MASL Tutorial

Version 1.4

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

This tutorial is not an introduction to programming but an overall description of programming in MASL[MASL06]. It is assumed that the reader will have significant experience of programming in some high level language, along with knowledge of Shlaer-Mellor OOA modelling ([Shlaer88], [Shlaer92] and [Mellor02]).

In Chapter 1, *Types*, we introduce the notion of declaration and assignment together with the ideas of scope and visibility. The important concepts of type, subtype and constraints are then explained. The remainder of this chapter discusses some of the specific and general types in MASL.

In Chapter 2, *Execution Control* we learn about the main sequential control structures of MASL. Chapter 3, *Composite Types* continues from where Chapter 1, *Types* left off, by describing the more complex types of MASL. Chapter 4, *Exceptions* describes the use of exceptions and exception handling. In Chapter 5, *Objects and Relationships* we finally get to the main use of OOA, and describe how instances of objects are created, found and deleted. In addition, this section discusses how we create, navigate and delete relationships between these instances. In Chapter 6, *Actions* we discuss actions, which are the places where statements, together with appropriate declarations, that encapsulate a set of processes are held. Finally, in Chapter 7, *Device Input/Output* we discuss how to read and write data to and from devices and files. This tutorial is a companion to the MASL Reference Manual [MASL06], which should be referenced for a completion language description.

Bibliography

Books

[MASL06] 2006.

[Mellor02] MELLOR, Stephen and BALCER, Marc. Addison Wesley Professional., 2002. ISBN ISBN 0-201-74804-5.

[Shlaer88] SHLAER, Sally and MELLOR, Stephen. Prentice Hall, Inc., 1988. ISBN ISBN 0-13-629023-X.

[Shlaer92] SHLAER, Sally and MELLOR, Stephen. Prentice Hall, Inc., 1992. ISBN ISBN 0-13-629940-7.

1. Types

This section starts by considering the declaration of entities, the assignment of values to them and the ideas of scope and visibility. We then introduce the important concepts of type, subtype and constraints. As examples of types, the remainder of the chapter discusses the numeric types integer and real, enumeration types in general, the type boolean, the date and time types, the character and string types and the operations on them.

1.1. Declarations and Assignments

Values can be stored in entities, which are declared to be of a specific type. Entities are either variables, in which case their value may change (or vary) as the program executes, or they may be constants, in which case they keep their same initial value throughout their life. A variable is introduced into a program by a declaration, which consists of the name (that is, the identifier) of the variable followed by a colon and then the name of the type. This can then optionally be followed by the := symbol and an initial value. The declaration terminates with a semicolon. Thus, we might write:

```
  x : integer;
  y : integer := 29;
```

This introduces the variable x of type integer but gives it no particular initial value and then the variable y and gives it the specific initial value of 29. If a variable is declared and not given an initial value then great care must be taken not to use the undefined value of the variable until one has been properly given to it. A common way to give a value to a variable is by using an assignment statement. In this, the identifier of the variable is followed by := and then some expression giving the new value. The statement terminates with a semicolon. An example might look like:

```
  declare
    x : integer;
    y : integer := 29;
  begin
    x := 14;
    y := x + 71;
  end;
```

Both are valid assignment statements and place new values in x and y thereby overwriting their previous values. Note that `:=` can be followed by any expression provided that it produces a value of the type of the variable being assigned to.
A constant is declared in a similar way to a variable by inserting the reserved word `readonly` after the colon. Of course, a constant must be initialised in its declaration otherwise it would be useless. An example might be:

```
  e : readonly real := 2.718281828;
```

1.2. Blocks and Scopes

MASL carefully distinguishes between declarations, which introduce new identifiers, and statements that do not. It is clearly only sensible that the declarations which introduce new identifiers should precede the statements which manipulate them. Accordingly, declarations and statements occur in separate places in the program text. The simplest fragment of text, which includes declarations and statements, is a block. A block commences with the reserved word declare, some declarations, begin, some statements and concludes with the reserved word end and the terminating semi-colon. A trivial example is:

```
  declare
    x : integer := 2;       // declarations here
  begin
    x := x + 63;            // statements here
  end;
```

A block is itself an example of a statement and so one of the statements in its body could be another block. This textual nesting of blocks can continue indefinitely. Since a block is a statement it can be executed like any other statement. When this happens the declarations in its declarative part (the bit between the `declare` and the `begin`) are elaborated in order and then the statements in the body (between the `begin` and the `end`) are executed in the usual way. Note the terminology, we elaborate declarations and execute statements. All that the elaboration of a declaration does is make the thing being declared come into existence and then evaluate and assign any initial value to it. When we come to the end of the block all the things, which were declared in the block automatically, cease to exist.

The scope determines both the visibility and lifetime of names defined within that scope. For example:

```
  declare
    x : integer := 47;
  begin
    // only x available
    declare
      y : integer := 3;
    begin
      // both x and y available
    end;
    // only x available, y out of scope
  end;
```

A variable declared within a scope is available only to the end of that scope. Note that you can do the following:

```
declare
  x : integer := 12;
begin
  declare
    x : integer := 96;   // previous x hidden
  begin

  end;
end;
```

The redeclaration of x in the inner block is allowed, but this hides the previous declaration, contained in the outer scope.

1.3. Types

A type is characterised by a set of values, and a set of operations, which implement the fundamental aspects of its semantics. Every type has a name, which is introduced in a type declaration. Moreover, every type declaration introduces a new type quite distinct from any other type. The set of values belonging to two distinct types are themselves quite distinct al-though in some cases the actual lexical form of the values may be identical – which one is meant at any point is determined by the context. Values of one type cannot be assigned to variables of another type. This is the fundamental rule of strong typing.

A type declaration consists of the reserved word `type`, the identifier to be associated with the type, the reserved word `is` and then the definition of the type followed by the terminating semicolon. The type definition between `is` and the semicolon gives in some way the set of values belonging to the type. As a concrete example, consider the following:

```
  type colour type is enum (BLUE, RED, GREEN);
```

This introduces a new type called colour_type. Moreover, it states that there are only three values of this type and they are denoted by the identifiers `BLUE`, `RED` and `GREEN`. Variables of this type can then be declared in the usual ways:

```
  c : colour_type;
  d : colour_type := RED;               // initial value can be supplied
  e : readonly colour_type := BLUE;     // a constant can be declared
```

We have stated that values (or variables) of one type cannot be assigned to variables of another type. Therefore, one cannot assign colours to integers and so the assignment shown below is illegal.

```
  declare
    i : integer;
    c : colour_type := RED;
  begin
    i := c;     //  illegal
  end;
```

1.4. Subtypes

We now introduce subtypes and constraints. A subtype, as its name suggests, characterises a set of values, which is just a subset of the values of some other type known as the base type. The subset is defined by means of a constraint. Constraints take various forms according to the category of the base type. As is usual with subsets, the subset may be the complete set. There is, however, no way of restricting this set of operations of the base type. The subtype takes all the operations; sub setting applies only to the values. As an example, we declare a subtype thus:

```
  subtype day_number_type is integer range 1 .. 31;
```

We can then declare variables and constants using the subtype identifier in exactly the same way as a type identifier:

```
  d : day_number_type;
```

We are then assured that the variable d can take only integer values from `1` to `31` inclusive. The compiler will insert run-time checks if necessary to ensure that this is so; if a check fails then an exception is raised.

It is important to realise that a subtype declaration does not introduce a new distinct type. An entity such as d is of type integer and so the following is perfectly legal from the syntactic point of view.

```
  declare
    d : day_number_type;
    i : integer;
  begin
    d := i;
  end;
```

Of course, on execution, the value of i may or may not lie in the range 1 ..31.If it does, then all is well; if not then an exception will be raised. Assignment in the other direction:

```
  i := d;
```

will always work. It is not always necessary to impose a constraint. It is perfectly legal to write:

```
  subtype unconstrained_day_number_type is integer;
```

although in this instance it is not of much value. A subtype may be defined in terms of a previous subtype:

```
  subtype february_day_number_type is day_number_type range 1 .. 29;
```

Any additional constraint must of course satisfy existing constraints:

```
  subtype illegal_type is day_number_type range 0 .. 10;        // illega
```

would be incorrect and cause a compile-time error.

We conclude this section by summarising the assignment statement and the rules of strong typing. Assignment has the form:

```
  variable := expression;
```

and the two rules are:

- Both sides must have the same base type.
- The expression must satisfy any constraints on the variables; if it does not, the assignment does not take place and an exception is raised instead.

Note

The general principle is that type errors (violations of the first rule) are detected during compilation whereas subtype errors (violations of the second rule) are detected during execution by the raising of an exception.

1.5. Simple Numeric Types

The types integer and real are not built-in MASL types. They both have been declared somewhere in terms of the built-in numeric type. For the moment, however, we will suppose that they are both built-in.

As we have seen, a constraint may be imposed on type integer by using the reserved word `range`. This is then followed by two expressions separated by two dots, which, of course, must produce values of integer type.

We turn now to a brief consideration of the floating-point types. It is possible to apply constraints to the type real in order to reduce the range and precision. A discussion of this is deferred until later. The predefined operations that can be performed on numeric types are much as one would expect. They are summarised in the table below.

Table 1.1. Floating-Point Operators

Operator	**Description**
`+ -`	These are either unary operators (that is, taking a single operand) or binary operators taking two operands. In the case of a unary operator, the operand can be of any numeric type; the result will be of the same type. Unary + effectively does nothing. Unary `-` changes the sign. In the case of a binary operator, both operands must have the same numeric base type; the result will be of that type. Normal addition or subtraction is performed.
`*`	Multiplication; both operands must be of some numeric type. If both operands have the same numeric base type, then the result will be of that type, otherwise the result will be the basis type of the operands.
`/`	Division; both operands must be of some numeric type. If both operands have the same numeric base type, then the result will be of that type, otherwise the result will be the basis type of the operands.
`rem`	Remainder; both operands must have the same integer base type and the result is of that type. It is the remainder on division.
`mod`	Modulo; both operands must have the same integer base type and the result is of that type. This is the mathematical modulo operation.
`abs`	Absolute value; this is a unary operator and the single operand may be of any numeric type. The result is of the same type and is the absolute value. That is, if the operand is positive, the result is the same but if it is negative, the result is the corresponding positive value.

Operator	Description
`**`	Exponentiation; this raises the first operand to the power of the second. Both operands must have the same numeric base type; the result will be of that type.

In addition, we can perform the operations =, `/=`, <, <=, > and >= in order to return a Boolean result `true` or `false`. Again, both operands must have the same base type.

Note

The form of the not equals operator is `/=`.

Numeric types follow the general rules of types and subtypes. It is not possible to add an integer value to a real value; both must have the same base type. A change of type from integer to real or visa versa can be done by using the desired type name (or indeed subtype name) followed by the expression to be converted in brackets. So given:

```
  i : integer := 8;
  j : integer;
  r : real := 5.4;
  s : real;
```

we cannot write:

```
  s := i + r;        // illegal
```

but we must write:

```
  s := real(i) + r;
```

which uses real addition to give `s` the value `13.4` ,or:

```
  j := i + integer(r);
```

which uses integer addition to give `j` the value `13`.

Conversion from real to integer always rounds rather than truncates, therefore 3.2 becomes 3 and 1,8 becomes 2. A value midway between two integers, such as 9.5, may be rounded up or down according to the implementation.

We conclude this section with a brief discussion on combining operators in an expression. As is usual the operators have different precedence levels and the natural precedence can be overruled by the use of brackets. Operators of the same precedence are applied in order from left to right. A sub expression in brackets obviously has to be evaluated before it can be used. The precedence levels of the operators we have met so far are shown below in increasing order of precedence:

`= /=`

`< <= > >=`

`+ -` (binary)

`* / mod rem **`

`+ - abs` (unary)

1.6. Enumeration Types

Here are some examples of declarations of enumeration types starting with colour_type, which we introduced when discussing types in general.

```
type colour_type is enum (BLUE, RED, GREEN);
type signal_colour_type is enum (RED, AMBER, GREEN);
```

This introduces an example of overloading. The literal `RED` can represent a colour_type or a signal_colour_type. Both meanings of the same name are visible together and the second declaration does not hide the first. We can usually tell which is meant from the context but in those odd cases when we cannot we can always qualify the literal by preceding it by an appropriate type mark (that is its type name or a relevant subtype name) and a dot. Therefore:

```
  declare
    p : colour_type;
    f : signal_colour_type;
  begin
    p := colour_type.RED;
    f := signal_colour_type.RED;
  end;
```

There is no upper limit on the number of values in an enumeration type, but there must be at least one; an empty enumeration type is not allowed. Constraints on enumeration types and subtypes are much as for integer numeric types. So we can write:

```
  subtype week_day_type is day_type range MON .. FRI;
```

and then we know that any entity declared of type week_day_type cannot be `SAT` or `SUN`. There are built-in characteristics to give the successor or predecessor of an enumeration value. These consist of succ and pred following an entity (of the type) and a prime (’). Therefore given:

```
  declare
    d : day_type := WED;
    f : signal_colour_type := AMBER;
  begin
    d := d'succ;
    f := f'pred;
  end;
```

the values of d and f will be THU and RED respectively. If we try to take the predecessor of the first value or the successor of the last then an exception is raised.

Another characteristic is pos. This gives the positional number of the enumeration value that is the position in the declaration with the first one having a position number of one. So:

```
  declare
    d : day_type := SAT;
    i : integer;
  begin
    i := d'pos;
  end;
```

will result in i having the value 6.

Finally, a characteristic called image is provided that will take an enum value and return its text representation. Therefore, we could write the following:

```
  declare
    d : day_type := SAT;
    s : string;
  begin
    s := d'image;
  end;
```

and s would contain the string “SAT”. There is currently no enum characteristic for the reverse direction (string => enum value).

It should be noted that these characteristics succ, pred, pos and image may also be applied to subtypes but they are identical to the same characteristics of the corresponding base type. Finally, the operators =, /=, <, <=, > and >= also apply to enumeration types. The result is defined by the order of the values in the type declaration. So we could write:

```
  declare
    b : boolean;
  begin
    b := BLUE < GREEN;
    b := WED >= THU;
  end;
```

and the value of b will first be set to `true` and then `false`.

1.7. The Boolean Type

The boolean type is a predefined enumeration type whose declaration is considered to be:

```
  type boolean is enum (false, true);
```

Boolean values are used in constructions such as the if statement that we will meet later. Boolean values are produced by the operators =, /=, <, <=, > and >= which have their expected meaning and apply to many types. So we can write constructions such as:

```
  declare
    today : day_type;
    tomorrow : day_type;
  begin
    if today = SUN then
      tomorrow := MON;
    end if;
  end;
```

The equality operators = and /= apply to values of the boolean type and have the obvious meaning. The boolean type also has other operators which are as follows :

Table 1.2. boolean Type Operators

Operator	Description
`not`	This is an unary operator and changes `true` to `false` and vice versa.
`and`	This is a binary operator. The result is `true` if both operands are `true`, and `false` otherwise.
`or`	This is a binary operator. The result is `true` if one or other or both operands are `true`, and `false` only if they are both `false`.

Operator	**Description**
`xor`	This is a binary operator. The result is `true` if one or other operand but not both are `true`. (Hence the name eXclusive OR). Another way of looking as it is to note that the result is `true` if and only if the operands are different.

In the case of `and` and `or` the left hand operand is always evaluated first and the right hand operand is only evaluated if it is necessary in order to determine the result. So in:

```
X and Y
```

X is evaluated first. If X is false, the answer is false whatever the value of Y and so Y is not evaluated. If X is true, Y has to be evaluated and the value of Y is the answer. Similarly in:

```
X or Y
```

X is evaluated first. If X is true, the answer is true whatever the value of Y and so Y is not evaluated. If X is false, Y has to be evaluated and the value of Y is the answer.

The precedences of `and`, `xor` and `or` are in that order (from high to low) and lower than that of any other operator. In particular they are of lower precedence than the relational operators =, /=, <, <=, > and >=.

In particular the precedence of `not` is higher than `and`, `or` and `xor` and so :

```
not A or B
```

means

```
(not A) or B
```

Boolean variables and constants can be declared and manipulated in the usual way.

```
declare
  danger : boolean;
  signal : colour_type;
begin
  danger := (signal = RED);
end;
```

The variable `danger` is then `true` if the `signal` is `RED`. We can then write:

```
  if danger then
    // …
  end if;
```

1.8. Characters and Strings

There are two predefined character types; character and wcharacter. The first has values, which are 8-bit quantities and encode a single-byte character. The second en-codes wide characters from any character set.

A character literal consists of a single character within a pair of single quotes. Examples of simple character literals include:

```
'a', 'G'
```

More complex examples include:

```
'\'', '\065', '\u0074'
```

The first shows how the single quote character is represented. The second uses an octal escape to represent the ASCII character ’5’. The last one uses a Unicode escape to represent the ASCII character ’`t`’.

