , ';' , pragma list;
[69]	service visibility	=	'private' | 'public';
[70]	service name	=	identifier ;
[71]	state type	=	'assigner' | 'assigner' , 'start' | 'creation' | 'terminal';
[72]	state name	=	identifier ;
[73]	parameter list	=	'(' , [parameter definition] , { ',' , parameter definition } , ')';
[74]	parameter definition	=	parameter name , ':' , parameter mode , parameter type;
[75]	parameter name	=	identifier ;
[76]	parameter mode	=	'in' | 'out';
[77]	parameter type	=	type ref ;
[78]	return type	=	type ref ;

4.2. Service, Function and State Action Definitions

to do

[67]	domain service definition	=	[service visibility] , 'service' , domain name , '::' , service name , parameter list , 'is' , code block , 'end' , ['service'] , ';' , pragma list;
[68]	domain function definition	=	[service visibility] , 'function' , domain name , '::' , service name , parameter list , 'return' , return type , 'is' , code block , 'end' , ['function'] , ';' , pragma list;
[98]	terminator service definition	=	[service visibility] , 'service' , domain name , '::' , terminator name , '~>' , service name , parameter list , 'is' , code block , 'end' , ['service'] , ';' , pragma list;
[99]	terminator function definition	=	[service visibility] , 'function' , domain name , '::' , terminator name , '~>' , service name , parameter list , 'return' , return type , 'is' , code block , 'end' , ['function'] , ';' , pragma list;
[114]	object service definition	=	[service visibility] , ['instance'] , 'service' , domain name , '::' , object name , '.' , service name , parameter list , 'is' , code block , 'end' , ['service'] , ';' , pragma list;
[115]	object function definition	=	[service visibility] , service type , 'function' , domain name , '::' , object name , '.' , service name , parameter list , 'return' , return type , 'is' , code block , 'end' , ['function'] , ';' , pragma list;
[117]	state definition	=	state type , 'state' , domain name , '::' , object name , '.' , state name , parameter list , 'is' , code block , 'end' , ['state'] , ';' , pragma list;
[69]	service visibility	=	'private' | 'public';
[63]	domain name	=	identifier ;
[94]	terminator name	=	identifier ;
[101]	object name	=	identifier ;
[70]	service name	=	identifier ;
[71]	state type	=	'assigner' | 'assigner' , 'start' | 'creation' | 'terminal';
[72]	state name	=	identifier ;
[73]	parameter list	=	'(' , [parameter definition] , { ',' , parameter definition } , ')';

[74]	parameter definition	=	parameter name , ':' , parameter mode , parameter type;
[75]	parameter name	=	identifier ;
[76]	parameter mode	=	'in' | 'out';
[77]	parameter type	=	type ref ;
[78]	return type	=	type ref ;
[79]	code block	=	{ variable declaration } , 'begin' , statement sequence , ['exception' , { exception handler } , [other handler]];

5. Relationships

5.1. Definition

to do

[80]	relationship definition	=	'relationship' , relationship name , 'is' , (regular relationship definition | assoc relationship definition | subtype relationship definition) , ';' , pragma list;
[81]	regular relationship definition	=	half relationship definition , ',' , half relationship definition;
[82]	assoc relationship definition	=	half relationship definition , ',' , half relationship definition , 'using' , assoc object name;
[83]	half relationship definition	=	object name , ['unconditionally' | 'conditionally'] , role phrase , ['one' | 'many'] , object name;
[84]	subtype relationship definition	=	supertype object name , 'is_a' , '(' , subtype object name , { ',' , subtype object name } , ')';
[85]	relationship name	=	relationship name ;
[86]	role phrase	=	identifier ;
[87]	assoc object name	=	object name ;
[88]	supertype object name	=	object name ;
[89]	subtype object name	=	object name ;
[101]	object name	=	identifier ;

5.2. Simple Relationships

to do

5.3. Associative Relationships

to do

5.4. Supertype Relationships

to do

5.5. Formalism

to do

5.6. Formalism

to do

6. Exceptions

6.1. Declaration

to do

[90]	exception declaration	=	[exception visibility] , 'exception' , exception name , ';' , pragma list;
[91]	exception name	=	identifier ;
[92]	exception visibility	=	'private' | 'public';

7. Terminators

A terminator is used to encapsulate services that the domain can call to publish information about itself, and services and functions that it can call to provide information information it requires. These services and functions differ from domain services and functions in that they typically are used as an interface to subject matters which lie outside the scope of the domain. Further details on services and functions in general can be found in Chapter 4, *Services, Functions and State Actions*.

7.1. Definition

Each terminator definition defines the services and functions that the enclosing domain wishes to provide or requires regarding one subject matter.

[93]	terminator definition	=	'terminator' , terminator name , 'is' , { terminator item } , 'end' , ['terminator'] , ';' , pragma list;
[94]	terminator name	=	identifier ;
[95]	terminator item	=	terminator service declaration | terminator function declaration;

7.1.1. Terminator Service

7.1.1.1. Declaration

A terminator service or function declaration declares that the domain wishes to publish a set of information about itself or that it requires certain information to be provided to it. The parameters on the service define what that information is, with published information contained in `in` parameters, and required information contained in `out` parameters or a function return value. Full details of service and function declarations are provided in Section 4.1, “Service, Function and State Declarations”.

[96]	terminator service declaration	=	[service visibility] , 'service' , service name , parameter list , ';' , pragma list;
[97]	terminator function declaration	=	[service visibility] , 'function' , service name , parameter list , 'return' , return type , ';' , pragma list;
[69]	service visibility	=	'private' | 'public';
[94]	terminator name	=	identifier ;

[70]	service name	=	identifier ;
[73]	parameter list	=	'(' , [parameter definition] , { ',' , parameter definition } , ')';
[74]	parameter definition	=	parameter name , ':' , parameter mode , parameter type;
[75]	parameter name	=	identifier ;
[76]	parameter mode	=	'in' | 'out';
[77]	parameter type	=	type ref ;
[78]	return type	=	type ref ;

7.1.1.2. Declaration

A terminator service definition defines the action that will publish or provide the information provided or required by the declaration. Full details of action definitions are provided in Section 4.2, “Service, Function and State Action Definitions”.

[98]	terminator service definition	=	[service visibility] , 'service' , domain name , '::' , terminator name , '~>' , service name , parameter list , 'is' , code block , 'end' , ['service'] , ';' , pragma list;
[99]	terminator function definition	=	[service visibility] , 'function' , domain name , '::' , terminator name , '~>' , service name , parameter list , 'return' , return type , 'is' , code block , 'end' , ['function'] , ';' , pragma list;
[63]	domain name	=	identifier ;
[94]	terminator name	=	identifier ;
[70]	service name	=	identifier ;
[79]	code block	=	{ variable declaration } , 'begin' , statement sequence , ['exception' , { exception handler } , [other handler]];

8. Objects

Objects are entities such that all the instances of the entities have the same characteristics and are subject to and conform to the same set of rules and policies.

Typically, an object contains declarations of the attributes that instances of the object contain, the declarations of services provided by the object and a description of the lifecycle of an instance of the object, if any.

8.1. Object Declaration

An object declaration declares a new object. The declaration specifies the name of the object only, to enable the object to be used in relationship and type definitions before it has been fully defined in an object definition.

A compile-time errors occurs if the name of the object is already in use within the current domain scope.

[100] object declaration = 'object' , object name , ';' , pragma list;

[101] object name = identifier ;

8.2. Object Definition

An object definition defines an object that has previously been declared. The definition specifies the name of the object together with the declarations and definitions of its members, that is, attribute definitions, identifier definitions, service declarations, function declarations, event declarations, state declarations, and lifecycle definitions.

A compile-time error occurs if the name of the object has not already been declared as an object within the current domain scope by means of an object declaration.

[102] object definition = 'object' , object name , 'is' , { object member } , 'end' , ['object'] , ';' , pragma list;

[101] object name = identifier ;

[103] object member = attribute definition | identifier definition | object service declaration | object function declaration | event definition | state declaration | transition table;

8.3. Object Attributes

Object attributes are used to store the internal state of an object. The type of an attribute is used to determine the values that the attribute can hold. An attribute's

type may not be convertible to an instance type, or be a structure or collection containing an instance type

A compile-time errors occurs if the name of the attribute is already in use within the current object scope.

