DRAFT
Rapide 1.0 Executable Language Reference Manual

RAPIDE Design Team
Program Analysis and Verification Group
Computer Systems Lab
Stanford University

July 17, 1997

Preface

This manual is part of a suite of manuals on RAPIDE-1.0. The complete set of manuals is:

- Guide to the RAPIDE 1.0 Language Reference Manuals
- RAPIDE 1.0 Types Language Reference Manual
- RAPIDE 1.0 Architectures Reference Manual
- RAPIDE 1.0 Executable Language Reference Manual
- RAPIDE 1.0 Event Pattern Language Reference Manual
- RAPIDE 1.0 Constraint Language Reference Manual
- RAPIDE 1.0 Predefined Types and Objects Reference Manual

The part of RAPIDE intended for use as a general purpose programming language is known as the RAPIDE 1.0 Executable Language (or *executable* for short). This reference manual describes the executable language, which includes module generators (the general mechanism for implementing objects in RAPIDE) and most constructs which can be used within module generators. These include expressions, if, case and looping statements, statements for reacting to and generating patterns, functions, exception handling constructs, processes, and clocks.

Static visibility rules are also defined in this manual.

The manual proper consists of the syntax and semantics sections of each chapter. Examples and commentaries are for explanatory purposes only.

Contents

List of Figures

Chapter 1

Introduction

This reference manual defines the RAPIDE 1.0 Executable Language. The Executable Language defines the module generator construct and its enclosed code. It is this code which is executed when a module generator is called to construct a module, and thereafter performs the computation. It is thus the role of the Executable Language to provide implementations (module generators) for the interface types defined in the RAPIDE 1.0 Types Language.

1.1 Organization

This reference manual is divided into the following parts:

1. Chapter 2 defines the static visibility rules of RAPIDE.
2. Chapter 3 defines declarations.
3. Chapter 4 describes the expressions, qualified expressions (type casting), and attributes.
4. Chapter 5 describes sequential statements.
5. Chapter 6 describes function declarations, function conformance rules, and function calling semantics.
6. Chapter 7 describes processes, reactive statements (await and when), and event generation semantics (action call).
7. Chapter 8 describes exceptions and interrupts.
8. Chapter 9 describes modules, module conformance rules, module inheritance, and module initialization and finalization.
9. Chapter 10 describes clocks and timed statements (in, pause, and delay).

1.2 Notational Conventions

Grammar productions are given in the form:

```
non_terminal ::=
        rhs_1
|       rhs_2
|       ...
|       rhs_n
```

The rhs_i (*rhs* stands for right hand side) are a sequence of terminals, non-terminals and the following constructs:

`[ rhs ]`	zero or one occurrences of `rhs`
`{ rhs }`	zero or more occurrences of `rhs`
`( rhs )`	grouping
$\texttt{rhs}_1$ | ... | $\texttt{rhs}_n$	choice

Keywords are written in **boldface**. Terminals (other than keywords) are written in all UPPERCASE.

Chapter 2

Visibility and Overloading

Names in RAPIDE are either *identifiers* or *placeholders*. *Visibility* defines where a name is visible in a RAPIDE program. The rules of visibility are compiletime rules defined in the Algol tradition in terms of the concepts of *declarative region* and *scope*. Ultimately, the purpose of visibility is to allow the reuse of names in different scopes. A compiler can decide whether a name is visible at a given position, and which declaration of a name is being referred to. Visibility is said to be a *static* concept.

A name may be *overloaded* in RAPIDE by declaring it in more than one declaration in the same declarative region, providing the declarations differ so that any use of the name can uniquely determine which declaration is being referred to. Rules for overloading are described in this chapter.

RAPIDE visibility rules are permissive. That is, as discussed in Section 2.3 of the Guide to Rapide, a module should comunicate only by accessing its own interface features. But this is too restrictive to allow easy use of predefined types, for example. So, the visibility rules also allow a module direct visibility to other modules in its declarative scope (e.g., other components of the architecture in which it is a component), and also indirect visibility to the modules that are visible to its parent. If this visibility is not used with care, the communication in an interface connection architecture can be subverted, leading to a system of modules that does not conform to a given architecture.