We can perform the operations =, `/=`, <=, > and >= on characters in order to return a boolean result `true` or `false`. Both operands must have the same base type. The result of the comparison is based upon the lexicographic order of characters. The following are all true:

```
  'r' = 'r'
  'c' /= 'C'
  'd' < 'n'
  'p' > 's'
  'k' <= 'k'
  't' >= 'h'
```

There are also two predefined string types; string and wstring. The string type is similar to a sequence of characters . The wstring type is similar to a sequence of wcharacters. General sequences are discussed in Section 3.2.3, “Sequences”.

A typical string declaration might be:

```
  s : string := "Hello World!";
```

This declaration uses a string literal to give the string an initial value. A string literal consists of zero or more characters enclosed in double quotes.

Example 1.1. Example String Literals

"A simple string"

"This has \0163ome \"strange\" char\u0061cter’s in it!"

Like characters, we can perform the operations =, `/=`, <=, > and >= on strings. The result of the comparison is based upon the lexicographic order of characters. The following are all true:

```
"gnu" = "gnu"
"gnu" /= "gNu"
"cat" < "dog"
"cattle" > "cat"
"azure" <= "baboon"
"aadvark" >= ""
```

Strings can be concatenated together using the operator &. It has the same precedence as binary plus. The two operands must have the same base type and the result is a string whose value is obtained by juxtaposing the two operands. So after:

```
s := "cab" & "bage";
```

the value of s will be "cabbage". One or other of the operands of & can also be a single character.

```
s := "dais" & 'y';
s := 'd' & "anger";
```

After the first assignment the value of s will be "`daisy`" and after the second "`danger`". The individual characters of a string can be referred to following the string name with an expression in brackets giving a value in the range of indices of the string. If this expression, known as the index value, has a value outside of the range, then an exception will be raised.

One way of obtaining a valid index value is to use one of the two characteristics `first` or `last`, which give the lower, and upper bounds of the indices of a string.

```
declare
  first_postion : integer;
  last_postion  : integer;
  s : string := “Hello World!”;
begin
  first_position := s'first;
  last_postion := s'last;
end;
```

We could also set the first character of a string to the character ’J’ by writing:

```
  s[s'first] := ’J’;
```

If a string is empty then first and last will both raise an exception.

Another characteristic of a string is length, which gives the number of characters in the string. So:

```
  declare
    s : string := "Hello World!";
    l : integer;
  begin
    l := s'length;
  end;
```

would result in l having the value 12.

One common thing to do is to loop over all the characters in a string. This can be done in two different ways. The first uses a for statement together with the two characteristics `first` and `last` to loop through the index values of a string. The `for` statement is discussed in Section 2.3, “ Loop Statements”.

1.9. Bytes

The byte type is a predefined type, it is an 8-bit quantity that is guaranteed not to undergo any conversion when transmitted by a communication system. We can perform the operations = and /= on bytes to test for equality or inequality.

1.10. Date and Time

MASL incorporates a type call time_type, which is used to store a current point in time, the implementation of this type is defined by the specific architecture in use. As this type represents a current point in time it represents not just a time (hours, minutes and seconds) but also a date (day, month and year). The date and time elements that can be extracted from this type are represented by two structures. The date is defined as a structure type called date_type and is shown below:

```
  type day_type is numeric range 1..31;
  type month_type is enum (JANUARY=1, FEBRUARY, MARCH, APRIL, MAY, JUNE,

  type year_type is numeric range 1970..2036;

  type date_type is structure
```

```
    day : day_type;
    month : month_type;
    year : year_type
  end structure;
```

The time is defined as a structure type called time_of_day_type and is shown below:

```
  type hour_type    is numeric range 0..23;
  type minute_type  is numeric range 0..59;
  type second_type  is numeric delta 0.1 range 0..59;

  type time_of_day_type is structure
    hour   : hour_type;
    minute : minute_type;
    second : second_type;
  end structure;
```

A series of architecture-defined services are provided to manipulate and convert the date and time types described, from one form to another (see [MASL06]). For example:

```
  declare
    previous_date : Calendar::date_type := (12,Calendar::JANUARY,2000);
    previous_time : Calendar::time_of_day_type;
    current_date  : Calendar::date_type;
    current_time  : Calendar::time_of_day_type;
    current_time_point : Calendar::time_type;

  begin
    previous_time := (12,30,00);
    Calendar::get_clock(current_time_point);
    Calendar::split(current_time_point,current_date,current_time);
  end;
```

The usual relational operators =, !=, <,> etc. are available for operations on variables of type time_type, but as the date_type and time_of_day_type are structures these relational operators are not allowed. Therefore:

```
  declare
    previous_date : Calendar::date_type := (12,Calendar::JANUARY,2000);
    current_date  : Calendar::date_type := (14,Calendar::JANUARY,2000);
  begin
    if previous_date = current_date then        // illegal := not availab
```

```
      // …
    end if;
  end;
```

is illegal. To achieve the desired operation the example MASL code below would be required:

```
  declare
    previous_date : Calendar::date_type := (12,Calendar::JANUARY,2000);
    current_date  : Calendar::date_type := (14,Calendar::JANUARY,2000);
  begin
    if previous_date.day = current_date.day and
      previous_date.month = current_date.month and
      previous_date.year = current_date.year then
      // …
    end if;
  end;
```

1.11. Expression Summary

All the operators introduced so far are shown in Table 1.3, “Operators” , grouped by precedence level. In all the cases of binary operators, except for multiplication (`*`) and division (`/`), the two operands must have same base type. For multiplication and division, the two operands must have the same basis type.

Table 1.3. Operators

Operator	Operation	Operand(s)	Result
`or`	inclusive or	boolean	boolean
`xor`	exclusive or	boolean	boolean
`and`	conjunction	boolean	boolean
`=`	equality	boolean, numeric, enumeration, character, wcharacter , string, wstring, byte	boolean
`/=`	inequality	boolean, numeric, enumeration, character, wcharacter , string, wstring, byte	boolean

Operator	Operation	Operand(s)	Result
<	less than	numeric, enumeration, character, wcharacter , string, wstring	boolean
>	greater than	numeric, enumeration, character, wcharacter , string, wstring	boolean
<=	less than or equal	numeric, enumeration, character, wcharacter , string, wstring	boolean
>=	greater than or equal	numeric, enumeration, character, wcharacter , string, wstring	boolean
+	addition	numeric	*same*

2. Execution Control

This chapter describes the three sequential control structures of MASL. These are the `if` statement, the `case` statement and the `loop` statements.

2.1. `if` Statement

The `if` statement is probably the most basic way to control program flow. The simplest form of `if` statement starts with the reserved word if followed by a condition and the reserved word then. This is then followed by a sequence of statements, which will be executed if the condition turns out to the true. The end of the sequence is indicated by the closing `end if`. The condition can, be of arbitrary complexity and the sequence of statements can be of arbitrary length. A simple example is:

```
declare
  danger : Boolean := false;
begin
  if danger then
    run();
  end if;
end;
```

In this, `danger` is a boolean variable and run is a service describing the details of the running activity. The statement `run();` merely calls the service (services are dealt with in Chapter 6, *Actions*). The effect of this if statement is that should the variable `danger` be true then we invoke the service run, otherwise we do nothing. In either case we then obey the statement following the `if` statement.
As we have said there could be a long sequence between `then` and `end if`. Therefoer we might breeak the process into more detail.

```
declare
  danger : boolean;
begin
  if danger then
    think();
    discuss();
    run();
  end if;
end;
```

Note how we indent the statements to show the flow structure of the code. This is most important since it enables the program to be understood so much more easily also be aware that the end if will always be followed by a semicolon. This is because semicolons terminate statements rather than separate them as in other languages.

Often we will want to do alternative actions according to the value of the condition. In this case we add `else` followed by the alternative sequence to be obeyed if the condition is false.

```
  declare
    today : day_type;
    tomorrow : day_type;
  begin
    if today = SUN then
      tomorrow := MON;
    else
      tomorrow := today'succ;
   end if;
end;
```

The statements in the sequences after then and else can be quite arbitrary and so could be further nested `if` statements.

```
  declare
    a : real; b : real;
    c : real; d : real;
    root : real;
  begin
    if a = 0.0 then
      root := -c/b;
    else
      if b**2 - 4.0*a*c >= 0.0 then
        solveRealRoots();
      else
        solveComplexRoots();
      end if;
    end if;
  end;
```

Observe the repetition of `end if`. This is rather ugly and occurs sufficiently frequently to justify an additional construction. This uses the reserved word `elsif` as follows:

```
  declare
    a : real; b : real;
    c : real; d : real;
    root : real;
  begin
```

```
    if a = 0.0 then
      root := -c/b;
    elsif b**2 - 4.0*a*c >= 0.0 then
      solveRealRoots();
    else
      solveComplexRoots();
    end if;
  end;
```

This construction emphasised the essentially equal status of the three cases and also the sequential nature of the tests. The `elsif` part can be repeated an arbitrary number of times and the final else part is optional. The behaviour is simply that each condition is evaluated in turn until one that is true is encounted; the corresponding sequence is then obeyed. In none of the conditions turns out to be true then the else part, if any, is taken; if there is no else part then none of the sequences are obeyed.

Note

The spelling of `elsif` and the layout – we align `elsif` and `else` with the `if` and `end if` and all the sequences are indented equally.

2.2. case Statement

A `case` statement allows a choice between several possible sequences of statements according to the value of an expression. For instance, an example could be:

```
  declare
    signal : signal_colour_type;
  begin
    case signal is
      when RED =>
        stop();
      when AMBER =>
        null;
      when GREEN =>
        go();
    end case;
  end;
```

All possible values of the expression must be provided for in order to guard against accidental omissions. If, as in this example, no action is required for one or more values then the `null` statement can to be used.

The null statement, written :

```
  null;
```

does absolutely nothing but its presence indicates that we truly want to do nothing. The sequence of statements here, as in the if statement, must contain at least one statement. It often happens that the same action is desired for several values of the expression. Consider the following:

```
  declare
    key : character;
  begin
    key := 's';
    case key is
      when 'a' | 'b' | 'c' | 'd' =>
        output(key);
      when 'e' =>
        output(key);  process(key);
      when 's' =>
        ignore(key);
    end case;
  end;
```

The alternative values are separated by the vertical bar character (`|`). Note again the use of the `null` statement. Another example, shown below, demonstrates the use of the others key word, this is used to indicate that all other possibilities are covered by this case (`SAT` and `SUN`).

```
  declare
    today : day_type;
  begin
    case today is
      when MON | TUE | WED | THU =>
        work();
      when FRI =>
        work(); party();
      when others =>
        null;
    end case;
  end;
```

There are several restrictions in the use of `case` statements. One is that if we use `others` then it must appear alone and as the last alternative. As already stated it covers all values not explicitly covered by the previous alternatives. It should also be noted that there is an implied break at the end of each when block; the execution cannot fall through to subsequent case statements as in C/C++. Another very important restriction is that all the choices must be constant so that they can be evaluated at compilation time. Finally, all possible values of the expression after the `case` must be provided for. This usually means all values of the type of the expression after the reserved word `case`.

2.3. Loop Statements

MASL has two forms of loop statement; the `while` statement and the `for` statement. Both repeat a sequence of statements. The `while` statement starts with the reserved word while followed by a condition and the reserved word `loop`. This is then followed by a sequence of statements, which will be executed at each iteration of the loop. The end of the sequence is indicated by the closing `end loop`.A simple example is:

```
declare
  n : integer;
  i : integer;
  term : real;
  e : real;
begin
  while i /= n loop
    i :=i+1;
    term := term / i;
    e := e + term;
  end loop;
end;
```

At each iteration of the loop, the condition is tested, in this case `i /=`n, and if true the sequence of statements is executed. If the condition is false then the loop terminates at once and control passes to the point immediately after end loop. Note that if the condition is false the first time it is evaluated, then the sequence of statements is never executed.

Sometimes it is necessary to break out of a loop prematurely. We can do this with the `exit` statement:

```
exit;
```

If this is executed inside a loop then the loop terminates at once and control passes to the point immediately after the end loop. Suppose we decide to stop when term is equal to zero. We can do this as follows:

```
  declare
    n : integer; i : integer;
    term : real; e : real;
  begin
    while i /= n loop
      i :=i+1;
      term := term / i;
      if term = 0.0 then
        exit;
      end if;
      e := e + term;
    end loop;
  end;
```

The construction:

```
  if condition then
    exit;
  end if;
```

is so common that a special shorthand is provided `exit` when condition. So we could have written the loop as:

```
  declare
    n : integer; i : integer;
    term : real; e : real;
  begin
    while i /= n loop
      i :=i+1;
      term := term / i;
      exit when term = 0.0;
      e := e + term;
    end loop;
  end;
```

The `for` statement allows a specific number of iterations with a loop parameter taking in turn all the values of a discrete range. Our example could be recast as:

```
  declare
    n : integer;
    term : real;
    e : real;
  begin
```

```
    for i in 1 .. n loop
      term := term / i;
      e := e + term;
    end loop;
  end;
```

where i takes the values 1, 2, 3, ... n. The variable i is declared by its appearance in the iteration scheme and does not have to be declared outside of the for statement. It takes its type from the discrete range and within the loop behaves as a constant so that it cannot be changed except by the loop mechanism itself. When we leave the loop (by whatever means) i ceases to exist (because it was declared by the loop) and so we cannot read its final value from outside.

The values of the discrete range are normally taken in ascending order. Descending order can be specified by writing:

```
for i in reverse 1 .. n loop
```

but the range itself is always written in ascending order. It is not possible to specify a numeric step size other than one. This should not be a problem since the vast majority of loops go up by steps of one and almost all the rest go down by steps of one. The very few which do behave otherwise can be explicitly programmed using the while form of loop.

The range can be null (this would happen if n was zero or negative in our example) in which case the sequence of statements will not be executed at all. Of course, the range itself is evaluated only once and cannot be changed inside the loop. Thus declare:

```
    n : integer;
  begin
    n := 4;
    for i in 1 .. n loop
      // …
      n := 10;
    end loop;
  end;
```

results in the loop being executed just four times despite the fact that the value of n is changed to ten.

Our examples have all shown the lower bound of the range being 1. This, of course, need not be the case. Both bounds can be arbitrary dynamically evaluated expressions. Furthermore the loop parameter need not be of integer type. We could, for instance, simulate a week’s activity by:

```
  for today in MON .. SUN loop
    // …
  end loop;
```

This declares today to be of type day and obeys the loop with the values `MON`, `TUE`, … SUN in turn. Other forms of discrete range are of advantage here. The essence of `MON .. SUN` is that it embraces all the values of type day. It is therefore better to write the loop using a form of discrete range that conveys the idea of completeness. For example:

```
  for today in day_type'range loop
    // …
  end loop;
```

As already stated, the type of the loop parameter is determined from the type of the discrete range after the reserved word `in`. It is therefore necessary for the type of the discrete range to be unambiguous in the `for` statement. This is usually the case but if we had two enumeration types with two overloaded literals such as:

```
  type colour_type is enum (BLUE, RED, GREEN);
  type signal_colour_type is enum (RED, AMBER, GREEN);
```

then:

```
for s in RED .. GREEN loop
```

would be ambiguous. We could resolve the problem be qualifying either of the expressions. For example:

```
for s in signal_colour_type.RED .. GREEN loop
```

3. Composite Types

In this section we describe the composite types, these are either structures or collections.

3.1. Structures

A structure is a composite entity consisting of named components which may be of different types. Consider the following:

```
  subtype day_number_type is integer range 1 .. 31;
  type month_name_type is enum (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, S

  type date is structure
    day : day_number_type;
    month : month_name_type;
    year : integer;
  end structure;
```

This declares the type date to be a structure containing three named components: `day`, `month` and `year`. We can declare variables and constants of structure types in the usual way.

```
  d : date;
```

declares a variable d which is a date. The individual components of d can be denoted by following d with a dot and the component name. Therefore we could write:

```
  d.day := 20;
  d.month := SEP;
  d.year := 1969;
```

in order to assign new values to the individual components. Structures can be manipulated as whole entities. Literal values can be written as aggregates. So we could write:

```
  d : date := (20, SEP, 1969);
  e : date;
```

and then:

```
  e := d;
```

or:

```
  e := (20, SEP, 1969);
```

It is possible to give default expression for some or all of the components in the type definition. Therefore:

```
  type complex_type is structure
    real : integer := 0;
    image : integer := 0;
  end structure;
```

declares a structure type containing two components of type integer and gives a default expression of 0 for each. The components of a structure can be of any type except those in terms of some instance type [MASL06].

It is important to note that variables declared as a structure type cannot be subsequently used in statements that incorporate the comparison operators.

3.2. Collections

A collection is a composite entity consisting of a number of elements all of the same type. MASL has four types of collections; sets, bags, sequences and arrays, each of which have different behaviours.