[104] attribute definition = attribute name , ':' , ['preferred'] , ['unique'] , [attribute referentials] , attribute type , [':=' , attribute default value] , ';' , pragma list;

[105] attribute name = identifier ;

[106] attribute type = type ref ;

8.3.1. Preferred Attribute

An attribute should be marked as `preferred` if it forms part of the preferred identifier for the object. Note that the preferred identifier is not specified explicity using an identifier definition, but is formed implicitly from the attributes marked as preferred. As an object must have a preferred identifier, at least one attribute of each object must be marked as `preferred`.

8.3.2. Unique Attribute

An attribute should be marked as `unique` if the actual value of the attribute is unimportant, but required to be unique within all instances of the current object. This is typically the case when an object has no natural combination of attributes to form an identifier, and an artificial identifier must be created. The type of a preferred identifier attribute must be convertible to an integer.

8.3.3. Default Value

If a default value is specified, then the expression is evaluated and assigned to the attribute each time an instance of the object is created using an instance creation expression, and an explicit value is not supplied for the attribute. The expression must be valid within the scope of the current object.

A compile time error occurs if the default value is not assignable to the attribute.

[107] attribute default value = const expression ;

8.3.4. Referential Attributes

A relationship between two or more objects must be formalized using referential attributes. The object or objects that formalize the relationship must specify

referential attributes for each of the identifier attributes of one identifier of the other objects participating in the relationship. Each attribute may specify multiple referentials, which indicates that the matching identifier attributes must all have the same value for the relationship to be valid. Each referential must specify which relationship it refers to, along with the name of the identifier attribute that it formalizes. Further details on formalizing relationships may be found in Section 5.5, “Formalism”.

The value of a referential attribute is automatically set when the first relationship which it formalizes is formed, or on instance creation if it is also an identifier. A non-identifier attribute has an undefined value if it is not involved in formalizing any relationships. An attempt to read an undefined value will result in an exception being raised. Once the attribute has a defined value, any further relationships formed with the instance must cause the attribute to have the same value. The value may not be set manually, other than during instance creation for an identifier attribute.

[108] attribute referentials = 'referential' , '(' , attribute referential , { ',' , attribute referential } , ')';

[109] attribute referential = relationship spec , '.' , attribute name;

8.4. Object Identifiers

An identifier defines the set of attributes which uniquely identify an instance. All identifier attributes must be set when the instance is created, and may not be changed after that point. Identifiers are used in the formalizing of relationships to set the value of any referential attributes which refer to them.

[110] identifier definition = 'identifier' , 'is' , '(' , attribute name , { ',' , attribute name } , ')' , ';' , pragma list;

8.4.1. Preferred Identifier

Every object must have exactly one preferred identifier, which is specified by marking the participating attributes as preferred. No special status is attached to the preferred identifier, other than denoting that it is the identifier which will most usually be used to refer to the object. In addition if an ordering expression is used to sort a collection of instances, but no attributes are supplied to denote the order, then the preferred identifier will be used in the order the attributes are defined on the object.

8.5. Object Services and Functions

Object services and functions define the interface to an object, and may be instance or non-instance based. Further details on services and functions in general can be

found in Chapter 4, *Services, Functions and State Actions*, so this section will just highlight where object services differ.

8.5.1. Object Service Declaration

Object services and functions are declared within an object definition. In addition to the standard service and function declaration information detailed in Section 4.1, “Service, Function and State Declarations” an object service or function declaration also specifies whether it is instance based or deferred. An object may overload service declarations with the same name based on parameter types as with the general case, but must not declare both instance and object based services with the same name.

[111]	object service declaration	=	[service visibility] , [service type] , 'service' , service name , parameter list , ';' , pragma list;
[112]	object function declaration	=	[service visibility] , [service type] , 'function' , service name , parameter list , 'return' , return type , ';' , pragma list;
[113]	service type	=	'instance' , ['deferred' , '(' , relationship name , ')'];
[69]	service visibility	=	'private' | 'public';
[70]	service name	=	identifier ;
[73]	parameter list	=	'(' , [parameter definition] , { ',' , parameter definition } , ')';
[74]	parameter definition	=	parameter name , ':' , parameter mode , parameter type;
[75]	parameter name	=	identifier ;
[76]	parameter mode	=	'in' | 'out';
[77]	parameter type	=	type ref ;
[78]	return type	=	type ref ;

8.5.2. Instance Based Services and Functions

Object Services and functions may be instance or non-instance based. Instance based services act upon a given instance of the object, whereas non-instance based services act at the level of the of the instance population as a whole. To declare a service or function as instance based, the `instance` qualifier must be used in the declaration.

8.5.2.1. Instance Service Deferral

An instance based service or function which is declared with the `deferred` modifier is called a deferred service. Deferred services do not have a definition themselves, but defer the definition to another object further down a subtype hierarchy. The relationship sepcified in the deferal declaration must be a subtype relationship, with the current object as the supertype. All subtypes of that relationship must also declare a service or function with exactly the same signature as the deferred service. Those subtypes may in turn defer definition to their own subtypes.

When a deferred service or function is invoked, the instance's currently active subtype is examined. That subtype then becomes the current instance, and the equivalent service is invoked.

8.5.3. Object Service Action Defintion

An object service or function definition defines the action to be performed when the service or function is called. In addition to the standard action definition features detailed in Section 4.2, “Service, Function and State Action Definitions”, object based services work within the scope of their enclosing object, so all object services, functions, and events may be referred to by name without being qualified by the object name. Note that any parameters or local variables with identical names will hide these object level names. Instance services and functions allow reference to attributes, services and functions of the current instance by name, and introduce the name `this` which may be used to refer to the current instance explicity.

[114]	object service definition	=	[service visibility] , ['instance'] , 'service' , domain name , '::' , object name , '.' , service name , parameter list , 'is' , code block , 'end' , ['service'] , ';' , pragma list;
[115]	object function definition	=	[service visibility] , service type , 'function' , domain name , '::' , object name , '.' , service name , parameter list , 'return' , return type , 'is' , code block , 'end' , ['function'] , ';' , pragma list;
[63]	domain name	=	identifier ;
[101]	object name	=	identifier ;
[70]	service name	=	identifier ;
[79]	code block	=	{ variable declaration } , 'begin' , statement sequence , ['exception' , { exception handler } , [other handler]];

8.6. Object Lifecycles

An object may be either passive or active. Active objects have a lifecycle that is modelled using states and events. A state represents a stage within the lifecycle, and has an action associated with it that is carried out upon entry to the state. An event causes and instance or assigner to change from one state to another, and can carry data to the state action. The state that is entered on receipt of a particular event is defined in the state transition table.
Any object may define an instance lifecycle. In this case every instance of the object has a lifecycle.
Associative objects may also have an assigner lifecycle. Assigner lifecycles provide a single point of control through which competing requests are serialized. They are used to manage the associations formed by relationship that the associative object formalizes. Because of this, there is only one copy of an assigner lifecycle for all instances of an object.

8.6.1. States

A state represents a stage in an instance or assigner lifecycle. Further details on services, functions and state actions in general can be found in Chapter 4, *Services, Functions and State Actions*, so this section will just highlight where state actions differ from the general case.

8.6.1.1. State Declaration

A state declaration declares a state in the lifecycle of an object instance or assigner object. It specifies the type and name of the state, and any parameters which must be passed to the state action by any events which the state transition table shows as causing the state to be entered. As states are always entered asynchronously, they must not declare parameters with a mode of `out`.

[116] state declaration = [state type] , 'state' , state name , parameter list , ';' , pragma list;

[72] state name = identifier ;

[71] state type = 'assigner' | 'assigner' , 'start' | 'creation' | 'terminal';

[73] parameter list = '(' , [parameter definition] , { ',' , parameter definition } , ')';

[74] parameter definition = parameter name , ':' , parameter mode , parameter type;

[76] parameter mode = 'in' | 'out';

[75] parameter name = identifier ;

[77] parameter type = type ref ;

8.6.1.1.1. Creation States

A state that is declared as `creation` is entered when a creation event is sent to an object. It does not have an associated instance, but would typically result in an instance of the object being created with its current state set to the creation state.

> **Deprecated**
>
> Use of creation states is deprecated, and they are likely to be removed in a future version of the language. Synchronous creation of instances should be preferred.