This chapter explains the visibility rules of RAPIDE at a global level; it depends upon the separate Reference Manuals to explain the rules for each particular kind of declarative region. The special visibility rules for two experimental constructs of RAPIDE are not described here: Maps and Bindings. These are described in the Architectures LRM. The overloading rules are also explained here.

2.1 Declarative Regions

Static visibility of names is defined in terms of the concept of *declarative region*, which is an area of program text where declarations may occur, and the associated concept of the *scope* of a declaration.

A declaration is a construct that declares a name and gives the name a denotation (or meaning). [1] A declarative region is a region of program text whose syntactic definition allows one

[1] A name declaration is a special kind of declaration; it is an incomplete declaration where the denotation of the name it declares is given in another declaration.

or more lists of declarations to be placed at specific positions in the region. These declarations are said to be *associated* with the region. Identical declarations are not allowed in the same list.

There are four classes of declarative regions in RAPIDE:

1. **Interface type expressions:** declarative regions that define types. Interface type expressions are defined in Chapter 4 of the Types LRM. All the declarations in an interface type expression are associated with the region.

   Also, the predefined types described in Predefined Types LRM are all instances of interface types and likewise define declarative regions. Some predefined types have special syntax:

   - records (Predefined Types LRM Chapter 9),
   - unions (Predefined Types LRM Chapter 10),
   - enumeration types (Predefined Types LRM Chapter 11),
   - event types (Predefined Types LRM Chapter 19).

2. **Parameterized constructors:** declarative regions consisting of a list of parameter declarations , a declarations part containing a list of declarations, and a body. The parameter declarations and the declarations in the declarations part are associated with the region. The parameterized constructors are:

   - type constructors (Types LRM Chapter 6),
   - functions (Executable LRM Chapter 6),
   - module generators (Executable LRM Chapter 9),
   - architecture generators (Architectures LRM Chapter 3).

3. **Parameterized name declarations:** declare only a name and a list of parameters. In some cases a full declaration will be provided later that completes the name declaration by supplying a denotation. The parameter declarations are associated with the region. The parameterized name declarations are:

   - function name declarations (Executable LRM Chapter 6)
   - action declarations (Architectures LRM Section 2.4)
   - exception declarations (Executable LRM Chapter 8)
   - type constructor name declarations (Types LRM Section 4.2)
   - module generator name declarations (Executable LRM Chapter 9)

4. **Blocks:** declarative regions consisting of a declarative part containing a list of declarations and a body. The declarations in the declarative part are associated with the region. The blocks are:

   - the behavior part of an interface (Architectures LRM Section 2.6),
   - transition rules (Architectures LRM Section 2.6),
   - module and architecture literals (Executable LRM Chapter 9),
   - do statements (Executable LRM Section 5.3),
   - case statements (Executable LRM Section 5.6),
   - await statements (Executable LRM Section 7.3.1),
   - when statements (Executable LRM Section 7.3.2),
   - process generators (Executable LRM Section 7.4),
   - connection generators (Architectures LRM Section 3.4),
   - connection rules (Architectures LRM Section 3.4),
   - constraints (Constraint LRM Chapter 1).

The form and content of each of these declarative regions is defined in the relevant LRM.

2.1.1 Nesting of declarative regions

A declarative region may be placed inside of another declarative region, in which case it is said to be an *inner* region, and to be *nested* inside of the enclosing region. Declarations associated with a declarative region are *local* to the region, and declarations in enclosing regions are *global* to the inner region.

A declaration of a name in an inner region may *hide* a declaration of the same name in an outer region. In this case the outer declaration is *indirectly visible* in the scope of the inner declaration, and may be referred to in that scope by *qualifying* the name with the name of the outer region. This permits both denotations of the name to be used in the scope of the inner declaration. The concepts of hiding and name qualification are explained in Section 2.3.

2.1.2 Examples

Example: *Interface type expression*

```
interface
    provides
        Out_Object : T;
    requires
        In_Object : T;
end Interface
```

Commentary:

An interface type expression is a declarative region that may contain several lists of declarations, associated with different parts of the interface such as **provides**, **requires**, **action**, etc. The interface above has one list of declarations consisting of the declaration Out_Object, and another list consisting of the declaration In_Object.