3.2.1. Sets

A set consists of a number of elements. The elements of a set are unordered. Sets are used to group elements together when there is no need for any particular order for the elements. Beside lacking order, sets do not allow multiple occurrences of the same element. This means that inserting a value that is already an element of a set leaves the set unchanged.

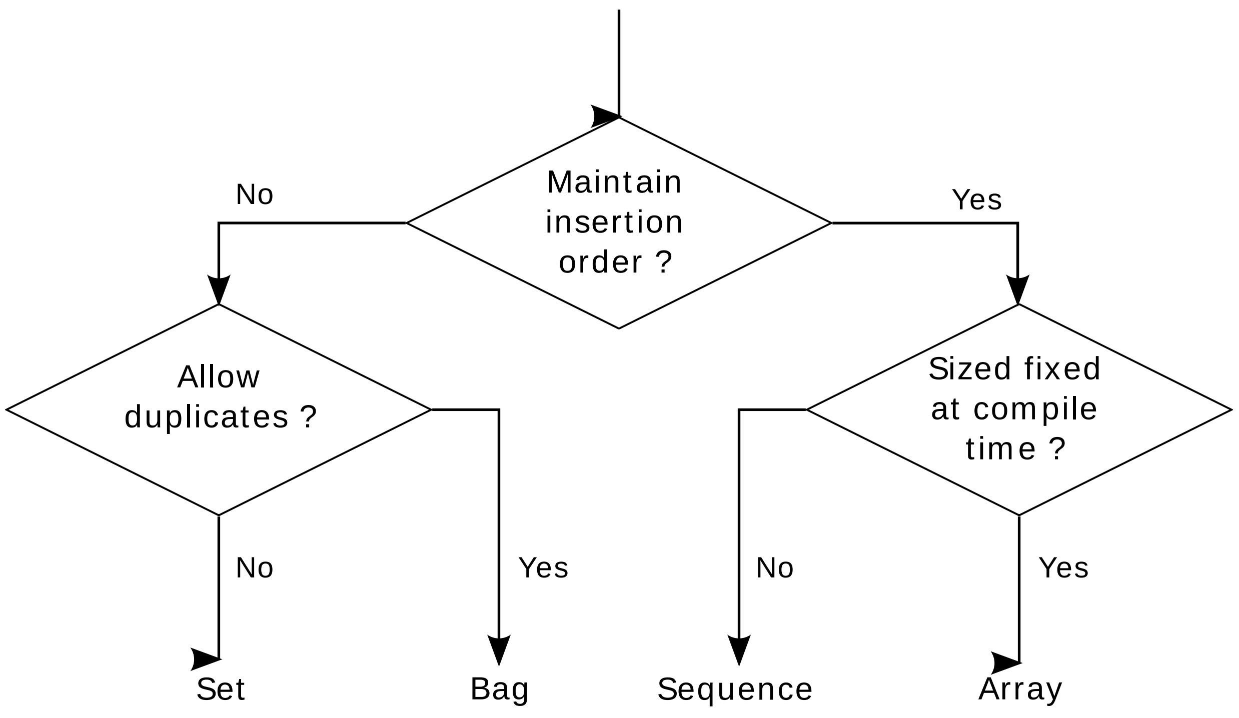
A bag is a collection that keeps track of what it's elements are and also the number of occurrences of each element. In other words, bags allow duplicate elements. Like sets,bags are unordered.

A sequence is a collection that associates a position with each element based on insertion order. sequences allow duplicates.

An array is like a sequence except that the number of elements in the array is fixed at compile time.

Figure 3.1, “Decision Tree for Collection Types” gives a simple decision tree to help choose the collection type to suit a particular behaviour requirement.

Figure 3.1. Decision Tree for Collection Types



After the first assignment, s will contain the single element 15. After the second assignment, s will contain the three elements 15, 31 and 57. After the third assignment, s will still contain the three elements 15, 31 and 57. Notice that duplicates are removed and order is not preserved.

If we now declare:

```
  r : set of integer;
  t : set of integer;
```

then:

```
  t := t & 31 & 44  & 15;
  r := s & t;
```

will result in r containing the elements 15, 31, 44 and 57. Sets have various characteristics. length gives the number of elements in a set. So using our current example:

```
r'length = 4
s'length = 3
t'length = 3
```

In order to loop over all the elements of a set we can use a for statement together with the characteristic elements. For example:

```
  for i in r'elements loop
    // …
  end loop;
```

This declares `i` to be of type integer and executes the loop with the values of the elements of `r` in turn. Since the elements in a set are unordered, the order in which the elements are taken is arbitrary. In our example, `i` might take the values 44, 31, 15 and 57 in turn.

Set types have four other operators; `union`, `intersection`, `not_in` and `disunion`. These are all binary operators which take operands of the same set base type and whose result is of that type. They correspond to the notions of set union, set intersection, set subtraction and set disunion respectively. Union and not_in have the same precedence as binary plus, intersection and disunion have the same precedence as multiplication.

Note

For set types, & is equivalent to `union`.

3.2.2. Bags

A typical bag declaration might be:

```
  b : bag of integer;
```

This declares b to be a variable which has a number of elements, each of which is of type integer. Like sets, bags can be concatenated together using the operator &. Again it has the same precedence as binary plus. The two operands must be of the same base type and the result is a bag of that type whose value contains all the elements of the two bags. One or other of the operands of & can also be a single value of the element type. So, for example:

```
  b := b & 15;
  b := 31 & b & 57;
  b := b & 31;
```

After the first assignment, b will contain the single element 15. After the second assignment, b will contain the three elements 15, 31 and 57. After the third assignment, b will contain the elements 15, 31 twice and 57. Notice that unlike `sets`, duplicates are not removed.

If we now declare:

```
  a : bag of integer;
  c : bag of integer;
```

then:

```
  c := c & 31 & 44 & 15;
  a := b & c;
```

will result in a containing the elements 15 twice, 31 three times, 44 once and 57 once. Bags, like sets, have various characteristics. `length` gives the number of elements in a bag. So using our current example:

```
a’length = 7
b’length = 4
c’length = 3
```

We can loop over all the elements in a bag in the same way as we did with sets by using a `for` statement together with the characteristic elements.

```
  for i in a’elements loop
    // …
  end loop;
```

This declares `i` to be of type integer and executes the loop with the values of the elements of a in turn. Again, like a set, the order in which the elements are taken is arbitrary. If a bag contains duplicate elements then the loop will be executed for each of these duplicate elements. In our example, `i` might take the values 15, 31, 44, 31, 15, 57 and 31 in turn.

3.2.3. Sequences

A typical sequence declaration might be:

```
  q : sequence of integer;
```

This declares q to be a variable which has a number of elements, each of which is of type integer. Like sets and bags, sequences can be concatenated together using the operator &. Again it has the same precedence as binary plus. The two operands must be of the same type and the result is a sequence of the same type whose value contains all the elements of the two sequences. In addition, the order of the elements in the resultant sequence is obtained by juxtaposing the two operands. One or other of the operands of & can also be a single value of the element type.

So, for example:

```
  q := q & 15;
  q := 31 & q & 57;
  q := q & 31;
```

After the first assignment, q will contain the single element 15. After the second assignment, q will contain the three elements 31, 15 and 57 in that order. After the third assignment, q will contain the elements 31, 15, 57 and 31 in that order.

If we now declare:

```
  n : sequence of integer;
  p : sequence of integer;
```

then:

```
  p := p & 31 & 44 & 15;
  n := p & q;
```

will result in n containing the elements 31, 44, 15, 31, 15, 57 and 31 in that order. sequences, like sets and bags, have various characteristics. `length` gives the number elements in a sequence. So using our current example:

```
 n’length = 7
 p’length = 3
 q’length = 4
```

We can loop over all the elements in a sequence in the same way as we did with sets and bags by using a for statement together with the characteristic elements.

```
  for i in n’elements loop
    // …
  end loop;
```

This declares `i` to be of type integer and executes the loop with the values of the elements of n in turn. Since the elements in a sequence are ordered, the order in which the elements are taken is given by that order. In our example, i would take the values 31, 44, 15, 31, 15, 57 and 31 in that order.

The individual elements of a sequence can be referred to by following the sequence name with an expression in brackets giving a value in the range of indices of the sequence. If this expression, known as the index value, has a value outside of the range, then an exception will be raised.

One way of obtaining a valid index value is to use one of the two characteristics first or last, which give the lower and upper bounds of the indices of q. For example, we could set the first element of a sequence to zero by writing:

```
  q[q'first] := 0;
```

If a sequence is empty both first and last will raise an exception. Using first and last we could loop though the index values of a sequence.

```
  for i in n'first .. n'last loop
    q[i] := 0;
  end loop;
```

The expression q'first .. q'last is so common that a shorthand q'range is provided. Hence we could have written the previous loop as:

```
  for i in n'range loop
    q[i] := 0;
  end loop;
```

Finally, we can denote a slice of a sequence by following the sequence name with a discrete range in brackets. So:

```
  q := n[n'first + 2 .. n'last - 2];
```

results in q containing the elements 15, 31 and 15 in that order.

3.2.4. Arrays

A typical array declaration might be:

```
  a : array (1 .. 6) of  integer;
```

This declares a to be a variable which has a six elements, each of which is of type integer. The range of an array can be in terms of any discrete type that is any integer numeric type or any enumeration type. For example, given:

```
  type day_type is enum (MON, TUE, WED, THU, FRI, SAT, SUN);
```

we could declare an array to have elements for each day of the week by writing:

```
  hours_worked : array (MON .. SUN) of integer;
```

or better:

```
  hours_worked : array (day_type’range) of integer;
```

Since arrays have a fixed size they cannot be concatenated together like bags, sets and sequences. Arrays, like all the other collection types, have various characteristics. Length gives the number of elements in an array. So using our current example:

```
a’length = 6
```

Note that because arrays have a fixed size this value will never change.

We can loop over all the elements in an array in the same way as we did with sequences by using a `for` statement together with the characteristic elements.

```
  for i in a’elements loop
    // …
  end loop;
```

This declares `i` to be of type integer and executes the loop with the values of the elements of a in turn. Since the elements in an array are ordered, the order in which the elements are taken is given by that order.

Like sequences, the individual elements of an array can be referred to by following the array name with an expression in brackets giving a value in the range of indices of the array. If this expression, known as the index value, has a value outside of the range, then an exception will be raised.

One way of obtaining a valid index value is to use one of the two characteristics `first` or `last` which give the lower and upper bounds of the indices of a. For example we could set the first element of an array to zero by writing:

```
  a[a'first] := 0;.
```

Since we explicitly define the upper and lower bounds of the indices of an array in its declaration, we could have equally done this by using an explicit value in this range.

```
  a[1] := 0;
  hours_worked[WED] := 0;
```

Using `first` and `last` we could loop though the index values of an array.

```
  for i in a'first .. a'last loop
    a[i] := 0;
  end loop;
```

Again, as for sequences, we could have used the shorthand `a'range` instead of `a'first.. a'last`.

```
  for i in a'range loop
    a[i] := 0;
  end loop;
```

Finally, we can denote a slice of an array by following the array name with a discrete range in brackets. So:

```
  declare
    q : sequence of integer;
  begin
    q := a[3 .. 5];
  end;
```

results in q containing the elements at indices 3, 4 and 5 of the array a in that order. Notice that taking a slice of an array results in a sequence of the same element type.

3.3. Collection Types

The collections we introduced in the last section did not have an explicit type name. They were in fact of anonymous type. Reconsidering the example of the set, we could write:

```
  type integer_set_type is set of integer;
```

and then declare `s` using the type name in the usual way:

```
  s : integer_set_type;
```

Whether or not we introduce a type name for particular collection types depends very much on the abstract view of each situation. If we are thinking of a collection as a complete entity in its own right then we should use a type name.

3.4. Assigning Collections to other Collections

Values of a collection type can be converted to a different collection type with the same element type by using an explicit type conversion. This type conversion may raise an exception if the original value cannot be converted into a value of the new type.

For example, if we declare:

```
  s : set of T;
  b : bag of T;
  q : sequence of T;
  a : array (1 .. 10) of T;
```

Then the following is perfectly legal from the syntactic point of view.

`s := set of T(b);`	Raises an exception if there are duplicate elements in the bag.
`s := set of T(q);`	Raises an exception if there are duplicate elements in the sequence.
`s := set of T(a);`	Raises an exception if there are duplicate elements in the array.
`b := bag of T(s);`	Never raises an exception.
`b := bag of T(q);`	Never raises an exception.
`b := bag of T(a);`	Never raises an exception.
`q := sequence of T(s);`	Never raises an exception. The order of the elements in the sequence will be arbitrary.
`q := sequence of T(b);`	Never raises an exception. The order of the elements in the sequence will be arbitrary.
`q := sequence of T(a);`	Never raises an exception. The order of the elements in the sequence will be the same as the array.
`a := array (1 .. 10) of T(s);`	Raises an exception if the number of elements in the set is not the same as the size of the array. The order of the elements in the sequence will be arbitrary.
`a := array (1 .. 10) of T(b);`	Raises an exception if the number of elements in the bag is not the same as the size of the array. The order of the elements in the sequence will be arbitrary.

`a := array (1 .. 10) of T(q);`	Raises an exception if the number of elements in the sequence is not the same as the size of the array. The order of the elements in the array will be the same as the sequence.

As you can see converting bags, sequences and arrays to sets will raise an exception if there are duplicate elements in the collection. It is often the case that we want to take a collection and remove all the duplicates without raising an exception. To do this the characteristic `get_unique` is provided for bags, sequences and arrays.

```
  s := b'get_unique;
  s := q'get_unique;
  s := a'get_unique;
```

Now consider a subtype S of type T. A collection of S is a subtype of a collection of T.

For example, if we declare:

```
  sb : bag of S;
  ss : set of S;
  sq : sequence of S;
  sa : array (1 .. 10) of S;
```

he following is perfectly legal from the syntactic point of view.

```
  sb := b;
  ss := s;
  sq := q;
  sa := a;
```

Of course, on execution, the elements of a collection of type T may not satisfy the constraints of subtype S, in which case an exception is raised. Assignment if the other direction:

```
  b := sb;
  s := ss;
  q := sq;
  a := sa;
```

will, of course, always work. Finally, conversion between arrays is legal but only if the following conditions are all met:

- The types of the array are the same or one type is a subtype of another.
- The array size is not only the same but the indices are in the same range.

Hence, if we declare:

```
  subtype X is T;
  aa : array (1 .. 10) of T;
  ab : array (1 .. 10) of T;
  ac : array (5 .. 10) of T;
  ad : array (1 .. 10) of X;
  ae : array (10 .. 20) of T;
  af : array (1 .. 10) of Y;
```

Then the following are legal:

```
  aa := ab;
  ac := aa[5..10];
  aa := ad;
```

and the following are illegal:

```
  aa := ac;                // illegal : different array size
  aa := af;                // illegal : different types
  aa := ae;                // illegal : different indices
  ab := ae[10 .. 20];      // illegal : different size and different indi
```

3.5. Summary

All the operators introduced in this chapter are shown in Table 3.1, “More Operators” grouped by precedence level. In all the cases of binary operators, the two operands must be of the same base type.

Table 3.1. More Operators

Operator	**Operation**	**Operand(s)**	**Result**
&	collection concatenation	set, bag or sequence	same
`union`	set union	set, bag or sequence	same
`not_in`	set subtraction	set, bag or sequence	same

Operator	**Operation**	**Operand(s)**	**Result**
`intersection`	set intersection	set	same
`disunion`	set disunion	set, bag or sequence	same

4. Exceptions

MASL, like many other programming languages, provides an exception-handling mechanism for reporting and processing run-time error conditions during the execution of an application. This also means that the error handling code blocks are explicitly separated from the core business logic of the application.

When a MASL program violates the semantic constraints of the language, i.e. navigation via a null instance handle or indexing outside the bounds of an array, a run-time exception is raised by the architecture to signal the violation to the application. The exception will subsequently propagate up the call stack, starting from the point where it occurred, until one of two things happens. Either the exception is handled by a user defined exception block or if none are found, by the architectures own internal exception handling code blocks; these will determine the severity of the exception and either terminate or continue the execution of the application.

MASL provides several keywords for dealing with exceptions. A user-defined exception is declared using the `exception` keyword, while an exception is raised using the `raise` statement and handled using a combination of the `exception` and `when` statements. See Example 4.1, “Declaration, raising and handling of a user defined exception.”.

Example 4.1. Declaration, raising and handling of a user defined exception.