8.6.1.1.2. Instance States

A state that is declared without a state type is an instance state. An instance state is always entered with respect to a particular instance, which becomes the instance to which the name `this` refers within the state definition.

8.6.1.1.3. Terminal States

A state that is declared as `terminal` is an instance state which represents a state in which an instance will be deleted. A terminal state is always entered with respect to a particular instance, which becomes the instance to which the name `this` refers within the state definition.

8.6.1.1.4. Assigner Start States

A state that is declared as `assigner start` is an an assigner state which represents the state in which an assigner lifecycle begins. Exactly one state of an assigner lifecycle must be declared as a start state. Once the lifecycle is active, the start state acts exactly as any other assigner state.

8.6.1.1.5. Assigner States

A state that is declared as `assigner` is an an assigner state which represents any state of an assigner lifecycle other than the start state.

8.6.1.2. State Action Defintion

A state definition defines a state action to be performed upon entry to the state. In addition to the standard action definition features detailed in Section 4.2, “Service, Function and State Action Definitions”, state actions work within the scope of their

enclosing object, so all object services, functions, and events may be referred to by name without being qualified by the object name. Note that any parameters or local variables with identical names will hide these object level names. Instance state actions allow reference to attributes, services and functions of the current instance by name, and introduce the name `this` which may be used to refer to the current instance explicity. At the end of the state action, the lifecycle of the instance or assigner is defined to be in the relevant state.

[117]	state definition	=	state type , 'state' , domain name , '::' , object name , '.' , state name , parameter list , 'is' , code block , 'end' , ['state'] , ';' , pragma list;
[63]	domain name	=	identifier ;
[101]	object name	=	identifier ;
[72]	state name	=	identifier ;
[71]	state type	=	'assigner' | 'assigner' , 'start' | 'creation' | 'terminal';
[79]	code block	=	{ variable declaration } , 'begin' , statement sequence , ['exception' , { exception handler } , [other handler]];

8.6.2. Events

An event signals to an instance or assigner that its lifecyle should move from one state to another. An event can carry data to the destination state for its state action to act upon. The state transition table shows which state the lifecyle should move into depending on which state it is already in. An event is generated to an instance or assigner by an action by means of an event generation statement.

[118]	event definition	=	[event type] , 'event' , event name , parameter list , ';' , pragma list;
[119]	event name	=	identifier ;
[120]	event type	=	'assigner' | 'creation';
[73]	parameter list	=	'(' , [parameter definition] , { ',' , parameter definition } , ')';
[74]	parameter definition	=	parameter name , ':' , parameter mode , parameter type;

8.6.2.1. Creation Events

An event that is defined as an `creation` event is always generated without reference to a particular instance, and causes a creation state to be entered.

Deprecated

Use of creation events is deprecated, and they are likely to be removed in a future version of the language. Synchronous creation of instances should be preferred.

8.6.2.2. Instance Events

An event that is defined without an event type is an instance event. An instance event is always generated with respect to a particular instance, and is destined for the lifecyle of that instance.

8.6.2.3. Polymorphic Events

Any instance events that are defined on a supertype object are inherited by all subtypes. Once the state action (if any) for the supertype has completed, the event is passed on to any subtypes. Polymorphic events are not defined on the subtype, but referenced in the subtype's state transition table by prefixing them with the supertype object name.

8.6.2.4. Assigner Events

An event that is defined as an `assigner` event is always generated without reference to a particular instance, and is destined for the assigner lifecyle of the containing object.

8.6.3. State Transition Table

A state transition table defines the transitions in the lifecycle of an instance or assigner object. Given any starting state, it specifies whether a particular event is valid for a lifecycle in that state, and if so whether the event should cause transition to another state or be ignored.

The following conditions must be met for any object:

- If any creation, instance or terminal states are declared, the object must define exactly one instance state transition table.
- If any assigner or assigner start states are declared, the object must define exactly one assigner state transition table.

The following conditions must be met in any instance state transition table:

- Every instance, creation or terminal state declared by the object must occur as a start state in exactly one row of the instance state transition table.

- Assigner or assigner start states must not occur as a start state.
- Exactly one row must have a start state of `Non_Existent`.
- Every row must contain a transition option for each instance or creation event defined in the enclosing object.
- Every row must contain a transition option for each polymorphic event defined in every supertype of the enclosing object.
- For the `Non_Existent` start state, all transition options for instance or terminal events must be marked as `Cannot_Happen`
- For the `Non_Existent` start state, all transition options for creation events must result in a creation state.
- All transitions from creation, instance or terminal start states for creation events must be marked as `Cannot_Happen`
- All transitions from creation, instance or terminal start states for instance or polymorphic events must not result in a creation state.
- All transitions from a terminal start state must be marked as `Ignore` or `Cannot_Happen`.
- Each incoming event with parameter types that do not match exactly the parameter types of the end state must be marked as `Ignore` or `Cannot_Happen`.

The following conditions must be met in any assigner state transition table:

- Every assigner or assigner start state declared by the enclosing object must occur as a start state in exactly one row of the assigner transition table.
- Instance, creation or terminal states must not occur as a start state.
- All rows must not have a start state of `Non_Existent`.
- All rows must contain exactly one transition option for each assigner event defined in the enclosing object.
- Each incoming event with parameter types that do not match exactly the end state parameter types must be marked as `Ignore` or `Cannot_Happen`.

[121] transition table = ['assigner'] , 'transition' , 'is' , transition row , { transition row } , 'end' , 'transition' , ';' , pragma list;

[122] transition row = start state , '(' , transition option , { ',' , transition option } , ')' , ';' , pragma list;

[123] transition option = incoming event , '=>' , end state;

[124] incoming event = [object name , '.'] , event name;

[125] start state = 'Non_Existent' | state name;

[126] end state = state name | 'Ignore' | 'Cannot_Happen';

9. Statements

A statement defines an action to be performed upon its execution. Some statements contain other statements as part of their structure; such other statements are sub-statements of the statement. In the same manner, some statements contain expressions as part of their structure.

A statement will always complete the action specified in the following sections unless any of the sub-statements or sub-expressions cause an exception to be raised or an exit or return statement are encountered as sub-statements. In this case the statement is deemed to have completed unsuccessfully. A statement sequence will complete each of the statements it contains in that order, unless once of those statements does not complete successfully. In this case, any remaining statements in the sequence will not be performed, and the sequence is also deemed to have completed unsucessfully.

[127] statement sequence = { statement };

[128] statement = (code block statement | assign statement | stream statement | null statement | call statement | exit statement | return statement | delay statement | raise statement | delete statement | link statement | unlink statement | schedule statement | cancel timer statement | generate statement | if statement | case statement | for statement | while statement | empty statement) , ';' , pragma list;

9.1. Code Block

A code block firstly executes each of the variable declarations, and then executes the sequence of statements as described above. If the sequence of statements is not sucessfully completed due to an exception being raised, then each of the exception handlers are examined in turn. If the exception name of the handler matches the exception that was raised, then the statement sequence defined for that handler is performed and the code block is deemed to have completed successfully. If no handler matches the exception, and an other handler is present, then the statement sequence for the other handler is performed and the code block is deemed to have completed successfully.

[129] code block statement = ['declare' , { variable declaration }] , 'begin' , statement sequence , ['exception' , { exception handler } , [other handler]] , 'end';

[130] variable declaration = variable name , ':' , ['readonly'] , type ref , [':=' , expression] , ';' , pragma list;

[131]	exception handler	=	'when' , qualified exception name , '=>' , statement sequence ;
[132]	other handler	=	'when' , 'others' , '=>' , statement sequence;
[133]	qualified exception name	=	[domain name , '::'] , exception name;
[225]	variable name	=	identifier ;
[40]	type ref	=	named type ref | instance type ref | collection type ref;
[63]	domain name	=	identifier ;
[91]	exception name	=	identifier ;

9.2. Empty Statement

An empty or null statement has no effect.

[134]	empty statement	=	;
[135]	null statement	=	'null';

9.3. Assignment

An assignment statement assigns the value of the rhs expression to the lhs expression. The following conditions must be met:

- The right hand side expression must be assignable to the left hand side expression.
- The left hand side expression must be writeable.

[136]	assign statement	=	expression , ':=' , expression;

9.4. Stream

A stream statement streams (i.e. reads or writes) data to or from a device. The following condition must be met:

- The stream value expression must be assignable to `anonymous device`.