Declaration of an exception (defined within the type section of a mod file):

```
  exception my_exception;
```

Use and handling of my_exception:

```
  begin
    // undertake required operation(s)
    …
    // program determines application error encountered
    // raise a user defined exception.
    raise my_exception;

    // Exception handling code
    exception
      when my_exception =>
        // log some error and clean-up;
end;
```

MASL has several pre-defined `exception` types, that may be encountered during the execution of an application, they are listed below:

- constraint_error
- io_error
- IOP_error
- program_error
- relationship_error
- referential_access_error
- storage_error
- system_error

Just as a user-defined exception can be raised and caught by the application, the built in exception types can also be raised by the application and handled by a used defined code block. Putting it all together gives the following example:

```
file_open : boolean := false;

begin

  // Attempt to open a file
  Text_IO::open(…);
  if file_open = true then
    // undertake further file processing
    …
    // error encountered during processing
    raise my_exception;
  end if;

  Text_IO::close(…);

  exception
    when Standard::program_error =>
      // Architecture reported run-time error.

    when Standard::io_error =>
      // File IO error occurred, must have happened on the open
      // close the file. ( this does not throw)
      Text_IO::close(…);
```

```
        // propagate the error back up the application
        raise;

      when my_exception  =>
        Text_IO::close(…);
  end;
```

Table 4.1, “Exception Types”below describes each of the built in exception types and provides details on when and why these exceptions are raised by the architecture.

Table 4.1. Exception Types

Exception	Description	Example
`constraint_error`	Raised on breach of an architecture constraint.	• Out of bounds access to a container • `find_only` returns more than single instance • Any operation on a null instance; i.e. navigation, attribute access, etc.
`io_error`	Raised by a file access or file manipulation error.	Cannot open file.
`IOP_error`	• Raised as a wrapper for CORBA IOP errors. • Raised on a pack/ unpack error for MessageBus.	See CORBA specification.
`program_error`	• Raised on architecture and code-generator inconsistencies . • Raised on state model errors. • Raised as a wrapper for C++ exceptions.	• Cannot Happen within a State Model. • Creation events directed at wrong objects. • No active subtype for deferred polymorphic service.

Exception	Description	Example
	• Raised on certain polymorphic operations.	
`relationship_error`	Raised on relationship errors.	• Deleting object instance, which is still linked to other object instances(s). • Double link to same instance. • Unlink of object which is not currently linked.
`referential_access_error`	Raised on errors regarding referentials.	• Inconsistent values for merged referentials. • Reading referentials for non-existent relationship instances.
`storage_error`	Raised on allocation failure	Failure of create expression
`system_error`	Any system error condition affecting the executing model.	

5. Objects and Relationships

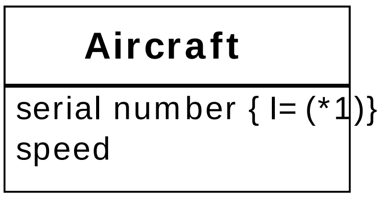
This chapter starts by describing how instances of objects are created followed by a description of how instances that have already been created are obtained, and how to delete instances.

We then move onto relationships and mirror the first part of this chapter describing how relationships are created followed by a description of navigating existing relationships, and how to delete relationships.

5.1. Creating Instances

An instance of an object is created by using a create expression. A create expression starts with the reserved word `create` followed by the name of an object and an object aggregate. The object aggregate is a list of attribute associations, each of which consists of a name of an attribute followed by => and then some expression giving the initial value, all enclosed by brackets.

Figure 5.1. Aircraft Object



Given the `Aircraft` object shown in Figure 5.1, “Aircraft Object” we might write:

```
  create Aircraft(serial_number => 76);
```

This creates a new Aircraft instance with a serial number of 76. The object aggregate must supply initial values for all non-referential preferred attributes; otherwise, a compile-time error will occur. If, at run-time, these initial values do not define a unique instance, object instance with same preferred id(s) already exists, then an exception will be raised.

If the `Aircraft` object was an active object, and hence instances of it had a `Current_State` attribute, then the initial state of an instance must also be set when it is created (this also includes creation states).

```
  create Aircraft(serial_number => 76, Current_State => parked);
```

This creates a new `Aircraft` instance with a serial number of `76`, in the initial state `parked`.

Having created an instance we might want to access its attributes. We can do this by assigning the created instance to a variable of the appropriate instance type. Each object within a model has a corresponding instance type. Variables of a specific instance type can refer to specific instances of the corresponding object. To declare a variable of the instance type of the `Aircraft` object we might write:

```
  aircraft_instance : instance of Aircraft;
```

We could then assign the created instance to this variable.

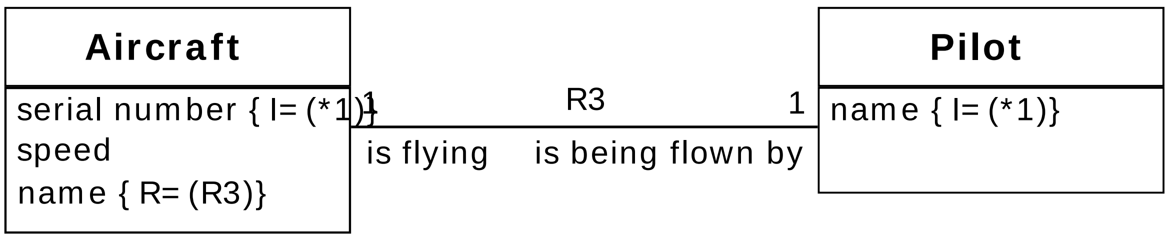
```
  declare
    aircraft_instance : instance of Aircraft;
  begin
    aircraft_instance := create Aircraft(serial_number => 76);
  end;
```

Attributes of an instance can be denoted by following the instance name with a dot and the attribute name. Therefore we might write:

```
  declare
    serial_number : integer;
  begin
    aircraft_instance := create Aircraft(serial_number => 76);
    serial_number := aircraft.serial_number;
    aircraft_instance.speed := 0;
  end;
```

Notice that we can both set and get the values of attributes. Certain types of attributes cannot have their values set in this way. If an attribute is either preferred or referential then its value cannot be set in this manner.

Figure 5.2. Relationship between Pilot and Aircraft



For example, given the relationship shown in Figure 5.2, “Relationship between Pilot and Aircraft” the following assignments are illegal:

```
  declare
    aircraft_instance : instance of Aircraft;
    pilot_instance    : instance of Pilot;
  begin
    aircraft_instance  := create Aircraft(serial_number => 76);
    pilot_instance     := create Pilot(name => “Peter Smith”);

    aircraft_instance.serial_number := 38;     // illegal : preferred id
    aircraft_instance.pilot_name := "Fred";    // illegal : referential c
  end;
```

5.2. Obtaining instances

In that last section we showed how we could obtain the instance produced by a `create` expression. If we want to obtain instances that have already been created we can use a `find` expression. There are four forms of find expression:

- `find`
- `find_all`
- `find_one`
- `find_only`

The `find` locates all the instances of an object satisfying a specified condition. This form starts with the reserved word `find` followed by the name of an object and a condition enclosed by brackets. Given the `Aircraft` object we might write:

```
  find Aircraft(speed < 100);
```

This finds all instances of the `Aircraft` object whose speed is less than `100`. The condition can of course, be of arbitrary complexity with the restriction that left-hand side of an equality (= and `/=`) or relational expression (>, <, >= and <=) must be the name of an attribute of the object being found. A more complex example might be:

```
  find Aircraft(speed >= 3 * (10 ** 8) and pilot_name /= "Fred Bloggs");
```

This would find all instances of the `Aircraft` object whose speed is greater than or equal to the speed of light and is not been flown by Fred Bloggs. In order to use the set of instances returned from the find expression we must assign it to a variable of the appropriate type. Therefore we might write:

```
  aircraft_set : set of instance of Aircraft;
  begin
    aircraft_set := find Aircraft(speed < 100);
  end;
```

If there are no instances that satisfy the condition then the sequence returned from the `find` expression will be empty. One of the most common things to do once we have obtained a set of instances is to loop around the set performing some operation on each instance.

```
  for i in aircraft_set’elements loop
    // …
  end loop;
```

The order in which instances from the set are taken is arbitrary. If we wanted to use a specific order we would have to use an ordering expression. An ordering expression allows the analyst to order a collection of instances using a specific attribute of each instance. For example we could order the aircraft set by their speeds by writing :

```
  aircraft_sequence := aircraft_set ordered_by (speed);
```

The resultant sequence will be such that for each successive element in the sequence, the first will be the element with the lowest speed. We could reverse the ordering by writing:

```
  aircraft_sequence := aircraft_set reverse_ordered_by (speed);
```

It is also possible to specify multiple attributes. For example:

```
  aircraft_sequence := aircraft_set ordered_by (speed, serial_number);.
```

The resultant sequence will be sorted by the speed, and then within each value of this, by the serial number. It is so common to order the set of instances returned from a `find` expression that a `find` expression can be immediately followed by an ordering expression. Hence, we could have written:

```
  aircraft_sequence := find Aircraft(speed < 100) ordered_by (serial_numb
```

It is common to want to find all the instances of an object. We might try and write:

```
  find Aircraft(true);    // illegal
```

However, this syntax is illegal. To find all instances of an object the second form of find must be used:

```
  find_all Aircraft();
```

Notice that no condition is supplied, since we are finding all the instances.

The third form of find, finds an arbitrary instance of an object satisfying a condition. This form starts with the reserved word `find_one` followed by the name of an object and a condition enclosed by brackets. So we might write:

```
  find_one Aircraft(speed < 100);
```

This finds an arbitrary instance of the `Aircraft` object whose speed less than `100`. Again, the condition can be of arbitrary complexity with the same restrictions as before. In order to use the instance returned from the `find` expression we must assign it to a variable of the appropriate type. Thus we might write:

```
  aircraft_instance : instance of Aircraft;
  begin
    aircraft_instance := find_one Aircraft(speed < 100);
  end;
```

If there are no instances that satisfy the condition then the `null` instance is returned. We could check for this by writing:

```
  if aircraft_instance = null then
```

```
    // …
  end if;
```

It is common to want to find an instance of an object and in addition make sure that it is the only such instance. The fourth, and final, form of find is provided to do this:

```
  find_only Aircraft(speed < 100);
```

If there is only one `Aircraft` instance that satisfies the condition then all is well and the find expression returns that instance. If there are either more than one or no `Aircraft` instances that satisfy the condition then an exception will be raised.

For each form of find expression, the name of the object being found can be replaced with an expression that denotes a collection of instances. This can be used to filter a collection of instances to a either a set of instances or a specific instance that satisfies a condition. For example given the appropriate declarations we might write:

```
  aircraft_set := find Aircraft(speed < 100);
  stopped_aircraft_set := find aircraft_set(speed = 0);
  freds_aircraft := find_one aircraft_set(pilot_name = "Fred Bloggs");
```

In summary:

- `find` locate all the instances satisfying a condition.
- `find_all` finds all the instances.
- `find_one` finds an arbitrary instance satisfying a condition.
- `find_only` finds the only instance satisfying a condition.
- `ordered_by` orders a collection of instances by attribute values.
- `reverse_ordered_by` reverse orders a collection of instances by attribute values.

5.3. Deleting Instances

Once we have finished with an instance or a set of instances we can delete them using a `delete` statement. A `delete` statement starts with the reserved word `delete` followed by an expression that denotes either a specific single instance or a collection of instances. Given the examples already seen we might write:

```
delete aircraft_instance;
```

to delete a single aircraft instance, or:

```
delete aircraft_set;
```

to delete all the aircraft instances in a set.

If an instance that participates in a relationship is deleted, before the relationship is unlinked, then an exception will be raised.

Deleting either the null instance or an empty collection does not have any effect.

5.4. Creating Relationships

Relationships between instances are created using a `link` statement. The simplest form of `link` statement starts with the reserved word link followed by an expression that denotes an specific single instance, a specification of a relationship and then another expression that denotes another specific single instance. Given the relationship between the `Pilot` and `Aircraft` objects in Figure 5.2, “Relationship between Pilot and Aircraft” we might write:

```
  link aircraft_instance R3 pilot_instance;
```

This links the instance denoted by `aircraft_instance` to the instance denoted by `pilot_instance` via the relationship R3.

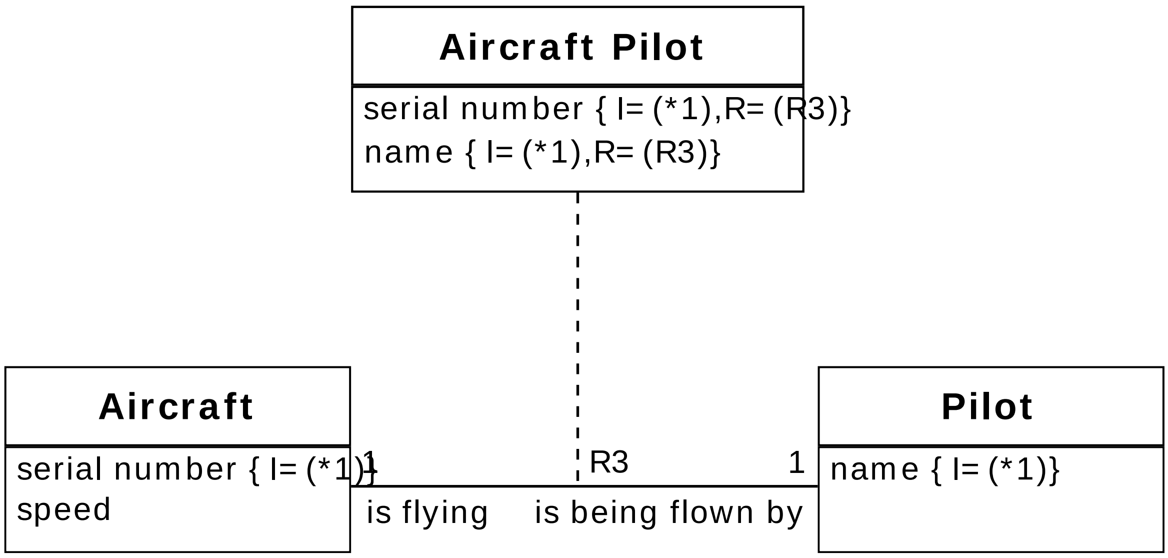
The specification of the relationship between the two instances must be a valid navigation for the `link` statement to be valid. This is discussed in Section 5.5, “Navigating Relationships”. If by linking the two instances together we “break” the model, then an exception is raised. This would happen, for example, if an instance was related to more than one instance in a one-to-one relationship.

When a relationship is created between two instances, any referential attributes that are determined by the relationship are set to the correct values. In our example, if the name of the pilot was “Fred Bloggs” then the pilot name attribute of the aircraft instance would have this value set. Relationships are bi-directional, hence, we would have the same result if we had written:

```
  link pilot_instance R3 aircraft_instance;
```

To create associative relationships the using form of the `link` statement is used. This looks exactly like the simple form with the addition of the reserved word using followed by an expression that denotes a specific single instance of the associative object.

Figure 5.3. Associative Relationship



Given the relationship shown in Figure 5.3, “Associative Relationship” we might write:

```
  link aircraft_instance R11 pilot_instance using aircraft_pilot_instance
```

This links the instance denoted by `aircraft_instance` to the instance denoted by `pilot_instance` via the relationship R3 using the instance denoted by `aircraft_pilot_instance`. Again, the specification of the relationship between the two instances must be a valid navigation for the link statement to be valid. In addition, by linking the two instances using the third we must not “break” the model, any referential attributes that are determined by the relationship are set to the correct values. In this example, all the attributes of the associative instance will have their values set to the appropriate values.

5.5. Navigating Relationships

Relationships that have been instantiated between instances can be navigated using a relationship navigation expression. A relationship navigation starts with an expression that denotes either a single instance or a collection of instances followed by -> and a specification of the relationship. For example, given the relationship shown in Figure 5.3, “Associative Relationship” and an aircraft instance we could navigate to the corresponding pilot by writing any of:

```
  pilot_instance := aircraft_instance -> R3;
  pilot_instance := aircraft_instance -> R3.is_being_flown_by;
  pilot_instance := aircraft_instance -> R3.Pilot;
  pilot_instance := aircraft_instance -> R3.is_being_flown_by.Pilot;
```

Similarly, if we had a collection of aircraft instances we could navigate to the corresponding pilots by writing:

```
  pilot_bag := aircraft_set -> R3;
```

Whether a single instance or a collection of instances is returned from a navigation depends upon the multiplicity of the relationship in the direction being navigated. This is summarised in Table 5.1, “Multiplicity of Relationship Navigation”.

Table 5.1. Multiplicity of Relationship Navigation

Navigation Type	**Multiplicity of Relationship**	
	Single	**Many**
single instance	single instance	set of instances
collection of instance	bag of instances	bag of instances

Given the relationship shown in Figure 5.3, “Associative Relationship” and an aircraft instance we might try to navigate to the corresponding associative instance by writing:

```
  aircraft_pilot_instance := aircraft_instance -> R11;    // illegal
```

This is invalid, since it is not obvious from the relationship specification that we are navigating to the associative instance. Wherever a relationship specification is given, the analyst must ensure that the following are unambiguously specified:

- the relationship
- the direction of navigation

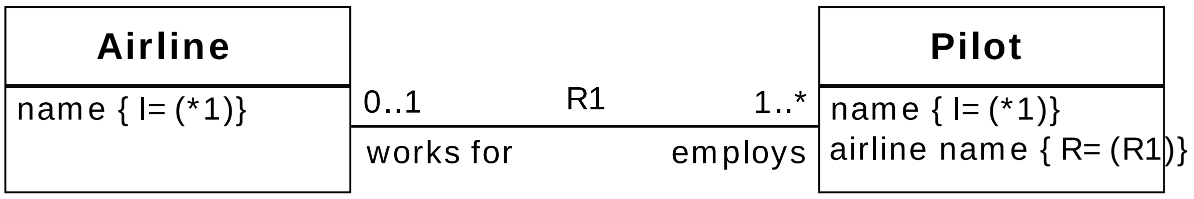
- the destination object

This is achieved by using one of the four forms of relationship specification already seen. For example, in this case we could write either of:

```
  aircraft_pilot_instance := aircraft_instance -> R11.Aircraft_Pilot;
  aircraft_pilot_instance := aircraft_instance -> R11.is_being_flown_by.A
```

Both of these unambiguously specify the relationship, the direction and the destination object. Full examples of all possible relationship specifications are given in [MASL06]

Figure 5.4. Relationship between Airline and Pilot



It is often the case that we wish to navigate two relationships in one go. For example, given the relationship in Figure 5.2, “Relationship between Pilot and Aircraft” and Figure 5.4, “Relationship between Airline and Pilot” we could find out which aircrafts are been flown by pilots employed by an airline by writing:

```
  pilot_bag := airline_instance -> R1;
  aircraft_bag := pilot_bag -> R3;
```

or we could combine the two navigations into one compound navigation.

```
  aircraft_bag := airline_instance -> R1 -> R3;
```

Finally, it is common to have two instances that we know are related via an associative relationship and to want to find the associative instance, or instances, that relate them. For example, given the relationship in Figure 5.3, “Associative Relationship” we could find the associative instance that an `Aircraft` and `Pilot` instance are related to by writing:

```
  aircraft_pilot_bag_1 := airline_instance -> R3.Aircraft_Pilot;
  aircraft_pilot_bag_2 := route_instance -> R3.Aircraft_Pilot;
  aircraft_pilot_set := set of instance of Aircraft_Pilot(aircraft_pilot_
  set of instance of Aircraft_Pilot(aircraft_pilot_bag_2);
  aircraft_pilot_instance := find_only aircraft_pilot_set();
```

This is all a bit longwinded so a special form is provided:

```
  aircraft_pilot_instance := aircraft_instance with pilot_instance -> R3.
```

This is known as correlated navigation.

5.6. Deleting Relationships

Once we have finished with a relationship we can delete it using an `unlink` statement. The `unlink` statement is the exact opposite of a link statement. The simplest form of `unlink` statement starts with the reserved word `unlink` followed by an expression that denotes a single instance, a specification of a relationship and then another expression that denotes another specific single instance.

So to delete the relationship between an `Aircraft` and a `Pilot` via the relationship shown in Figure 5.2, “Relationship between Pilot and Aircraft” we might write:

```
  unlink aircraft_instance R3 pilot_instance;
```

This unlinks the instance denoted by the `aircraft_instance` from the instance denoted by `pilot_instance` via the relationship R3. As before the specification of the relationship between the two instances must be valid. If the two instances are not linked via the relationship then an exception is raised. Any attempt to access the values of any referential attributes that where determined by the relationship will result in an exception. As before, relationships are bidirectional, so we could have deleted the relationship by writing:

```
  unlink pilot_instance R3 aircraft_instance;
```

To delete associative relationships the using form of the `unlink` statement must be used. So given the relationship shown in Figure 5.3, “Associative Relationship” we might write:

```
  unlink aircraft_instance R11 pilot_instance using aircraft_pilot_instan
```

This unlinks the instance denoted by `aircraft_instance` from the instance denoted by `pilot_instance` via the relationship R3 using the instance denoted by `aircraft_pilot_instance`. Again, the specification of the relationship between the two instances must be a valid navigation for the unlink statement to be valid. Also the two instances must have been linked together using the third otherwise an exception is raised. Like before, any attempt to access the values of any referential attributes that where determined by the relationship will result in an exception.

6. Actions

Actions are the places where statements, together with appropriate declarations, that encapsulate a set of processes are held. These can then be invoked from within the processing of any thread of control. Essentially, there are three types of actions; object services, instance services and states. There are also two special types of actions known as externals and scenarios.

6.1. Object Services

Object services are actions that are associated with an object, but not a specific instance of that object. The exact form of a object service definition is unimportant as it is automatically generated from the case tool. The code that is placed in the definition looks exactly like a block as described in Section 2.2 except that the reserved word `declare` is not needed. For example, the `Aircraft` object seen previously might provide an object service called `get_fast_pilots` that returns the set of pilots who are flying aircraft, which are going very fast. The object service definition might look something like:

```
public service Air_Management::Aircraft.get_fast_pilots( pilot_set : ou
  aircraft_set : set of instance of Aircraft;
  pilot_bag : bag of instance of Pilot;
begin
  aircraft_set := find Aircraft(speed >= 3 * (10 ** 8));
  pilot_bag := aircraft_set -> R1;
  pilot_set := set of instance of Pilot(pilot_bag);
end service;
```

Notice the use of the reserved word `public` that states that the service may be accessed by any code that can access the object. An object service can also be declared as `private`, in which case access is permitted only from actions defined by the object. Also the parameters of the service are tagged with the reserved words `in` or `out` which conveys the direction of information transfer. We could invoke this service by writing:

```
pilot_set : set of instance of Pilot;
...
Aircraft.get_fast_pilots(pilot_set);
```

If we wanted to call an object service from an action defined by the object we could drop the object name and the dot from the invocation. So if the invocation above was in an action defined by the `Aircraft` object we could have written:

```
aircraft_instance.get_fast_pilots(pilot_set);
```

Note that the service body does not have access to this specific instance. Hence, invoking the object service on any other Aircraft instance would have the same result.

6.2. Instance Services

Instance services are actions that are associated with a specific instance of an object. Like object services, the exact form of an instance service definition is unimportant as it is automatically generated from the case tool. The code that is placed in the definition looks exactly like a block as described before. For example, the `Aircraft` object seen previously might provide an instance service called `increase_speed` that increases the speed of a specific `Aircraft` instance. The instance service definition might look something like:

```
  public instance service Air_Management::Aircraft.increase_speed(increas
  begin
    speed := speed + increase;
  end service;
```

Again notice the use of the reserved word public. It has the same meaning as object services. Also the use of the reserved words in or out are as before. We could invoke this service by writing:

```
  aircraft_instance : instance of Aircraft;
  …
  aircraft_instance.increase_speed(61);
```

On calling `increase_speed`, the expression `61` is evaluated and assigned to the parameter increase, which behaves as a constant. The value of the attribute `speed` is then added to the parameter `increase` and assigned to the attribute `speed`.

Inside the body of the instance service, all of the attributes of the instance are available as if declared as variables. If an attribute is not a preferred or referential attribute then its value can be set by using an assignment statement. Values of preferred and referential attributes can only be read. This matches the rules described in Section 5.1, “Creating Instances”. In addition, the reserved word this denotes the instance that the service was invoked upon. Therefore, we could have written the assignment in our example as:

```
  this.speed := this.speed + increase;
```

If we wanted to call an instance service from an action being performed on a specific instance, we could drop the instance and the dot from the invocation. So if the invocation above was in an action being performed on a specific `Aircraft` instance we could have written:

```
  increase_speed(61);
```

This invokes the service on the instance that the action is been performed on.

6.3. Polymorphic Services

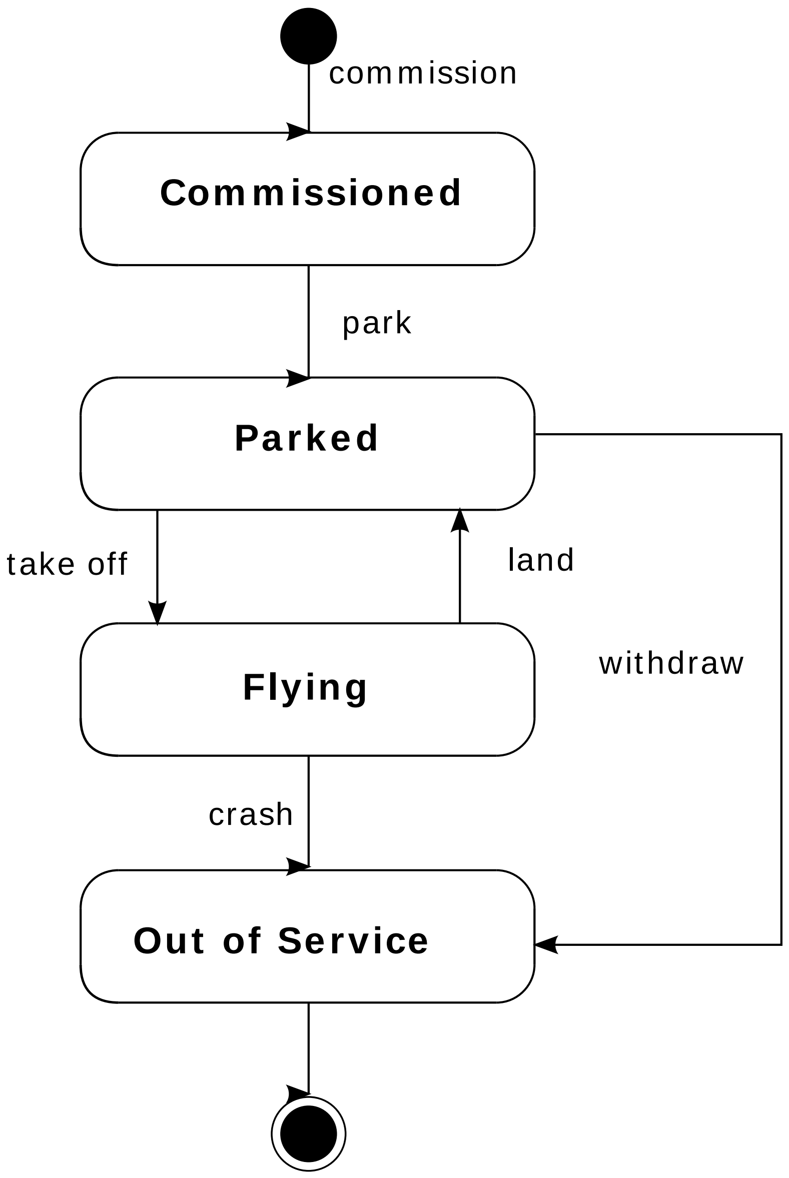
A polymorphic service is an instance service whose implementation is deferred to a subtype of an object. When a polymorphic service is invoked, which action is actually executed depends on the current subtype of the instance that the service is invoked on. Within the body of a polymorphic service, the same rules as normal instance services apply. In particular, the attributes of the current subtype instance are available as if declared as variables. Also, the reserved word `this` denotes the current subtype instance of the instance that the service was invoked upon.

Note that a polymorphic service can also be deferred, by a subtype, to one of its subtype hierarchies.

6.4. States and Events

States are actions that are associated with the life cycle of either a specific instance or an assigner. Again, like all actions, the exact form of a state definition is unimportant as it is automatically generated from the case tool. The code that is placed in the definition looks exactly like a block as described above.

Figure 6.1. Life cycle of an Aircraft



Given the lifecycle shown in Figure 6.1, “Life cycle of an Aircraft” the state definition of the parked state might look something like:

```
  state Air_Management::Aircraft.parked(airport_name : in string, gate_nu
    gate : instance of Airport_Gate;
  begin
    speed := 0;
    gate := find_only Airport_Gate(airport_name = airport_name and
    number = gate_number);
    link this R15 gate;
  end state;
```

Like instance services, inside the body of the state, all of the attributes of the instance are available as if declared as variables. The same rules about assignment of values to preferred and referential attributes apply. Again, the reserved word `this` denotes the instance that has moved into the state. Unlike services, states cannot be invoked directly. Instead we generate events to specific instances to move the instance to a new state. So we might write:

```
  aircraft_instance : instance of Aircraft;
  …
  generate Aircraft.land("Heathrow", 5) to aircraft_instance;
```

If the specific instance was currently flying then this event would cause it to move into the parked state and the action associated with this state would be invoked. Note that this invocation is asynchronous. The number and types of the parameters of an event that causes an instance to move into a particular state must match the number and types of the parameters of that state.

The values of the arguments in a `generate` statement are evaluated, at the time of the generate, and assigned to the matching parameters of the state when it is actioned. Because, the `generate` and the corresponding actioning of the state is asynchronous only input parameters on events and hence states are allowed. If we wanted to generate an event from an action defined by the object we could drop the object name and the dot from the generation. So if the generate statement above was in an action defined by the aircraft object we could have written:

```
  generate land("Heathrow", 5) to aircraft_instance;
```

Notice that we still need to provide the specific instance that we are generating the event to. We could use the reserved word `this` if it is defined by the action.

There are two types of events can be generated without providing a specific instance; creation events and assigner events. Before we discuss these, a quick discussion about polymorphic events is required.

6.4.1. Polymorphic Events

Polymorphic events are special type of event that, as well as causing a specific instance to move to a new state, also cause the current subtype of the instance to move to a new state.

When a polymorphic event is consumed, the specific instance that the event was generated to, moves to a new state and the action associated with this state is invoked. The event is then propagated to the current subtype of the instance. This causes the subtype instance to move to a new state and the action associated with this state to be invoked. This is repeated all the way down the subtype hierarchy.

Note that, for some instances in the subtype hierarchy, including the first one, the polymorphic event could be ignored. In this case the instance does not move to a new state and no action is invoked. The event is propagated as before.

6.4.2. Creation Events

Creation events are used to create instances of an object asynchronously. For example, we would generate the commission event in Figure 6.1, “Life cycle of an Aircraft” by writing:

```
generate Aircraft.commission();
```

Again, if we wanted to generate the creation event from an action defined by the object we could drop the object name and the dot from the `generate` statement.

```
generate commission();
```

Unlike normal states, creation states do not have access to a specific instance. It is up to the body of the creation state to create an instance. For example, the definition of the commissioned state might look something like the following:

```
creation state Air_Management::Aircraft.commissioned() is
  new_aircraft : instance of Aircraft;
begin
  new_aircraft := create unique Aircraft(Current_State => commissioned)
end state;
```

Notice that the create expression specifies the state that the created instance starts in. This will usually be the creation state.

6.4.3. Assigner Events

Each associative object in a domain can have an assigner life cycle. Assigner life cycles provide a single point of control through which competing requests are serialised. Because of this, there is only one copy of an assigner life cycle for all instances of the associative object.