Four streaming operators are available:

>> Skips any leading whitespace, reads a text representation of a value of the same type as the stream value expression and assigns this value to the

expression. No more characters are read from the stream once the next character to be read does not form part of a valid text representation, or, in the case of a string value, is a whitespace character. The following conditions must be met:

- The stream value expression must be writeable.

>>> Reads all data from the device until the next newline character, and writes it to the stream value expression. The following conditions must be met:

- The stream value expression must be writeable.
- The stream value expression must be assignable to `anonymous string`.

<< Writes a text representation of the stream value expression to the device.

<<< Writes a text representation of the stream value expression to the device followed by a newline character.

[137] stream statement = device expression , stream value , { stream value };

[138] stream value = ('>>' | '<<' | '>>>' | '<<<') , expression;

[139] device expression = expression ;

9.5. Service Call

A service call invokes the action defined by a matching service definition of the correct type. Any arguments supplied in the call are passed to the action as parameters. A matching service is defined by the following rules:

- The service name in the call must be the same as the name in the service declaration.
- The number of arguments in the call must be the same as the number of parameters in the service declaration.
- Each of the arguments in the call must be assignable to the corresponding parameter type of the service declaration.
- Any arguments which correspond to a parameter with a parameter mode of `out` must be writeable.

- The service must be the only service for which the previous rules apply.

The type of service to be invoked cannot be inferred by syntax alone, so the following rules will be applied:

- If the call prefix is of the form `domainName::serviceName` then the call is a domain service call to a matching service on the named domain.
- If the call prefix is of the form `terminatorName~>serviceName` then the call is a terminator service call to a matching service on the named terminator.
- If the call prefix is of the form `objectName.serviceName` then the call is an object service call to a matching object service on the named object.
- If the call prefix is of the form `expression.serviceName` then the call is an instance service call to a matching instance service on the instance type of the expression. The expression must be of an instance type. The name `this` inside the service action will resolve to the same instance as the expression.
- If the call prefix is of the form `serviceName`, then the following rules are applied in order.

  1. If the enclosing scope is an instance based service, function or state, and an instance or object service with the same name is declared on the enclosing object, then the call is an instance service call to a matching instance service on the enclosing object. The name `this` inside the service action will resolve to the same instance as `this` in the current scope.

  2. If the enclosing scope is an object based service, function or state, and an instance or object service with the same name is declared on the enclosing object, then the call is an object service call to a matching object service on the enclosing object.

  3. Otherwise, the call is a domain service call to a matching domain service on the enclosing domain.

[140]	call statement	=	domain service call | terminator service call | object service call | instance service call;
[141]	domain service call	=	[domain name , '::'] , service name , '(' , [argument list] , ')';
[142]	terminator service call	=	terminator name , '~>' , service name , '(' , [argument list] , ')';

[143]	object service call	=	[object name , '.'] , service name , '(' , [argument list] , ')';
[144]	instance service call	=	[suffix expression , '.'] , service name , '(' , [argument list] , ')';
[63]	domain name	=	identifier ;
[94]	terminator name	=	identifier ;
[101]	object name	=	identifier ;
[70]	service name	=	identifier ;
[209]	argument list	=	argument , { ',' , argument };
[210]	argument	=	expression;

9.6. Exit Statement

An exit statement causes control to transfer immediately from the current sequence of statements to the innermost enclosing while or for statement. If a `when` clause is present, this transfer of control only happens when the supplied condition evaluates to `true`. The enclosing `while` or `for` statement is then deemed to have completed normally.

The following conditions must be met:

- The statement must be a enclosed by a while or for statement.
- If present, the condition expression must be assignable to `anonymous boolean`.

[145]	exit statement	=	'exit' , ['when' , condition];
[168]	condition	=	expression ;

9.7. Return Statement

A return statement causes control to transfer immediately from the current sequence of statements to the end of the current function action definition. The action definition is deemed to have completed normally, and the return value of the function is set to the supplied expression.

The following conditions must be met:

- The statement must be enclosed in a function action definition.
- The expression must be assignable to the return type of the enclosing function.

[146] return statement = 'return' , expression;

9.8. Delay Statement

A delay statement causes the current process to pause for duration of the supplied expression.

The following conditions must be met:

- The expression must be assignable to `anonymous duration`.

[147] delay statement = 'delay' , expression;

9.9. Raise Exception

A raise statement causes an exception to be raised, and the statement is deemed to have completed unsuccessfully.

[148] raise statement = 'raise' , qualified exception name , ['(' , [expression] , ')'];

9.10. Delete Instance

A delete instance statement causes the instance or instances referred to by the expression to be deleted. Attempting to delete a null instance has no effect. If the expression is an instance type, then it will be set to null on completion of the statement, and if a collection it will contain no elements.

If an instances to be deleted is linked to other instances, then that instance will not be deleted, an exception will be raised and the statement will be deemed to have completed unsuccessfully. If the deletion is of an ordered collection of instances (eg a `sequence`), then any instances that precede the instance in question will be deleted, but any instances following will not. If the collection is unordered, then no guarantees are made regarding which instances will be deleted, other than that the instance in question will not be deleted.

The following conditions must be met:

- The expression must be assignable to an instance type or a collection of instance types.

  The expression must be writeable.

[149] delete statement = 'delete' , expression;

9.11. Link Instances

The `link` statement causes a relationship between two or more instances to be formed.

[150] link statement = 'link' , navigate expression , relationship spec , navigate expression , ['using' , navigate expression];

9.11.1. Simple Relationships

This section describes link statements where the relationship spec specifies a navigation along a simple relationship from object A to object B.

The link statement causes a relationship to be formed between the instance or instances referenced by the first expression and the instance or instances referenced by the second expression. If referential integrity is compromised by any of the requested relationships an exception will be raised and the relationship will not be created. If a relationship already exists between any pair of instances then an exception will be raised. If an exception is raised during the linking of a collection of instances it is undefined which, if any, of the other requested relationships will have been successfully formed if the exception is subsequently handled.

The following conditions must be met:

- The `using` clause must not be present.
- The first instance expression must be assignable to `anonymous instance of A` or, when the cardinality of A in the relationship is `many`, the first instance expression may also be assignable to `anonymous sequence of anonymous instance of A`.
- The second instance expression must be assignable to `anonymous instance of B` or, when the cardinality of B in the relationship is `many`, the second instance expression may also be assignable to `anonymous sequence of anonymous instance of B`.

9.11.2. Associative Relationships

This section describes link statements where the relationship spec specifies a navigation along an associative relationship from object A to object B with associative object C. See also the link expression for details of linking collections of instances in associative relationships, and for details of linking without explicitly specifying an associative instance.

The link statement causes a relationship to be formed between the instance referenced by the first expression and the instance referenced by the second expression using the instance referenced by the third expression as an associative.

If referential integrity is compromised the requested relationship an exception will be raised and the relationship will not be created. If a relationship already exists between the instances then an exception will be raised.

The following conditions must be met:

- The `using` clause must be present.
- The first instance expression must be assignable to `anonymous instance of A`.
- The second instance expression must be assignable to `anonymous instance of B`.
- The third instance expression must be assignable to `anonymous instance of C`.

9.11.3. Supertype Relationships

This section describes link statements where the relationship spec specifies a navigation along a supertype relationship from object A to object B.

The link statement causes a relationship to be formed between the instance referenced by the first expression and the instance referenced by the second expression. If referential integrity is compromised the requested relationship an exception will be raised and the relationship will not be created. If a relationship already exists between the instance acting as a supertype and any instance acting as a subtype in the same relationship, then an exception will be raised and the relationship will not be created.

The following conditions must be met:

- The `using` clause must not be present.
- The first instance expression must be assignable to `anonymous instance of A`.
- The second instance expression must be assignable to `anonymous instance of B`.

9.12. Unlink Instances

The `unlink` statement causes a relationship between two or more instances to be removed.

[151] unlink statement = 'unlink' , navigate expression , relationship spec , [navigate expression , ['using' , navigate expression]];

9.12.1. Simple Relationships

This section describes unlink statements where the relationship spec specifies a navigation along a simple relationship from object A to object B.

If the second expression is present, the unlink statement causes a relationship to be removed between the instance or instances referenced by the first expression and the instance or instances referenced by the second expression. If a relationship does not already exist between any pair of instances then an exception will be raised. If an exception is raised during the unlinking of a collection of instances it is undefined which, if any, of the other requested relationships will have been successfully removed if the exception is subsequently handled.

If the second expression not is present, the unlink statement causes a relationship to be removed between the instance or instances referenced by the first expression and all instances found by navigating from those instances along the relationship spec.

The following conditions must be met:

- The `using` clause must not be present.
- The first instance expression must be assignable to `anonymous instance of A` or, when the cardinality of A in the relationship is `many`, the first instance expression may also be assignable to `anonymous sequence of anonymous instance of A`.
- If present, the second instance expression must be assignable to `anonymous instance of B` or, when the cardinality of B in the relationship is `many`, the second instance expression may also be assignable to `anonymous sequence of anonymous instance of B`.

9.12.2. Associative Relationships

This section describes unlink statements where the relationship spec specifies a navigation along an associative relationship from object A to object B with associative object C. See also the link expression for details of unlinking collections of instances in associative relationships, and for details of unlinking without explicitly specifying an associative instance.

The unlink statement causes a relationship to be removed between the instance referenced by the first expression and the instance referenced by the second expression using the instance referenced by the third expression as an associative. If a relationship does not already exist between the instances then an exception will be raised.

The following conditions must be met:

- The second instance expression and `using` clause must be present.

- The first instance expression must be assignable to `anonymous instance of A`.
- The second instance expression must be assignable to `anonymous instance of B`.
- The third instance expression must be assignable to `anonymous instance of C`.

9.12.3. Supertype Relationships

This section describes unlink statements where the relationship spec specifies a navigation along a supertype relationship from object A to object B (either A or B may be the supertype).

If the second expression is present, the unlink statement causes a relationship to be removed between the instance referenced by the first expression and the instance referenced by the second expression. If a relationship does not already exist between the instances then an exception will be raised.

If the second expression not is present, the unlink statement causes a relationship to be removed between the instance referenced by the first expression and the instance found by navigating from that instances along the relationship spec. If the first instance is the supertype in the relationship, this means that the relationship to any subtype will be removed.

The following conditions must be met:

- The `using` clause must not be present.
- The first instance expression must be assignable to `anonymous instance of A`.
- The second instance expression must be assignable to `anonymous instance of B`.

9.13. Event Generation

todo

[152]	generate statement	=	'generate' , qualified event name , '(' , [argument list] , ')' , ['to' , expression];
[153]	schedule statement	=	'schedule' , timer id , generate statement , ('at' | 'delay') , expression , ['delta' , expression];
[154]	cancel timer statement	=	'cancel' , timer id;

[155]	timer id	=	expression ;
[156]	qualified event name	=	[object name , '.'] , event name;
[119]	event name	=	identifier ;
[101]	object name	=	identifier ;
[209]	argument list	=	argument , { ',' , argument };
[210]	argument	=	expression;

9.14. Conditional Processing

todo

[157]	if statement	=	'if' , condition , 'then' , statement sequence , { elsif block } , [else block] , 'end' , ['if'];
[158]	elsif block	=	'elsif' , condition , 'then' , statement sequence;
[159]	else block	=	'else' , statement sequence;
[160]	case statement	=	'case' , expression , 'is' , { case alternative } , [case others] , 'end' , ['case'];
[161]	case alternative	=	'when' , choice list , '=>' , statement sequence;
[162]	case others	=	'when' , 'others' , '=>' , statement sequence;
[163]	choice list	=	expression , { '|' , expression };
[168]	condition	=	expression ;

9.15. Looping

todo

9.15.1. While Statement

todo

[164]	while statement	=	'while' , condition , 'loop' , statement sequence , 'end' , ['loop'];
[168]	condition	=	expression ;

9.15.2. For Statement

todo

[165] for statement = 'for' , loop variable spec , 'loop' , statement sequence , 'end' , ['loop'];

[166] loop variable spec = identifier , 'in' , ['reverse'] , expression;

10. Expressions

todo

[167] expression = range expression ;

[168] condition = expression ;

[169] const expression = expression ;

10.1. Concepts

10.1.1. Writeable

to do

10.2. Range

todo

[170] range expression = logical or , ['..' , logical or];

10.3. Logical Operators

todo

[171] logical or = [logical or , 'or'] , logical xor;

[172] logical xor = [logical xor , 'xor'] , logical and;

[173] logical and = [logical and , 'and'] , equality exp;

10.4. Comparison Operators

todo

[174] equality exp = [equality exp , ('=' | '/=')] , relational exp;

[175] relational exp = [relational exp , ('<' | '>' | '<=' | '>=')] , additive exp;

10.5. Additive Operators

todo

[176] additive exp = [additive exp , ('+' | '-' | '&' | 'union' | 'not_in')] , multiplicative exp;

10.6. Multiplicative Operators

todo

[177] multiplicative exp = [multiplicative exp , ('*' | '/' | 'mod' | '**' | 'rem' | 'intersection' | 'disunion')] , unary exp;

10.7. Unary Operators

todo

[178] unary exp = ('-' | '+' | 'not' | 'abs') , unary exp | link expression| unlink expression;

10.8. Relationship Linking

todo

10.8.1. Link Expression

todo

[179] link expression = navigate expression | 'link' , navigate expression , relationship spec , navigate expression;

10.8.2. Unlink Expression

todo

[180] unlink expression = navigate expression | 'unlink' , navigate expression , relationship spec , [navigate expression];

10.9. Relationship Navigation

todo

[181] navigate expression = extended expression | simple nav expression | correlated nav expression | ordering expression;

[182] simple nav expression = navigate expression , '->' , relationship spec , [where clause];

[183] correlated nav expression = navigate expression , 'with' , extended expression , '->' , relationship spec;

[184] relationship spec = relationship name , ['.' , role phrase] , ['.' , object name];

[185]	extended expression	=	suffix expression | create expression | find expression;
[193]	where clause	=	'(' , [find condition] , ')';

10.10. Ordering

todo

[186]	ordering expression	=	navigate expression , ['ordered_by' | 'reverse_ordered_by'] , sort order;
[187]	sort order	=	'(' , [sort order component , { ',' , sort order component }] , ')';
[188]	sort order component	=	['reverse'] , identifier;

10.11. Instance Creation

todo

[189]	create expression	=	'create' , ['unique'] , object name , create argument list;
[190]	create argument list	=	'(' , [create argument , { ',' , create argument }] , ')';
[191]	create argument	=	attribute name , '=>' , expression | 'Current_State' , '=>' , state name;
[101]	object name	=	identifier ;

10.12. Finds

todo

[192]	find expression	=	['find' | 'find_one' | 'find_only'] , suffix expression nc , [where clause];
[193]	where clause	=	'(' , [find condition] , ')';
[194]	find condition	=	find logical or ;
[195]	find logical or	=	[find logical or , 'or'] , find logical xor;
[196]	find logical xor	=	[find logical xor , 'xor'] , find logical and;
[197]	find logical and	=	[find logical and , 'and'] , find primary;
[198]	find primary	=	find comparison | find unary;

[199]	find comparison	=	find name , ('=' | '/=' | '<' | '>' | '<=' | '>=') , additive exp;
[200]	find unary	=	'not' , find unary | '(' , find condition , ')';
[201]	find name	=	attribute name , { '.' , component name | '[' , index , ']' };
[105]	attribute name	=	identifier ;
[51]	component name	=	identifier ;
[202]	index	=	expression ;

10.13. Suffix Expressions

todo

[203]	suffix expression	=	primary expression | component access expression | attribute access expression | slice expression | characteristic expression | cast expression | instance fn call expression | object fn call expression | domain fn call expression | terminator fn call expression;
[204]	suffix expression nc	=	primary expression | component access expression nc | attribute access expression nc | slice expression nc | characteristic expression nc;

10.13.1. Function Calls

todo

[205]	instance fn call expression	=	[suffix expression , '.'] , service name , '(' , [argument list] , ')';
[206]	object fn call expression	=	[object name , '.'] , service name, '(' , [argument list] , ')';
[207]	domain fn call expression	=	[domain name , '::'] , service name, '(' , [argument list] , ')';
[208]	terminator fn call expression	=	terminator name , '~>' , service name, '(' , [argument list] , ')';
[209]	argument list	=	argument , { ',' , argument };
[210]	argument	=	expression;

10.13.2. Attribute, Component and Collection Accessors

todo

[211] slice expression = suffix expression , '[' , index , ']';