Figure 6.2. Assigner Object

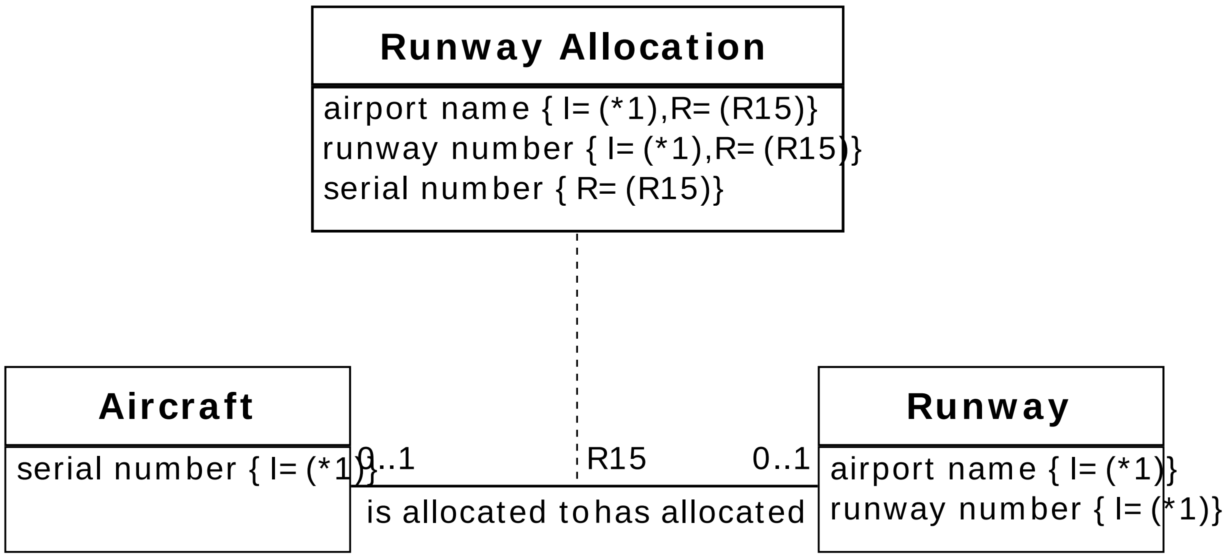
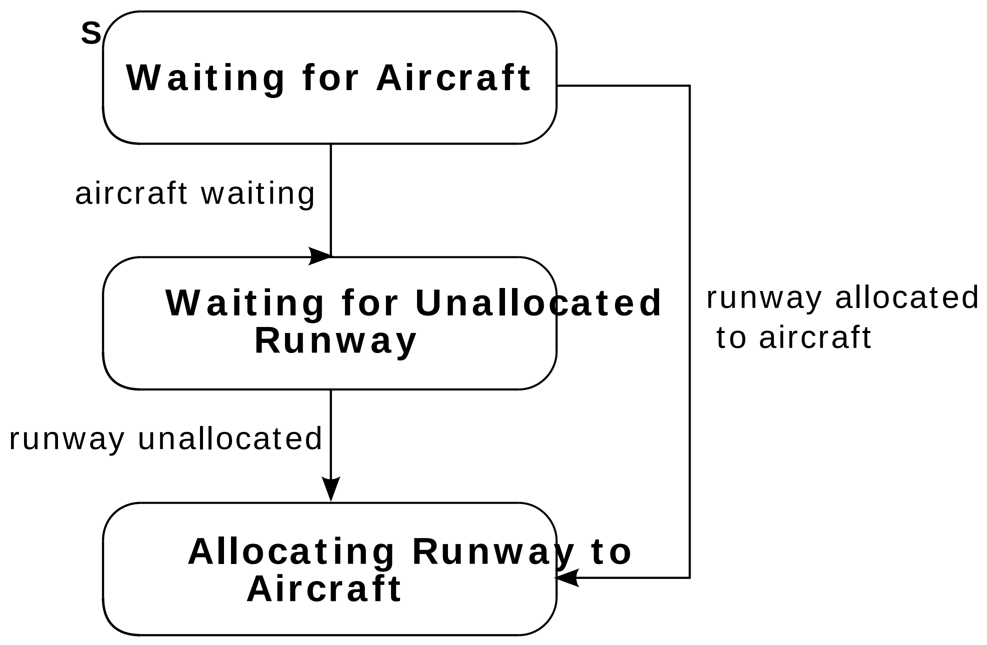


Figure 6.3. Life cycle of a Runway Allocation Assigner


S
Waiting for Aircraft
aircraft waiting
Waiting for Unallocated Runway
runway allocated to aircraft
runway unallocated
Allocating Runway to Aircraft

Given the relationship shown in Figure 6.2, “Assigner Object”and the corresponding assigner life cycle shown in Figure 6.3, “Life cycle of a Runway Allocation Assigner”, we might write:

```
  generate Runway_Allocation.aircraft_waiting();
```

Again, if we wanted to generate the assigner event from an action defined by the corresponding associative object we could drop the object name and the dot from the `generate` statement.

```
  generate aircraft_waiting();
```

Like creation states, assigner states do not have access to a specific instance. Notice that one state in an assigner life cycle is tagged as the start, or initial, state. This is the state in which the assigner starts in. Other than this fact there is nothing special about this state compared with other assigner states.d

6.4.4. Terminal States

Finally, to finish this section a quick discussion about terminal states. Going back to our example of the `Aircraft` life cycle, we can see that the *out of service* state is a terminal state. A terminal state does not necessarily imply that an instance in this state must cease to exist, it could simply hang around in this state for historic purposes.

However, what is special about terminal states is that they are the only type of instance states, that is a non-creation states in a non-assigner life cycle, where we are allowed to delete the current instance. That is, in a terminal state we are allowed to write:

```
  delete this;
```

It is invalid to try and do this is any non-terminal state.

7. Device Input/Output

MASL provides a mechanism for streaming data into and out of a series of built in or user defined devices. To achieve this a device type has been defined by the language. A built-in device called `console` is provided, which enables data to be streamed to standard output and from standard input respectively. The following sections describe each of these in turn.

7.1. Console Device

The `console` device is a built in device type used for streaming data to the output device or for reading data from the input device. It does not need to be defined within any statement blocks, it can just be used as in the examples in Section 7.1.1, “Console Output” and Section 7.1.2, “Console Input” below.

7.1.1. Console Output

Console output is available for all types (i.e. boolean, stringetc.) except instance handles where the address of the handle is streamed and container types where the number of elements held is streamed; this is basically garbage output for these types. The current output for container types and instance handles might change in the future and so applications should not become dependent upon their current format. The example below shows how the console device can be used for several types. The output for the boolean type will be streamed in its textual form.

```
    int_out  : Integer := 10000;
    real_out : real    := 1.2345;
    bool_out : Boolean := false;
  begin
    console <<  “TEST STARTED ”;
    console << "- TEST PASSED" << endl;
    console << int_out << “  “ << real_out  << “  “<< bool_out << endl;
  end;
```

The output produced from the above code is shown below :

```
  TEST STARTED - TEST PASSED
  10000 1.2345 false
```

Several manipulators are currently supported, `endl` and `flush`. The `endl` manipulator will stream out a new line character, while the `flush` manipulator will flush the current contents of the device’s stream buffer (should any buffering be provided by the device).

No exceptions are raised when streaming data out to the console device.

7.1.2. Console Input

The console device can also be used to read data from standard input. This only applies for non-composite defined types. It can therefore not be used for container or user defined composite types. An example of its use is shown below.

```
    name_string     : string;
    person_age      : integer;
    employed_status : boolean;
    childs_name     : string;
    childs_age      : integer;
    childs_height   : real;
    childs_weight   : real;

  begin
    begin
      console << “Enter name  ”;
      console >> name_string;          // read in name

      console << “Enter Age “;
      console  >>  person_age;        // read in age

      console << “Enter Employeed (true/false)  ”;
      console  >> employed_status;

      console << “Enter child details  “;
      console >> childs_name >> childs_age >> childs_height >> childs_wei
    end;

  exception
    when Standard::program_error =>
      // Stream error encountered…
      // i.e entered a float for the age when integer expected.
  end;
```

Should an error occur while reading data in from the `console` device, a MASL program error will be thrown. This can be caught by the application and processed accordingly.

8. MASL Examples

8.1. Characteristics

```
  t  : string;
  i  : integer   := 1;
  r  : real      := -99.9;
  e  : Time_Unit := DAY;
  b  : boolean   := false;
  d  : record_structure_type;
  ds : sequence of record_structure_type;
  p  : instance of Pilot;
  ps : sequence of instance of Pilot;

begin
   //image
  t := i'image;
  if (t/="1") then
    raise Standard::constraint_error;
  end if;
  t := r'image;
  if (t/="-99.900002") then
    raise Standard::constraint_error;
  end if;
  t := b'image;
  if (t/="FALSE") then
    raise Standard::constraint_error;
  end if;
  t := e'image;
  if (t/="DAY") then
    raise Standard::constraint_error;
  end if;

  //upper
  t := "loWer";
  t := t'upper;
  if (t/="LOWER") then
    raise Standard::constraint_error;
  end if;
```

```
//lower
t := "UppER";
t := t'lower;
if (t/="upper") then
  raise Standard::constraint_error;
end if;

//firstcharpos
t := "upper";
i := t'firstcharpos('u');
if (i/=t'first) then
  raise Standard::constraint_error;
end if;
i := t'firstcharpos('r');
if (i/=t'last) then
  raise Standard::constraint_error;
end if;
t := t[t'firstcharpos('p')..t'firstcharpos('e')];
if (t/="ppe") then
  raise Standard::constraint_error;
end if;
begin
  t := t[t'firstcharpos('p')..t'firstcharpos('x')];
  t := "error";
exception
  when Standard::constraint_error =>
    null;
end;
if (t = "error") then
  raise Standard::constraint_error;
end if;

//type first,last
if (Time_Unit'first/=MILLISECOND) then
  raise Standard::constraint_error;
end if;
if (Time_Unit'last/=DAY) then
  raise Standard::constraint_error;
end if;
if (octal_type'first/=0) then
```

```
    raise Standard::constraint_error;
  end if;
  if (octal_type'last/=7) then
    raise Standard::constraint_error;
  end if;


  //string first,last,size
  t := "";
  if (t'first/=0 or t'last>=0 or t'length/=0)  then
    raise Standard::constraint_error;
  end if;
  t := t & " ";
  if (t'first/=0 or t'last/=0 or t'length/=1)  then
    raise Standard::constraint_error;
  end if;
  t := t & " ";
  if (t'first/=0 or t'last/=1 or t'length/=2)  then
    raise Standard::constraint_error;
  end if;


  //collection first,last,size
  if (ds'first/=0 or ds'last>=0 or ds'length/=0)  then
    raise Standard::constraint_error;
  end if;
  ds := ds & d;
  if (ds'first/=0 or ds'last/=0 or ds'length/=1)  then
    raise Standard::constraint_error;
  end if;
  ds := ds & d;
  if (ds'first/=0 or ds'last/=1 or ds'length/=2)  then
    raise Standard::constraint_error;
  end if;
  p := create Pilot (age=>33);
  if (ps'first/=0 or ps'last>=0 or ps'length/=0)  then
    raise Standard::constraint_error;
  end if;
  ps := ps & p;
  if (ps'first/=0 or ps'last/=0 or ps'length/=1)  then
    raise Standard::constraint_error;
  end if;
  ps := ps & p;
  if (ps'first/=0 or ps'last/=1 or ps'length/=2)  then
    raise Standard::constraint_error;
  end if;

  //pred,succ
  e := MILLISECOND;
  if (e'succ/=SECOND) then
    raise Standard::constraint_error;
  end if;
  e := e'succ;
  if (e'pred/=MILLISECOND) then
    raise Standard::constraint_error;
  end if;

  i := 0;
  begin
    e := MILLISECOND;
    e := e'pred;
  exception
    when Standard::constraint_error =>
      i := i + 1;
  end;
  begin
    e := DAY;
    e := e'succ;
  exception
    when Standard::constraint_error =>
      i := i + 1;
  end;
  if (i /= 2) then
    raise Standard::constraint_error;
  end if;

  //pos
  //i := Time_Unit'pos(MILLISECOND);
  //if (i/=1) then
  //  raise Standard::constraint_error;
  //end if;
  //value
  //e := Time_Unit'value("MILLISECOND");
  //if (e/=MILLISECOND) then
  //  raise Standard::constraint_error;
  //end if;
  //t := "DAY";
  //e := Time_Unit'value(t);
  //if (e/=DAY) then
  //  raise Standard::constraint_error;
  //end if;

  //find length
  //i := find_all Pilot ()'length;
  //if (i=0)  then
  //  raise Standard::constraint_error;
  //end if;

end;
```

8.2. Creation and Navigation

```
  declare
  pilot          : instance of Pilot;
  airplane       : instance of Airplane;
  aps            : instance of Airplane_Pilot_Assignment;
  wing           : instance of Wing;
  wings          : sequence of instance of Wing;
  wheel          : instance of Wheel;
  wheels         : sequence of instance of Wheel;
  i              : integer;
begin
  i:= 0;
  begin
    wheels := pilot->R2->R3->R7;
    i := 1;
  exception
    when Standard::constraint_error =>
      i:= 2;
  end;
  if (i/=2 or wheels'length /= 0) then
    raise Standard::constraint_error;
  end if;

  airplane := create Airplane (model=>747);

  wing  := create Wing(span=>300);
  wheel := create Wheel(size=>36);
  link wheel R7 wing;
  link wing R3 airplane;

  wing  := create Wing(span=>300);
  wheel := create Wheel(size=>36);
  link wheel R7 wing;
  link wing R3 airplane;

  pilot := create Pilot(age=>33);
  aps := create Airplane_Pilot_Assignment(Current_State=>state_one);
  link airplane R2 pilot using aps;

  wheels := pilot->R2->R3->R7;
  if (wheels'length /= 2 or wheels[wheels'first].size /= 36) then
    raise Standard::constraint_error;
  end if;
end service;
```

8.3. Data Sets

```
declare
  i    : integer := 0;
  r    : real    := 0.0;
  t    : string  :="";
  d1   : record_structure_type;
  s1   : sequence of record_structure_type;
  fs   : sequence of Server::fixed_size_structure_type;
  date : Calendar::Date;
  time : Calendar::Time_of_Day;
```

```
begin
  //create a sequence of structures
  for x in 1..3 loop
    d1.int_field  := x;
    d1.real_field := real(x*1.1);
    d1.text_field := x'image;
    s1 := s1 & d1;
  end loop;
  for x in s1'elements loop
    i := i + x.int_field;
    t := t & x.text_field;
  end loop;
  if (i /= 6 or t /="123") then
    raise Standard::constraint_error;
  end if;

  //normal order
  s1 := s1 ordered_by(int_field);
  t := "";
  for x in s1'elements loop
    t:=t & x.text_field;
  end loop;
  if (t /="123") then
    raise Standard::constraint_error;
  end if;

  //reverse order
  s1 := s1 reverse_ordered_by(int_field);
  t := "";
  for x in s1'elements loop
    t:=t & x.text_field;
  end loop;
  if (t /="321") then
    raise Standard::constraint_error;
  end if;

  //slice
  d1 := s1[s1'last-1];
  t := "";
  for x in s1'elements loop
    t:=t & x.text_field;
  end loop;
  if (t /="321") then
    raise Standard::constraint_error;
  end if;
  if (d1.text_field /= "2") then
    raise Standard::constraint_error;
  end if;

  s1[s1'last-1].text_field := "two";
  t := "";
  for x in s1'elements loop
    t:=t & x.text_field;
  end loop;
  if (t /="3two1") then
    raise Standard::constraint_error;
  end if;

  //normal order
  s1 := s1 ordered_by(int_field);
  t := "";
  for x in s1'elements loop
    t:=t & x.text_field;
  end loop;
  if (t /="1two3") then
    raise Standard::constraint_error;
  end if;

  //reverse order
  s1 := s1 reverse_ordered_by(text_field);
  t := "";
  for x in s1'elements loop
    t:=t & x.text_field;
  end loop;
  if (t /="two31") then
    raise Standard::constraint_error;
  end if;

  //multiple order
  s1 :=      (3,1.2,"bla",date,time,blue);
  s1 := s1 & (1,1.0,"bla",date,time,blue);
  s1 := s1 & (4,1.0,"bla",date,time,blue);
  s1 := s1 & (2,1.0,"bbla",date,time,blue);
  s1 := s1 & (2,1.0,"bla",date,time,blue);
  s1 := s1 & (3,1.0,"bla",date,time,blue);
  s1 := s1 ordered_by(int_field,real_field,text_field);
  //for x in s1'elements loop
  //  Text_IO::put_line(x'image);
  //end loop;
  if (s1[s1'first+1].text_field /= "bbla") then
    raise Standard::constraint_error;
  end if;

  //structure comparisons, removed from language
  //d1 := s1[s1'first];
  //if (d1 /= s1[s1'first]) then
  //  raise Standard::constraint_error;
  //end if;
  //d1.int_field := s1[s1'first].int_field & 2;
  //if (d1 = s1[s1'first]) then
  //  raise Standard::constraint_error;
  //end if;

  //double nested normal order, testing cgen here
  for x in s1'elements loop
    s1 := s1 ordered_by(text_field);
  end loop;
  s1 := s1 ordered_by(text_field);

  //other domain structure ordering, testing cgen here
  fs := fs ordered_by(i);
end service;
```

8.4. String Manipulation

```
declare
  text   : string  := "";
  text2  : string  := "";
  passed : boolean := false;
begin
```

```
  text := "hello";
  for c in text'elements loop
    text2 := text2 & c;
  end loop;
  if (text2 /= "hello") then
    raise Standard::constraint_error;
  end if;

  text := "hello";
  for c in reverse text'elements loop
    text2 := text2 & c;
  end loop;
  if (text2 /= "helloolleh") then
    raise Standard::constraint_error;
  end if;

  text  := "hello";
  text2 := "";
  for i in text'range loop
    text2 := text2 & text[i];
  end loop;
  if (text2 /= "hello") then
    raise Standard::constraint_error;
  end if;

  text := "hello";
  for i in reverse text'range loop
    text2 := text2 & text[i];
  end loop;
  if (text2 /= "helloolleh") then
    raise Standard::constraint_error;
  end if;

  text := "hello";
  if (text /= "hello") then
    raise Standard::constraint_error;
  end if;

  text := "hello";
  text := text[text'first];
  if (text /= "h") then
```

```
    raise Standard::constraint_error;
  end if;

  text := "hello";
  text := text[text'last];
  if (text /= "o") then
    raise Standard::constraint_error;
  end if;

  text[text'first] := 'h';
  if (text /= "h") then
    raise Standard::constraint_error;
  end if;

  text := text & "ello";
  if (text /= "hello") then
    raise Standard::constraint_error;
  end if;

  text := "hello";
  if (text < "goodbye") then
    raise Standard::constraint_error;
  end if;
  if ("goodbye" > text) then
    raise Standard::constraint_error;
  end if;

  text := "hello";
  text := text[0..2];
  if (text /= "hel") then
    raise Standard::constraint_error;
  end if;

  //should all raise exceptions
  begin
    passed := false;
    text := text[text'first-1];
  exception
    when Standard::constraint_error =>
      passed := true;
  end;
  if (passed = false) then
    raise Standard::constraint_error;
  end if;

  begin
    passed := false;
    text := text[text'length];
  exception
    when Standard::constraint_error =>
      passed := true;
  end;
  if (passed = false) then
    raise Standard::constraint_error;
  end if;

 begin
    passed := false;
    text[text'length]  := 'h';
  exception
    when Standard::constraint_error =>
      passed := true;
  end;
  if (passed = false) then
    raise Standard::constraint_error;
  end if;

 begin
    passed := false;
    text[text'first-1] := 'h';
  exception
    when Standard::constraint_error =>
      passed := true;
  end;
  if (passed = false) then
    raise Standard::constraint_error;
  end if;

 begin
    passed := false;
    text := text[100..2];
  exception
```

```
    when Standard::constraint_error =>
      passed := true;
  end;
  if (passed = false) then
    raise Standard::constraint_error;
  end if;
end service;
```