[212] slice expression nc = suffix expression nc , '[' , index , ']';

[213] attribute access expression = suffix expression , '.' , component name;

[214] attribute access expression nc = suffix expression nc , '.' , attribute name;

[215] component access expression = suffix expression , '.' , attribute name;

[216] component access expression nc = suffix expression nc , '.' , component name;

10.13.3. Characteristics

todo

[217] characteristic expression = suffix expression , "'" , characteristic , ['(' , [argument list] , ')'];

[218] characteristic expression nc = suffix expression nc , "'" , characteristic;

[219] characteristic = identifier ;

10.13.4. Type Casts

todo

[220] cast expression = type ref , '(' , expression , ')';

10.14. Primary Expressions

todo

[221] primary expression = literal | parenthesised expression | name expression;

[222] parenthesised expression = structure aggregate expression | '(' , expression , ')';

[223]	literal	=	integer literal | real literal | character literal | string literal | timestamp literal | duration literal | 'true' | 'false' | 'null' | 'flush' | 'endl' | 'this' | 'console' | '#LINE#' | '#FILE#';
[224]	name expression	=	variable name | parameter name | attribute name;
[225]	variable name	=	identifier ;
[75]	parameter name	=	identifier ;
[105]	attribute name	=	identifier ;
[226]	structure aggregate expression	=	'(' , expression , ',' , expression , { ',' , expression } , ')';

11. Projects

to do

[227]	project definition	=	'project' , project name , 'is' , { project item } , 'end' , ['project'] , ';' , pragma list;
[228]	project item	=	domain prj definition ;
[229]	domain prj definition	=	'domain' , domain name , 'is' , { domain prj item } , 'end' , ['domain'] , ';' , pragma list;
[230]	domain prj item	=	terminator definition ;
[231]	project name	=	identifier ;

12. Pragmas

Pragmas may be attached to any definition, declaration or statement. They have no meaning in the language, but may be used by compilers for any purpose.

[232] pragma list = { pragma , ';' };

[233] pragma = 'pragma' , pragma name , '(' , [pragma value , { ',' , pragma value }] , ')';

[234] pragma value = identifier | literal;

[235] pragma name = identifier ;

A. Alphabetical Syntax Reference

[176]	additive exp	=	[additive exp , ('+' | '-' | '&' | 'union' | 'not_in')] , multiplicative exp;
[210]	argument	=	expression;
[209]	argument list	=	argument , { ',' , argument };
[56]	array bounds	=	'(' , expression , ')';
[136]	assign statement	=	expression , ':=' , expression;
[87]	assoc object name	=	object name ;
[82]	assoc relationship definition	=	half relationship definition , ',' , half relationship definition , 'using' , assoc object name;
[213]	attribute access expression	=	suffix expression , '.' , component name;
[214]	attribute access expression nc	=	suffix expression nc , '.' , attribute name;
[107]	attribute default value	=	const expression ;
[104]	attribute definition	=	attribute name , ':' , ['preferred'] , ['unique'] , [attribute referentials] , attribute type , [':=' , attribute default value] , ';' , pragma list;
[105]	attribute name	=	identifier ;
[109]	attribute referential	=	relationship spec , '.' , attribute name;
[108]	attribute referentials	=	'referential' , '(' , attribute referential , { ',' , attribute referential } , ')';
[106]	attribute type	=	type ref ;
[18]	base	=	digits , [digits], '#';
[19]	based digit	=	digit | letter;
[20]	based digits	=	based digit , { based digit };
[21]	based exponent	=	'#' ['+' | '-'] , digits;

[43]	builtin type ref	=	['anonymous'] ('character' | 'string' | 'boolean' | 'byte' | 'integer' | 'long_integer' | 'real' | 'device' | 'duration' | 'timestamp' | 'timer');
[140]	call statement	=	domain service call | terminator service call | object service call | instance service call;
[154]	cancel timer statement	=	'cancel' , timer id;
[161]	case alternative	=	'when' , choice list , '=>' , statement sequence;
[162]	case others	=	'when' , 'others' , '=>' , statement sequence;
[160]	case statement	=	'case' , expression , 'is' , { case alternative } , [case others] , 'end' , ['case'];
[220]	cast expression	=	type ref , '(' , expression , ')';
[24]	character literal	=	single quote character , (character set - (escape character | single quote character | end of line character) | escape sequence) , single quote character;
[1]	character set	=	digit | letter | punctuation character | whitespace character;
[219]	characteristic	=	identifier ;
[217]	characteristic expression	=	suffix expression , ''' , characteristic , ['(' , [argument list] , ')'];
[218]	characteristic expression nc	=	suffix expression nc , ''' , characteristic;
[163]	choice list	=	expression , { '|' , expression };
[79]	code block	=	{ variable declaration } , 'begin' , statement sequence , ['exception' , { exception handler } , [other handler]];
[129]	code block statement	=	['declare' , { variable declaration }] , 'begin' , statement sequence , ['exception' , { exception handler } , [other handler]] , 'end';
[54]	collection type ref	=	['anonymous'] , 'sequence' , ['(' , expression , ')'] , 'of' , type ref | ['anonymous'] , 'array' , array bounds , 'of' , type ref | ['anonymous'] , 'set' , 'of' , type ref | ['anonymous'] , 'bag' , 'of' , type ref |

			['anonymous'] , 'dictionary' , [[dictionary key type] , 'of' , type ref];
[8]	comment	=	'/' , '/' , { single line character } , end of line;
[215]	component access expression	=	suffix expression , '.' , attribute name;
[216]	component access expression nc	=	suffix expression nc , '.' , component name;
[52]	component default value	=	const expression ;
[51]	component name	=	identifier ;
[168]	condition	=	expression ;
[169]	const expression	=	expression ;
[57]	constrained type definition	=	named type ref , type constraint;
[183]	correlated nav expression	=	navigate expression , 'with' , extended expression , '->' , relationship spec;
[191]	create argument	=	attribute name , '=>' , expression | 'Current_State' , '=>' , state name;
[190]	create argument list	=	'(' , [create argument , { ',' , create argument }] , ')';
[189]	create expression	=	'create' , ['unique'] , object name , create argument list;
[147]	delay statement	=	'delay' , expression;
[149]	delete statement	=	'delete' , expression;
[60]	delta constraint	=	'delta' , const expression , range constraint;
[139]	device expression	=	expression ;
[55]	dictionary key type	=	named type ref | instance type ref;
[2]	digit	=	'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

[16]	digits	=	digit , { digit };
[61]	digits constraint	=	'digits' , const expression , range constraint;
[62]	domain definition	=	'domain' , domain name , 'is' , { domain member } , 'end' , ['domain'] , ';' , pragma list;
[207]	domain fn call expression	=	[domain name , '::'] , service name, '(' , [argument list] , ')';
[66]	domain function declaration	=	[service visibility] , 'function' , service name , parameter list , 'return' , return type , ';' , pragma list;
[68]	domain function definition	=	[service visibility] , 'function' , domain name , '::' , service name , parameter list , 'return' , return type , 'is' , code block , 'end' , ['function'] , ';' , pragma list;
[64]	domain member	=	object declaration | domain service declaration | domain function declaration | terminator definition | relationship definition | object definition | type declaration | exception declaration;
[63]	domain name	=	identifier ;
[229]	domain prj definition	=	'domain' , domain name , 'is' , { domain prj item } , 'end' , ['domain'] , ';' , pragma list;
[230]	domain prj item	=	terminator definition ;
[141]	domain service call	=	[domain name , '::'] , service name , '(' , [argument list] , ')';
[65]	domain service declaration	=	[service visibility] , 'service' , service name , parameter list , ';' , pragma list;
[67]	domain service definition	=	[service visibility] , 'service' , domain name , '::' , service name , parameter list , 'is' , code block , 'end' , ['service'] , ';' , pragma list;
[33]	double quote character	=	"";
[23]	duration literal	=	'@' , ? ISO 8601:2004 Duration General Format with Designators (Section 4.4.3.2) ? , '@';
[159]	else block	=	'else' , statement sequence;
[158]	elsif block	=	'elsif' , condition , 'then' , statement sequence;
[134]	empty statement	=	;

[10]	end of line	=	? ISO 646 CR character ? | ? ISO 646 LF character ? | ? ISO 646 CR character ? , ? ISO 646 LF character ?;
[7]	end of line character	=	? ISO 646 CR character ? | ? ISO 646 LF character ?;
[126]	end state	=	state name | 'Ignore' | 'Cannot_Happen';
[45]	enumeration type definition	=	'enum' , '(' , enumerator , { ',' , enumerator } , ')';
[46]	enumerator	=	enumerator name , [':=' , enumerator value];
[47]	enumerator name	=	identifier ;
[48]	enumerator value	=	const expression ;
[174]	equality exp	=	[equality exp , ('=' | '/=')] , relational exp;
[26]	escape character	=	'\';
[27]	escape sequence	=	escape character , ('b' | 't' | 'n' | 'f' | 'r' | '"' | ''' | '\') | unicode escape | octal escape;
[118]	event definition	=	[event type] , 'event' , event name , parameter list , ';' , pragma list;
[119]	event name	=	identifier ;
[120]	event type	=	'assigner' | 'creation';
[90]	exception declaration	=	[exception visibility] , 'exception' , exception name , ';' , pragma list;
[131]	exception handler	=	'when' , qualified exception name , '=>' , statement sequence ;
[91]	exception name	=	identifier ;
[92]	exception visibility	=	'private' | 'public';
[145]	exit statement	=	'exit' , ['when' , condition];
[17]	exponent	=	('e' | 'E') , ['+' | '-'] , digits;
[167]	expression	=	range expression ;

[185]	extended expression	=	suffix expression | create expression | find expression;
[199]	find comparison	=	find name , ('=' | '/=' | '<' | '>' | '<=' | '>=') , additive exp;
[194]	find condition	=	find logical or ;
[192]	find expression	=	['find' | 'find_one' | 'find_only'] , suffix expression nc , [where clause];
[197]	find logical and	=	[find logical and , 'and'] , find primary;
[195]	find logical or	=	[find logical or , 'or'] , find logical xor;
[196]	find logical xor	=	[find logical xor , 'xor'] , find logical and;
[201]	find name	=	attribute name , { '.' , component name | '[' , index , ']' };
[198]	find primary	=	find comparison | find unary;
[200]	find unary	=	'not' , find unary | '(' , find condition , ')';
[165]	for statement	=	'for' , loop variable spec , 'loop' , statement sequence , 'end' , ['loop'];
[152]	generate statement	=	'generate' , qualified event name , '(' , [argument list] , ')' , ['to' , expression];
[83]	half relationship definition	=	object name , ['unconditionally' | 'conditionally'] , role phrase , ['one' | 'many'] , object name;
[30]	hex digit	=	digit | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';
[35]	identifier	=	((letter | '_') , { letter | digit | '_' }) - reserved word - relationship name;
[110]	identifier definition	=	'identifier' , 'is' , '(' , attribute name , { ',' , attribute name } , ')' , ';' , pragma list;
[157]	if statement	=	'if' , condition , 'then' , statement sequence , { elsif block } , [else block] , 'end' , ['if'];
[124]	incoming event	=	[object name , '.'] , event name;
[202]	index	=	expression ;
[6]	inline whitespace character	=	? ISO 646 SP character ? | ? ISO 646 HT character ? | ? ISO 646 FF character ?;
[205]	instance fn call expression	=	[suffix expression , '.'] , service name , '(' , [argument list] , ')';

[144]	instance service call	=	[suffix expression , '.'] , service name , '(' , [argument list] , ')';
[53]	instance type ref	=	['anonymous'] , 'instance' , 'of' , object name;
[14]	integer literal	=	digits | base , based digits;
[3]	letter	=	'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z';
[179]	link expression	=	navigate expression | 'link' , navigate expression , relationship spec , navigate expression;
[150]	link statement	=	'link' , navigate expression , relationship spec , navigate expression , ['using' , navigate expression];
[223]	literal	=	integer literal | real literal | character literal | string literal | timestamp literal | duration literal | 'true' | 'false' | 'null' | 'flush' | 'endl' | 'this' | 'console' | '#LINE#' | '#FILE#';
[173]	logical and	=	[logical and , 'and'] , equality exp;
[171]	logical or	=	[logical or , 'or'] , logical xor;
[172]	logical xor	=	[logical xor , 'xor'] , logical and;
[166]	loop variable spec	=	identifier , 'in' , ['reverse'] , expression;
[177]	multiplicative exp	=	[multiplicative exp , ('*' | '/' | 'mod' | '**' | 'rem' | 'intersection' | 'disunion')] , unary exp;
[224]	name expression	=	variable name | parameter name | attribute name;
[41]	named type ref	=	builtin type ref | user defined type ref;
[181]	navigate expression	=	extended expression | simple nav expression | correlated nav expression | ordering expression;
[135]	null statement	=	'null';
[100]	object declaration	=	'object' , object name , ';' , pragma list;
[102]	object definition	=	'object' , object name , 'is' , { object member } , 'end' , ['object'] , ';' , pragma list;

[206]	object fn call expression	=	[object name , '.'] , service name, '(' , [argument list] , ')';
[112]	object function declaration	=	[service visibility] , [service type] , 'function' , service name , parameter list , 'return' , return type , ';' , pragma list;
[115]	object function definition	=	[service visibility] , service type , 'function' , domain name , '::' , object name , '.' , service name , parameter list , 'return' , return type , 'is' , code block , 'end' , ['function'] , ';' , pragma list;
[103]	object member	=	attribute definition | identifier definition | object service declaration | object function declaration | event definition | state declaration | transition table;
[101]	object name	=	identifier ;
[143]	object service call	=	[object name , '.'] , service name , '(' , [argument list] , ')';
[111]	object service declaration	=	[service visibility] , [service type] , 'service' , service name , parameter list , ';' , pragma list;
[114]	object service definition	=	[service visibility] , ['instance'] , 'service' , domain name , '::' , object name , '.' , service name , parameter list , 'is' , code block , 'end' , ['service'] , ';' , pragma list;
[32]	octal digit	=	'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7';
[29]	octal escape	=	escape character , [octal first digit] , [octal digit] , octal digit;
[31]	octal first digit	=	'0' | '1' | '2' | '3';
[186]	ordering expression	=	navigate expression , ['ordered_by' | 'reverse_ordered_by'] , sort order;
[132]	other handler	=	'when' , 'others' , '=>' , statement sequence;
[74]	parameter definition	=	parameter name , ':' , parameter mode , parameter type;
[73]	parameter list	=	'(' , [parameter definition] , { ',' , parameter definition } , ')';
[76]	parameter mode	=	'in' | 'out';
[75]	parameter name	=	identifier ;

[77]	parameter type	=	type ref ;
[222]	parenthesised expression	=	structure aggregate expression | '(' , expression , ')';
[233]	pragma	=	'pragma' , pragma name , '(' , [pragma value , { ',' , pragma value }] , ')';
[232]	pragma list	=	{ pragma , ';' };
[235]	pragma name	=	identifier ;
[234]	pragma value	=	identifier | literal;
[221]	primary expression	=	literal | parenthesised expression | name expression;
[227]	project definition	=	'project' , project name , 'is' , { project item } , 'end' , ['project'] , ';' , pragma list;
[228]	project item	=	domain prj definition ;
[231]	project name	=	identifier ;
[4]	punctuation character	=	'_' | '{' | '}' | '[' | ']' | '#' | '(' | ')' | '<' | '>' | '%' | ':' | ';' |'.' | '?' | '*' | '+' | '-' | '/' | '^' | '&' | '|' | '~' | '!' | '=' | ',' | '\' | '"' | ''';
[13]	punctuator	=	'+' | '-' | '&' | '/' | '*' | '**' | '=' | '/=' | '>' | >=' | '<' | '<=' | '>>' |'>>>' | '<<' |'<<<' | ':=' | ':' | ',' | '.' | '<>' | '"' | '..' | '(' | ')' | '[' | ']' | '::' | ';' | '=>' | '->' | '~>' | '|';
[156]	qualified event name	=	[object name , '.'] , event name;
[133]	qualified exception name	=	[domain name , '::'] , exception name;
[148]	raise statement	=	'raise' , qualified exception name , ['(' , [expression] , ')'];
[59]	range constraint	=	'range' , expression;
[170]	range expression	=	logical or , ['..' , logical or];
[15]	real literal	=	[digits] , '.' , digits , [exponent] | digits , ['.' , digits] , exponent | base , [based digits] , '.' , based digits , [based exponent] | base , based digits , ['.' , based digits] , based exponent;