8.5. String Literals

```
declare
  i    : integer   := 10;
  r    : real      := 88.8;
  c    : character := 'g';
  s    : string    := "h";
  b    : boolean   := false;
  e    : Time_Unit := DAY;
  f    : Calendar::time_unit_type;
  ih   : instance of Pilot;
  ihc  : sequence of instance of Pilot;
  re   : record_structure_type;
  rec  : sequence of record_structure_type;
  date : Calendar::Date;
  time : Calendar::Time_of_Day;
begin
  //literals
  i :=  0;
  i :=  99;
  i := +99;
  i := -99;
  r :=  99.9;
  r := +99;
  r := -99;
  r := 0.9;
  c := 'f';
  s := "";
  s := "hello";
  b := true;
  b := false;
  e := DAY;
  f := Calendar::SECOND;
  ih := null;
  re := (4,5.0,"hello",date,time,green);

  //names
  i := 0;
  s[4] := 'r';
  ih := create Pilot(age=>44);
  ih.NI := 99;
  ihc := ihc & ih;
  ihc[ihc'first].NI := 10;
  re.int_field := 0;
  rec := rec & re;
  rec := rec & (4,5.0,"hello",date,time,blue);
  rec[rec'first].int_field := 1;

  //expressions, all on integers
  i := 100;
  if not (i=100 and i/=1) then
    raise Standard::constraint_error;
  elsif not (i<1000 or i>1) then
    raise Standard::constraint_error;
  elsif (i<=100 xor i>=100) then
    raise Standard::constraint_error;
  elsif not (i=99+1 or i=101-1) then
    raise Standard::constraint_error;
  elsif not (i=10*10) then
    raise Standard::constraint_error;
  elsif not (i=1000/10) then
    raise Standard::constraint_error;
  elsif not (i=10**2) then
    raise Standard::constraint_error;
  elsif not (i=1100 rem 1000) then
    raise Standard::constraint_error;
  elsif not (i mod 50 = 0) then
    raise Standard::constraint_error;
  elsif not (i = abs(-100)) then
    raise Standard::constraint_error;
  end if;
  //various expressions on other all base types
  r := 99.9;
  c := 'f';
  s := "hello";
  b := true;
  e := DAY;
  f := Calendar::SECOND;
  ih := null;
  if (r<100-1.3) then
    raise Standard::constraint_error;
  elsif (c /= 'f') then
    raise Standard::constraint_error;
  elsif (c < 'a') then
    raise Standard::constraint_error;
  elsif not (c > 'a') then
    raise Standard::constraint_error;
  elsif (s /= "hello") then
    raise Standard::constraint_error;
  elsif not (s = "hello") then
    raise Standard::constraint_error;
  elsif not (s > "gello") then
    raise Standard::constraint_error;
  elsif (b=false) then
    raise Standard::constraint_error;
  elsif (e=SECOND) then
    raise Standard::constraint_error;
  elsif not (e>SECOND) then
    raise Standard::constraint_error;
  elsif (f=Calendar::DAY) then
    raise Standard::constraint_error;
  elsif not (f<Calendar::DAY) then
    raise Standard::constraint_error;
  elsif not (date.month=date.month) then
    raise Standard::constraint_error;
  elsif not (date.month<=date.month) then
    raise Standard::constraint_error;
  elsif (ih/=null) then
    raise Standard::constraint_error;
  end if;
end;
```

8.6. Find Statements

```
declare
  pilot     : instance of Pilot;
  pilots    : sequence of instance of Pilot;
  xpilots   : sequence of instance of Pilot;
  total_age : integer := 0;
  temp      : integer := 0;
  today     : Calendar::date_type;
begin
  //find one on no population
  pilot := find_one Pilot(age=5);
  if (pilot/=null) then
    raise Standard::constraint_error;
  end if;

  //find one on single population
  pilot := create Pilot(age=>10,qualified=>today);
  pilot := find_one Pilot(age=5 and qualified.year=today.year);
  if (pilot/=null) then
    raise Standard::constraint_error;
  end if;
  pilot := find_one Pilot(age=10);
  if (pilot=null or pilot.age /=10) then
    raise Standard::constraint_error;
  end if;

  //find one on small population
  pilot := create Pilot(age=>20);
  pilot := create Pilot(age=>30);
  pilot := find_one Pilot(age=5);
  if (pilot/=null) then
    raise Standard::constraint_error;
  end if;
  pilot := find_one Pilot(age=10);
  if (pilot=null or pilot.age /= 10) then
    raise Standard::constraint_error;
  end if;
  pilot := find_one Pilot(age=20);
  if (pilot=null or pilot.age /= 20) then
    raise Standard::constraint_error;
  end if;
  pilot := find_one Pilot(age=30);
  if (pilot=null or pilot.age /= 30) then
    raise Standard::constraint_error;
  end if;

  //find one on large population
  for age in 1..100 loop
    pilot := create Pilot(age=>age);
  end loop;
  pilot := find_one Pilot(age=1);
  if (pilot=null or pilot.age /= 1) then
    raise Standard::constraint_error;
  end if;
  pilot := find_one Pilot(age=100);
  if (pilot=null or pilot.age /= 100) then
    raise Standard::constraint_error;
  end if;

  //cleanup
  pilot := find_one Pilot();
  while (pilot /= null) loop
    delete pilot;
    pilot := find_one Pilot();
  end loop;
  pilot := find_one Pilot();
  if (pilot/=null) then
    raise Standard::constraint_error;
  end if;

  //find all on no population
  pilots := find_all Pilot();
  if (pilots'length /= 0) then
    raise Standard::constraint_error;
  end if;

  //find all on single population
  pilot := create Pilot(age=>10);
  pilots := find_all Pilot();
  if (pilots'length /= 1 or pilots[pilots'first].age /= 10) then
    raise Standard::constraint_error;
  end if;

  //find all on small population
  pilot := create Pilot(age=>20);
  pilot := create Pilot(age=>30);
  pilots := find_all Pilot();
  if (pilots'length /= 3 or pilots[pilots'first+0].age /= 10
                     or pilots[pilots'first+1].age /= 20
                     or pilots[pilots'first+2].age /= 30) then
    raise Standard::constraint_error;
  end if;

  //find all on large population
  delete pilots;
  total_age := 0;
  for age in 1..100 loop
    pilot := create Pilot(age=>age);
    total_age := total_age+age;
  end loop;
  pilots := find_all Pilot();
  temp := 0;
  for pilot in pilots'elements loop
    temp := temp + pilot.age;
  end loop;
  if (total_age /= temp) then
    raise Standard::constraint_error;
  end if;

  //cleanup
  pilots := find_all Pilot();
  delete pilots;

  //find on no population
  pilots := find Pilot(age=4);
  if (pilots'length /= 0) then
    raise Standard::constraint_error;
  end if;
```

```
//find on single population
pilot := create Pilot(age=>10);
pilots := find Pilot(age=10);
if (pilots'length /= 1 or pilots[pilots'first].age /= 10) then
  raise Standard::constraint_error;
end if;
pilots := find Pilot(age=100);
if (pilots'length /= 0) then
  raise Standard::constraint_error;
end if;

//find on small population
pilot := create Pilot(age=>20);
pilot := create Pilot(age=>30);
pilots := find Pilot(age>10);
if (pilots'length /= 2 or pilots[pilots'first+0].age /= 20
                     or pilots[pilots'first+1].age /= 30) then
  raise Standard::constraint_error;
end if;
pilots := find Pilot(age<=10);
if (pilots'length /= 1 or pilots[pilots'first+0].age /= 10) then
  raise Standard::constraint_error;
end if;
pilots := find Pilot(age<1);
if (pilots'length /= 0) then
  raise Standard::constraint_error;
end if;

//find on large population
delete pilots;
for age in 1..100 loop
  pilot := create Pilot(age=>age);
end loop;
pilots := find Pilot(age>50);
if (pilots'length /= 50) then
  raise Standard::constraint_error;
end if;

//cleanup
pilots := find_all Pilot();
```

```
  delete pilots;

  //*****************************************************************************
  //find only no population
  pilot := find_only Pilot(age=4);
  if (pilot /= null) then
    raise Standard::constraint_error;
  end if;

  //find on single population
  pilot := create Pilot(age=>10);
  pilot := find_only Pilot(age=10);
  if (pilot.age /= 10) then
    raise Standard::constraint_error;
  end if;
  pilot := find_only Pilot(age=100);
  if (pilot /= null) then
    raise Standard::constraint_error;
  end if;

  //find on small population
  pilot := create Pilot(age=>20);
  pilot := create Pilot(age=>30);
  pilot := find_only Pilot(age=20);
  if (pilot.age /= 20) then
    raise Standard::constraint_error;
  end if;
  begin
    pilot := find_only Pilot(age>=10);
  exception
    when Standard::constraint_error =>
      pilot := find_only Pilot(age=30);
  end;
  if (pilot.age /= 30) then
    raise Standard::constraint_error;
  end if;

  //cleanup
  pilots := find_all Pilot();
  delete pilots;
  //find over empty collections
  if (xpilots'length /= 0) then
    raise Standard::constraint_error;
  end if;
  pilots := find_all xpilots();
  if (pilots'length /= 0) then
    raise Standard::constraint_error;
  end if;

  //find on single element collection
  pilot   := create Pilot(age=>10);
  xpilots := find_all Pilot();
  pilots  := find xpilots(age=10);
  if (pilots'length /= 1 or pilots[pilots'first].age /= 10) then
    raise Standard::constraint_error;
  end if;
  pilots := find xpilots(age=100);
  if (pilots'length /= 0) then
    raise Standard::constraint_error;
  end if;

  //find over small collections
  pilot := create Pilot(age=>20);
  pilot := create Pilot(age=>30);
  xpilots := find_all Pilot();
  pilots := find xpilots(age>10);
  if (pilots'length /= 2 or pilots[pilots'first+0].age /= 20
                      or pilots[pilots'first+1].age /= 30) then
    raise Standard::constraint_error;
  end if;
  pilots := find xpilots(age<=10);
  if (pilots'length /= 1 or pilots[pilots'first+0].age /= 10) then
    raise Standard::constraint_error;
  end if;
  pilots := find xpilots(age<1);
  if (pilots'length /= 0) then
    raise Standard::constraint_error;
  end if;

  //find over large collections
  pilots := find_all Pilot();
  delete pilots;
  for age in 1..100 loop
    pilot := create Pilot(age=>age);
  end loop;
  xpilots := find_all Pilot();
  pilots := find xpilots(age>50);
  if (pilots'length /= 50) then
    raise Standard::constraint_error;
  end if;

  //cleanup
  pilots := find_all Pilot();
  delete pilots;

  //various find conditions
  pilot := create Pilot(age=>10);
  pilot := create Pilot(age=>20);

  pilots := find Pilot(age/=10);
  if (pilots'length /= 1 or pilots[pilots'first].age /= 20) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(age=10);
  if (pilots'length /= 1 or pilots[pilots'first].age /= 10) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(age<10);
  if (pilots'length /= 0) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(age>20);
  if (pilots'length /= 0) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(age<=20);
  if (pilots'length /= 2) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(age>=20);
  if (pilots'length /= 1 or pilots[pilots'first].age /= 20) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(not (age=10));
  if (pilots'length /= 1 or pilots[pilots'first].age /= 20) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(age=10 or age=20);
  if (pilots'length /= 2) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(age=10 or age=20 or age>1);
  if (pilots'length /= 2) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(age=10 and age=20);
  if (pilots'length /= 0) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(age=10 xor age/=10);
  if (pilots'length /= 2) then
    raise Standard::constraint_error;
  end if;
  pilots := find Pilot(age=10 xor age=10 and age>1);
  if (pilots'length /= 0) then
    raise Standard::constraint_error;
  end if;

  //cleanup
  pilots := find_all Pilot();
  delete pilots;

  //various ordered_by conditions on finds
  pilot := create Pilot(age=>7);
  pilot := create Pilot(age=>3);
  pilot := create Pilot(age=>8);
  pilot := create Pilot(age=>1);
  pilot := create Pilot(age=>6);
  pilot := create Pilot(age=>2);
  pilot := create Pilot(age=>5);
  pilot := create Pilot(age=>4);
  pilot := create Pilot(age=>10);
  pilot := create Pilot(age=>9);
```

```
xpilots := find_all Pilot() ordered_by(age);
temp := 1;
for pilot in xpilots'elements loop
  if (pilot.age /= temp) then
    raise Standard::constraint_error;
  end if;
  temp := temp+1;
end loop;

xpilots := find_all Pilot() reverse_ordered_by(age);
temp := 1;
for pilot in xpilots'elements loop
  if (pilot.age = temp) then
    raise Standard::constraint_error;
  end if;
  temp := temp+1;
end loop;

pilots := find_all Pilot();
xpilots := pilots ordered_by(age);
temp := 1;
for pilot in xpilots'elements loop
  if (pilot.age /= temp) then
    raise Standard::constraint_error;
  end if;
  temp := temp+1;
end loop;

pilots := find_all Pilot();
xpilots := pilots reverse_ordered_by(age);
temp := 1;
for pilot in xpilots'elements loop
  if (pilot.age = temp) then
    raise Standard::constraint_error;
  end if;
  temp := temp+1;
end loop;

//cleanup
pilots := find_all Pilot();
```

```
  delete pilots;


  //*************************************************************************
  //various multiple ordered_by conditions on finds
  pilot := create Pilot(age=>3, NI=>455, name=>"fred",     qualified=>tod
  pilot := create Pilot(age=>3, NI=>222, name=>"fred",     qualified=>tod
  pilot := create Pilot(age=>8, NI=>455, name=>"sid",      qualified=>tod
  pilot := create Pilot(age=>8, NI=>455, name=>"bill",     qualified=>tod
  pilot := create Pilot(age=>6, NI=>455, name=>"harvey",   qualified=>tod
  pilot := create Pilot(age=>2, NI=>455, name=>"bob",      qualified=>tod
  pilot := create Pilot(age=>5, NI=>455, name=>"bernard",  qualified=>tod
  pilot := create Pilot(age=>4, NI=>455, name=>"baldrick", qualified=>tod
  pilot := create Pilot(age=>4, NI=>455, name=>"bonsai",   qualified=>tod
  pilot := create Pilot(age=>9, NI=>455, name=>"biggles",  qualified=>tod

  xpilots := find_all Pilot() ordered_by(age,name,NI);

  for x in 0..8 loop
    if (xpilots[x].age > xpilots[x+1].age) then
      raise Standard::constraint_error;
    end if;
  end loop;
  if (xpilots[1].NI /= 222) then
    raise Standard::constraint_error;
  end if;
  if (xpilots[8].name /= "sid") then
    raise Standard::constraint_error;
  end if;

  //xpilots := find_all Pilot() ordered_by(age,name,NI,qualified);

  xpilots := find_all Pilot() ordered_by(age,name,NI,qualified.year);

  xpilots := xpilots reverse_ordered_by(age,name,NI,qualified.year);
end;
```