[81]	regular relationship definition	=	half relationship definition , ',' , half relationship definition;
[175]	relational exp	=	[relational exp , ('<' | '>' | '<=' | '>=')] , additive exp;
[80]	relationship definition	=	'relationship' , relationship name , 'is' , (regular relationship definition | assoc relationship definition | subtype relationship definition) , ';' , pragma list;
[36]	relationship name	=	'R' , (digit - '0') , { digit };
[85]	relationship name	=	relationship name ;
[184]	relationship spec	=	relationship name , ['.' , role phrase] , ['.' , object name];
[12]	reserved word	=	'Cannot_Happen' | 'Current_State' | 'Ignore' | 'Non_Existent' | 'anonymous' | 'array' | 'assigner' | 'at' | 'bag' | 'begin' | 'cancel' | 'case' | 'conditionally' | 'console' | 'create' | 'creation' | 'declare' | 'deferred' | 'delay' | 'delete' | 'delta' | 'digits' | 'domain' | 'else' | 'elsif' | 'end' | 'endl' | 'enum' | 'event' | 'exception' | 'exit' | 'false' | 'find' | 'find_one' | 'find_only' | 'flush' | 'for' | 'function' | 'generate' | 'identifier' | 'if' | 'in' | 'instance' | 'is' | 'is_a' | 'link' | 'loop' | 'many' | 'null' | 'object' | 'of' | 'one' | 'ordered_by' | 'others' | 'out' | 'pragma' | 'preferred' | 'private' | 'project' | 'public' | 'raise' | 'range' | 'readonly' | 'referential' | 'relationship' | 'return' | 'reverse' | 'reverse_ordered_by' | 'schedule' | 'sequence' | 'service' | 'set' | 'start' | 'state' | 'structure' | 'terminal' | 'terminator' | 'then' | 'this' | 'to' | 'transition' | 'true' | 'type' | 'unconditionally' | 'unique' | 'unlink' | 'using' | 'when' | 'while' | 'with';
[146]	return statement	=	'return' , expression;
[78]	return type	=	type ref ;
[86]	role phrase	=	identifier ;
[153]	schedule statement	=	'schedule' , timer id , generate statement , ('at' | 'delay') , expression , ['delta' , expression];
[70]	service name	=	identifier ;

[113]	service type	=	'instance' , ['deferred' , '(' , relationship name , ')'];
[69]	service visibility	=	'private' | 'public';
[182]	simple nav expression	=	navigate expression , '->' , relationship spec , [where clause];
[9]	single line character	=	digit | letter | punctuation character | inline whitespace character;
[34]	single quote character	=	'''';
[211]	slice expression	=	suffix expression , '[' , index , ']';
[212]	slice expression nc	=	suffix expression nc , '[' , index , ']';
[187]	sort order	=	'(' , [sort order component , { ',' , sort order component }] , ')';
[188]	sort order component	=	['reverse'] , identifier;
[125]	start state	=	'Non_Existent' | state name;
[116]	state declaration	=	[state type] , 'state' , state name , parameter list , ';' , pragma list;
[117]	state definition	=	state type , 'state' , domain name , '::' , object name , '.' , state name , parameter list , 'is' , code block , 'end' , ['state'] , ';' , pragma list;
[72]	state name	=	identifier ;
[71]	state type	=	'assigner' | 'assigner' , 'start' | 'creation' | 'terminal';
[128]	statement	=	(code block statement | assign statement | stream statement | null statement | call statement | exit statement | return statement | delay statement | raise statement | delete statement | link statement | unlink statement | schedule statement | cancel timer statement | generate statement | if statement | case statement | for statement | while statement | empty statement) , ';' , pragma list;
[127]	statement sequence	=	{ statement };

[137]	stream statement	=	device expression , stream value , { stream value };
[138]	stream value	=	('>>' | '<<' | '>>>' | '<<<') , expression;
[25]	string literal	=	double quote character , (character set - (escape character | double quote character | end of line character) | escape sequence) , double quote character;
[226]	structure aggregate expression	=	'(' , expression , ',' , expression , { ',' , expression } , ')';
[50]	structure component definition	=	component name , ':' , type ref , [':=' , component default value] , ';' , pragma list;
[49]	structure type definition	=	'structure' , structure component definition , { structure component definition } , 'end' , ['structure'];
[89]	subtype object name	=	object name ;
[84]	subtype relationship definition	=	supertype object name , 'is_a' , '(' , subtype object name , { ',' , subtype object name } , ')';
[203]	suffix expression	=	primary expression | component access expression | attribute access expression | slice expression | characteristic expression | cast expression | instance fn call expression | object fn call expression | domain fn call expression | terminator fn call expression;
[204]	suffix expression nc	=	primary expression | component access expression nc | attribute access expression nc | slice expression nc | characteristic expression nc;
[88]	supertype object name	=	object name ;
[93]	terminator definition	=	'terminator' , terminator name , 'is' , { terminator item } , 'end' , ['terminator'] , ';' , pragma list;
[208]	terminator fn call expression	=	terminator name , '~>' , service name, '(' , [argument list] , ')';

[97]	terminator function declaration	=	[service visibility] , 'function' , service name , parameter list , 'return' , return type , ';' , pragma list;
[99]	terminator function definition	=	[service visibility] , 'function' , domain name , '::' , terminator name , '~>' , service name , parameter list , 'return' , return type , 'is' , code block , 'end' , ['function'] , ';' , pragma list;
[95]	terminator item	=	terminator service declaration | terminator function declaration;
[94]	terminator name	=	identifier ;
[142]	terminator service call	=	terminator name , '~>' , service name , '(' , [argument list] , ')';
[96]	terminator service declaration	=	[service visibility] , 'service' , service name , parameter list , ';' , pragma list;
[98]	terminator service definition	=	[service visibility] , 'service' , domain name , '::' , terminator name , '~>' , service name , parameter list , 'is' , code block , 'end' , ['service'] , ';' , pragma list;
[155]	timer id	=	expression ;
[22]	timestamp literal	=	'@' , ? ISO 8601:2004 Date Format (non-expanded) (Section 4.1) ? , '@' | '@' , ? ISO 8601:2004 Date and Time of Day Format (non-expanded) (Section 4.3) ? , '@';
[123]	transition option	=	incoming event , '=>' , end state;
[122]	transition row	=	start state , '(' , transition option , { ',' , transition option } , ')' , ';' , pragma list;
[121]	transition table	=	['assigner'] , 'transition' , 'is' , transition row , { transition row } , 'end' , 'transition' , ';' , pragma list;
[58]	type constraint	=	range constraint | delta constraint | digits constraint;
[37]	type declaration	=	[type visibility] , 'type' , type name , 'is' , type definition , ';' , pragma list;
[42]	type definition	=	structure type definition | enumeration type definition | constrained type definition | type ref;
[39]	type name	=	identifier ;

[40]	type ref	=	named type ref | instance type ref | collection type ref;
[38]	type visibility	=	'private' | 'public';
[178]	unary exp	=	('-' | '+' | 'not' | 'abs') , unary exp | link expression| unlink expression;
[28]	unicode escape	=	escape character , 'u' , hex digit , hex digit , hex digit , hex digit;
[180]	unlink expression	=	navigate expression | 'unlink' , navigate expression , relationship spec , [navigate expression];
[151]	unlink statement	=	'unlink' , navigate expression , relationship spec , [navigate expression , ['using' , navigate expression]];
[44]	user defined type ref	=	[domain name , '::'] , type name;
[130]	variable declaration	=	variable name , ':' , ['readonly'] , type ref , [':=' , expression] , ';' , pragma list;
[225]	variable name	=	identifier ;
[193]	where clause	=	'(' , [find condition] , ')';
[164]	while statement	=	'while' , condition , 'loop' , statement sequence , 'end' , ['loop'];
[11]	whitespace	=	whitespace character , { whitespace character };
[5]	whitespace character	=	inline whitespace character | end of line character;

Bibliography

Standards

[ISO646] *Information technology — ISO 7-bit coded character set for information interchange*. International Organization for Standardization, 1991. ©1991.

[ISO8601] *Data elements and interchange formats — Information interchange — Representations of dates and times*. International Organization for Standardization, 2004. ©2004.

Index