8.7. Control Statements

```
declare
  i    : integer := 1;
  b    : integer := 0;
  t    : string  := "hello";
  f    : boolean := true;
  date : Calendar::Date;
  time : Calendar::Time_of_Day;
  ds   : sequence of record_structure_type;
  ia   : array (octal_type'range) of string;
  ea   : array (colour_type'range) of string;
  p    : instance of Pilot;
  ins  : sequence of instance of Pilot;
begin
  //if statement
  if (i/=1) then
    raise Standard::constraint_error;
  end if;
  if (i=1) then
    i:= 2;
  end if;
  if (i/=2) then
    raise Standard::constraint_error;
  end if;

  if (i/=2) then
    raise Standard::constraint_error;
  else
    i := 3;
  end if;
  if (i/=3) then
    raise Standard::constraint_error;
  end if;
  if (i=3) then
    i := 4;
  else
    raise Standard::constraint_error;
  end if;
  if (i/=4) then
    raise Standard::constraint_error;
  end if;
```

```
if (i/=4) then
  raise Standard::constraint_error;
elsif (i/=4) then
  raise Standard::constraint_error;
end if;
if (i/=4) then
  raise Standard::constraint_error;
elsif (i=4) then
  i := 5;
end if;
if (i/=5) then
  raise Standard::constraint_error;
end if;

if (i/=5) then
  raise Standard::constraint_error;
elsif (i=5) then
  i := 6;
else
  raise Standard::constraint_error;
end if;
if (i/=6) then
  raise Standard::constraint_error;
end if;

if (i/=6) then
  raise Standard::constraint_error;
elsif (i/=6) then
  raise Standard::constraint_error;
else
  i := 7;
end if;
if (i/=7) then
  raise Standard::constraint_error;
end if;

if f then
  f := false;
end if;
if f then
```

```
    raise Standard::constraint_error;
  end if;

  //case statement
  i := 3;
  case i is
    when 1=>
      b:=1;
    when 2=>
      b:=2;
    when others=>
      b:=1000;
  end case;
  if (b/=1000) then
    raise Standard::constraint_error;
  end if;
  i := 1;
  case i is
    when 1=>
      b:=1;
    when 2=>
      b:=2;
    when others=>
      b:=1000;
  end case;
  if (b/=1) then
    raise Standard::constraint_error;
  end if;
  i := 2;
  case i is
    when 1=>
      b:=1;
    when 2=>
      b:=2;
    when others=>
      b:=1000;
  end case;
  if (b/=2) then
    raise Standard::constraint_error;
  end if;
  i := 2;
  case i is
    when 1 | 2=>
      b:=1;
    when 3=>
      b:=3;
    when others=>
      b:=1000;
  end case;
  if (b/=1) then
    raise Standard::constraint_error;
  end if;

  //while statement
  i:=0;
  b:=0;
  while (i>0) loop
    i := i-1;
    b := b+1;
  end loop;
  if (b/=0) then
    raise Standard::constraint_error;
  end if;
  i:=0;
  b:=0;
  while (i>=0) loop
    i := i-1;
    b := b+1;
  end loop;
  if (b/=1) then
    raise Standard::constraint_error;
  end if;
  i:=2;
  b:=0;
  while (i>=1) loop
    i := i-1;
    b := b+1;
  end loop;
  if (b/=2) then
    raise Standard::constraint_error;
  end if;
  //exit statement
  i:=0;
  b:= 0;
  while (i=0) loop
    b:=b+1;
    exit;
  end loop;
  if (b/=1) then
    raise Standard::constraint_error;
  end if;
  i:=0;
  b:= 0;
  while (i=0) loop
    b:=b+1;
    exit when b=2;
  end loop;
  if (b/=2) then
    raise Standard::constraint_error;
  end if;

  //for statement, int loop
  b:=0;
  for i in 1..10 loop
    b:=b+1;
    if (b/=i) then
      raise Standard::constraint_error;
    end if;
  end loop;
  if (b/=10) then
    raise Standard::constraint_error;
  end if;
  b:=0;
  for i in reverse 1..10 loop
    b:=b+1;
    if (b=i) then
      raise Standard::constraint_error;
    end if;
  end loop;
  if (b/=10) then
    raise Standard::constraint_error;
  end if;
```

```
//for statement, derived and subtypes of int loop
b:=0;
for i in octal_type'range loop
  if (b/=i) then
    raise Standard::constraint_error;
  end if;
  b:=b+1;
end loop;
if (b/=8) then
  raise Standard::constraint_error;
end if;

//for statement, enum loop
t := "";
for e in SECOND..DAY loop
  t := t & "," & e'image;
end loop;
if (t/=",SECOND,MINUTE,HOUR,DAY") then
  raise Standard::constraint_error;
end if;
t := "";
for e in reverse SECOND..DAY loop
  t := t & "," & e'image;
end loop;
if (t/=",DAY,HOUR,MINUTE,SECOND") then
  raise Standard::constraint_error;
end if;

//for statement, enum type loop
t := "";
for e in Time_Unit'range loop
  t := t & "," & e'image;
end loop;
if (t/=",MILLISECOND,SECOND,MINUTE,HOUR,DAY") then
  raise Standard::constraint_error;
end if;

//for statement, data set loop
for x in 1..3 loop
  ds := ds & (x,1.2,x'image,date,time,blue);
end loop;
t := "";
for x in ds'elements loop
  t := t & x.text_field;
end loop;
if (t /= "123") then
  raise Standard::constraint_error;
end if;
t := "";
for x in ds'range loop
  t := t & ds[x].text_field;
end loop;
if (t /= "123") then
  raise Standard::constraint_error;
end if;
t := "";
for x in ds'first..ds'last loop
  t := t & ds[x].text_field;
end loop;
if (t /= "123") then
  raise Standard::constraint_error;
end if;
t := "";
for x in ds'last..ds'first loop
  t := t & ds[x].text_field;
end loop;
if (t /= "") then
  raise Standard::constraint_error;
end if;
t := "";
for x in reverse ds'elements loop
  t := t & x.text_field;
end loop;
if (t /= "321") then
  raise Standard::constraint_error;
end if;
t := "";
for x in reverse ds'range loop
  t := t & ds[x].text_field;
end loop;
if (t /= "321") then
```

```
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ds'first..ds'last loop
    t := t & ds[x].text_field;
  end loop;
  if (t /= "321") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ds'last..ds'first loop
    t := t & ds[x].text_field;
  end loop;
  if (t /= "") then
    raise Standard::constraint_error;
  end if;

  //for statement, array loop integer index
  for x in octal_type'range loop
    ia[x] := x'image;
  end loop;
  t := "";
  for x in ia'elements loop
    t := t & x;
  end loop;
  if (t /= "01234567") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in ia'range loop
    t := t & ia[x];
  end loop;
  if (t /= "01234567") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in ia'first..ia'last loop
    t := t & ia[x];
  end loop;
  if (t /= "01234567") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in octal_type'range loop
    t := t & ia[x];
  end loop;
  if (t /= "01234567") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in octal_type'first..octal_type'last loop
    t := t & ia[x];
  end loop;
  if (t /= "01234567") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in ia'last..ia'first loop
    t := t & ia[x];
  end loop;
  if (t /= "") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in octal_type'last..octal_type'first loop
    t := t & ia[x];
  end loop;
  if (t /= "") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ia'elements loop
    t := t & x;
  end loop;
  if (t /= "76543210") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ia'range loop
    t := t & ia[x];
  end loop;
  if (t /= "76543210") then
```

```
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ia'first..ia'last loop
    t := t & ia[x];
  end loop;
  if (t /= "76543210") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse octal_type'range loop
    t := t & ia[x];
  end loop;
  if (t /= "76543210") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse octal_type'first..octal_type'last loop
    t := t & ia[x];
  end loop;
  if (t /= "76543210") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ia'last..ia'first loop
    t := t & ia[x];
  end loop;
  if (t /= "") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse octal_type'last..octal_type'first loop
    t := t & ia[x];
  end loop;
  if (t /= "") then
    raise Standard::constraint_error;
  end if;

  //for statement, array loop enum index
  for x in colour_type'range loop
    ea[x] := x'image;
  end loop;
  t := "";
  for x in ea'elements loop
    t := t & x;
  end loop;
  if (t /= "redgreenblue") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in ea'range loop
    t := t & ea[x];
  end loop;
  if (t /= "redgreenblue") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in ea'first..ea'last loop
    t := t & ea[x];
  end loop;
  if (t /= "redgreenblue") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in colour_type'range loop
    t := t & ea[x];
  end loop;
  if (t /= "redgreenblue") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in colour_type'first..colour_type'last loop
    t := t & ea[x];
  end loop;
  if (t /= "redgreenblue") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in ea'last..ea'first loop
    t := t & ea[x];
  end loop;
  if (t /= "") then
```

```
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in colour_type'last..colour_type'first loop
    t := t & ea[x];
  end loop;
  if (t /= "") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ea'elements loop
    t := t & x;
  end loop;
  if (t /= "bluegreenred") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ea'range loop
    t := t & ea[x];
  end loop;
  if (t /= "bluegreenred") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ea'first..ea'last loop
    t := t & ea[x];
  end loop;
  if (t /= "bluegreenred") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse colour_type'range loop
    t := t & ea[x];
  end loop;
  if (t /= "bluegreenred") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse colour_type'first..colour_type'last loop
    t := t & ea[x];
  end loop;
  if (t /= "bluegreenred") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ea'last..ea'first loop
    t := t & ea[x];
  end loop;
  if (t /= "") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse colour_type'last..colour_type'first loop
    t := t & ea[x];
  end loop;
  if (t /= "") then
    raise Standard::constraint_error;
  end if;

  //for statement, instance set loop
  p:= create Pilot(age=>20,name=>"1");
  ins := ins & p;
  p:= create Pilot(age=>20,name=>"2");
  ins := ins & p;
  p:= create Pilot(age=>20,name=>"3");
  ins := ins & p;
  t := "";
  for x in ins'elements loop
    t := t & x.name;
  end loop;
  if (t /= "123") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in ins'range loop
    t := t & ins[x].name;
  end loop;
  if (t /= "123") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in ins'first..ins'last loop
```

```
    t := t & ins[x].name;
  end loop;
  if (t /= "123") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in ins'last..ins'first loop
    t := t & ins[x].name;
  end loop;
  if (t /= "") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ins'elements loop
    t := t & x.name;
  end loop;
  if (t /= "321") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ins'range loop
    t := t & ins[x].name;
  end loop;
  if (t /= "321") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ins'first..ins'last loop
    t := t & ins[x].name;
  end loop;
  if (t /= "321") then
    raise Standard::constraint_error;
  end if;
  t := "";
  for x in reverse ins'last..ins'first loop
    t := t & ins[x].name;
  end loop;
  if (t /= "") then
    raise Standard::constraint_error;
  end if;
end;
```

8.8. Arrays

```
declare
  my_b_array        : array(1..2) of byte;
  my_int_array_1    : array(1..5) of integer;
  my_int_array_2    : my_int_array_type;
  my_int_array_3    : my_int_array_type;
  my_struct_array   : array(1..2)  of nested_structure_type;
  my_time_array     : array(2..3)  of Calendar::time_type;
  my_sub_type_array : array(3..30) of my_int_sub_type_type;
  my_array_array    : array(1..5)  of my_int_array_type;
  x                 : integer:=0;
  s                 : nested_structure_type;
begin
  for i in 1..10 loop
    my_int_array_2[i] := 9;
    my_int_array_3[i] := 10;
  end loop;
  if (my_int_array_2[5] /= 9) then
    raise Standard::constraint_error;
  end if;

  //general access
  my_int_array_1[1] := 55;
  if (my_int_array_1[1] /= 55) then
    raise Standard::constraint_error;
  end if;

  my_int_array_1[my_int_array_1'first] := 55;
  if (my_int_array_1[1] /= 55) then
    raise Standard::constraint_error;
  end if;

  my_int_array_1[my_int_array_1'last] := 55;
  if (my_int_array_1[5] /= 55) then
    raise Standard::constraint_error;
  end if;
```

```
//parser error
//my_int_array_1[my_int_array_type'last] := 55;

//copy
my_int_array_2 := my_int_array_3;
my_int_array_3[4] := 0;

//slices, parser error
for i in 1..5 loop
  my_int_array_1[i] := i;
end loop;
//my_int_array_1 := my_int_array_2[1..5];
x := 0;
for i in my_int_array_1'elements loop
  x:=x+i;
end loop;
//if (x /= 5) then
//  raise Standard::constraint_error;
//end if;

//arrays as parameters
Array_passing(my_int_array_2,my_int_array_3);

//array of strucures
my_struct_array[2].id := 9;
s := my_struct_array[2];
if (s.id /= 9) then
  raise Standard::constraint_error;
end if;

//array of arrays
my_array_array[1][1] := 55;
my_int_array_2 :=  my_array_array[1];
if (my_int_array_2[1] /= 55) then
  raise Standard::constraint_error;
end if;

//expected exceptions
x := 0;
begin
```

```
    my_b_array[10] := 0;
  exception
    when Standard::constraint_error =>
      x := x + 1;
  end;
  begin
    my_array_array[0][1] := 0;
  exception
    when Standard::constraint_error =>
      x := x + 1;
  end;
  begin
    my_array_array[1][0] := 0;
  exception
    when Standard::constraint_error =>
      x := x + 1;
  end;
  if (x /= 3) then
    raise Standard::constraint_error;
  end if;
end;
```

8.9. Exceptions

Exception type declared in the domain mod file.

```
Exception my_exception
```

```
declare
  handled : boolean;
begin
  handled := false;
  begin
    raise my_exception;
  exception
    when my_exception =>
```

```
      handled := true;
  end;
  if (not handled) then
    raise Standard::constraint_error;
  end if;
end;
```

8.10. Device Output

```
declare
  s : string;
  i : integer;
  r : real;
  b1 : boolean;
  b2 : boolean;
  e : Calendar::month_type;
  test : device;
  res : string;

begin

  s := "Hello";
  i := 123;
  r := 45.67;
  b1 := true;
  b2 := false;
  e := Calendar::MARCH;

  // Test single writes
  Device_IO::create_file("output.tst", true);
  Device_IO::open("output.tst", Device_IO::OUT, test);
  test << s;
  test << i;
  test << r;
  test << b1;
  test << b2;
  test << e;
  Device_IO::close(test);
  Device_IO::open("output.tst", Device_IO::IN, test);
  test >> res;
  Device_IO::close(test);

  if res /= "Hello12345.67truefalseMARCH" then
    console << res;
    raise Standard::constraint_error;
  end if;

  // Test concatenated writes
  Device_IO::create_file("output.tst", true);
  Device_IO::open("output.tst", Device_IO::OUT, test);
  test << s << i << r << b1 << b2 << e;
  Device_IO::close(test);

  Device_IO::open("output.tst", Device_IO::IN, test);
  test >> res;
  Device_IO::close(test);

  if res/= "Hello12345.67truefalseMARCH" then
    console << res;
    raise Standard::constraint_error;
  end if;

  Device_IO::delete_file("output.tst");

  console << "TEST PASSED" << endl;

end;
```

8.11. Device Input

```
declare
 s : string;
  i : integer;
  r : real;
  b1 : boolean;
  b2 : boolean;
  e1 : Calendar::month_type;
  e2 : Calendar::month_type;
  test : device;
  res : string;
  p1 : positive;
  p2 : positive;
 passed : boolean;
begin

  // Test single element reads
  Device_IO::create_file("input.tst", true);
  Device_IO::open("input.tst", Device_IO::OUT, test );
  test <<  "Hello 123   45.67   true false \n MARCH 2 -2 aaa";
  Device_IO::close(test);

  Device_IO::open("input.tst", Device_IO::IN, test );

  test >> s;
  if s /= "Hello" then
    raise Standard::constraint_error;
  end if;

  test >> i;
  if i /= 123 then
    raise Standard::constraint_error;
  end if;

  test >> r;
  if r < 45.66669 or r > 45.67001 then
    raise Standard::constraint_error;
  end if;

  test >> b1;
  if not b1 then
    raise Standard::constraint_error;
  end if;

  test >> b2;
  if b2 then
    raise Standard::constraint_error;
  end if;

  test >> e1;
  if e1 /= Calendar::MARCH then
    raise Standard::constraint_error;
  end if;

  test >> p1;
  if p1 /= 2 then
    raise Standard::constraint_error;
  end if;

  begin
    passed := false;
    test >> p2;
  exception
    when Standard::constraint_error =>
      passed := true;
  end;
  if passed = false then
    raise Standard::constraint_error;
  end if;

  begin
    passed := false;
    test >> e2;
  exception
    when Standard::constraint_error =>
      passed := true;
  end;
  if passed = false then
    raise Standard::constraint_error;
  end if;

  if Device_IO::eof(test) = false then
    raise Standard::constraint_error;
  end if;

  Device_IO::close(test);

  // Test concatenated reads
  Device_IO::open("input.tst", Device_IO::IN, test );
  test >> s >> i >> r >> b1 >> b2 >> e1 >> p1;

  if s /= "Hello" then
    raise Standard::constraint_error;
  end if;

  if i /= 123 then
    raise Standard::constraint_error;
  end if;

  if r < 45.66669 or r > 45.67001 then
    raise Standard::constraint_error;
  end if;

  if not b1 then
    raise Standard::constraint_error;
  end if;

  if b2 then
    raise Standard::constraint_error;
  end if;

  if e1 /= Calendar::MARCH then
    raise Standard::constraint_error;
  end if;

  if p1 /= 2 then
    raise Standard::constraint_error;
  end if;
  Device_IO::close(test);
  Device_IO::delete_file("input.tst");

  console << "TEST PASSED" << endl;

end;
```

Index