□

Example: *Parameterized constructor declarative region*

```
function Withdraw( A : Account; D : Dollar ) is
    ...  -- local declarations
begin
    ...
end;
```

Commentary:

A parameterized constructor such as a function declaration is a declarative region that may contain a list of parameter declarations, as well as a list of local declarations.

□

Example: *Nested declarative regions*

```
module Stack(...) return Stack_Type is
```

```
        ...                 -- declarations.
    parallel
        ...
    Region1: declare        -- an inner region.
                ...         -- declarations local to Region1.
             do
                ...
             end do;
        ...
    end;
```

Commentary:

> The module generator Stack is a declarative region. The declarations associated with it appear immediately after the keyword, **is**. The region contains a body (in this case a list of parallel processes following the keyword, **parallel**) which contains a nested declarative region called Region1. This is a labeled do-statement which contains its own local declarations. The declarations of Stack are *global* to Region1.

□

2.2 Scope of a Declaration

Every declaration has a *scope* which is a region of program text in which it is visible (Section 2.3). The scope of a declaration is determined syntactically by its associated declarative region and the position of that region. The scope of a declaration is also the scope of the names it declares.

The *immediate scope* of a declaration starts at the beginning of the declaration in the parameter list (if any) or declaration list of its associated declarative region, and extends to the end of the region. This implies that a declaration is visible to later declarations in the same region, but not conversely — i.e., *linear visibility* of the list of declarations in their associated region.

Some declarations have an *extended scope* that extends beyond their immediate scope. These declarations, and their extended scopes, are:

1. a visible declaration in an interface type expression that is part of a type declaration, an object declaration, a type constructor declaration, a parameter declaration, or a placeholder declaration. In these cases the scope of a visible declaration in an interface type expression includes the scope of the declaration in which the interface type expression occurs.

2. a parameter declaration. The scope of the parameter declaration includes the scope of the declaration in which it occurs as a parameter. Parameter declarations can occur in:

   - a function type expression in a function declaration,
   - a module or architecture generator declaration,
   - a type constructor declaration.

The scope of a declaration consists of its immediate scope and its *extended scope*, if any.

2.2.1 Names of scopes

A declarative region is given a name if, in certain cases it is part of a declaration, or if it is labelled. This name is called the *scope name* of declarations associated with the region.

A declarative region, and the scope of the declarations associated with it, is *named* in the following cases:

- An interface type expression, if it is given as the denotation of a type or type constrcutor declaration; the type or type constructor name is the scope name.
- A module or architecture generator, or module or architecture name generator; the module or architecture name is the scope name.
- A map or binding generator; the map or binding name is the scope name.
- A do-statement, if it is labeled; the label is the scope name.

The name S of the scope of a declaration can be used to refer to a name N declared by that declaration by means of the *scope name* notation, S :: N (see Section 2.3).

2.2.2 Examples

Example: *Scope of a declaration*

```
Region1: declare
    Value : Integer is Index;   -- illegal use of Index.
    Index : var Integer := 1;
do
    ...                         -- Index is visible here.
end do;
... Index ...             -- illegal, Index is not visible here.
```

Commentary:

The declaration of Index (and the name Index) is associated with the declarative region labelled Region1 which is a **do** statement; because the declarative region is named, the *scope name* of Index is Region1.

The scope of Index begins at the declaration of Index itself, and extends to the end of the **do** statement. This is the immediate scope of Index, and it has no extended scope beyond its immediate scope. Anywhere outside of its scope, for example in the declaration of Value and the in text following Region1, Index is not visible (see below).

□

Example: *Extended scope*

```
Plane : interface
            Loc   : Location;
            Speed : Knots;
        end Interface;
```

Commentary:

Constituents of the interface type of the Plane object have an *immediate* scope, namely the interface type expression itself, and an *extended* scope, namely the scope of the declaration of Plane in which the type expression appears. So the extended scope of Loc and Speed is the scope of Plane.

□

2.3 Static Visibility Rules

Visibility rules define where a declaration is visible and how to refer to the names declared in a declaration.

A declaration is visible within its scope, either *directly* or *indirectly*. A declaration is not visible outside of its scope.

The direct or indirect visibility of a declaration within its scope is defined by the following conditions:

1. Any declaration is directly visible within its immediate scope (Section 2.2) unless it is *hidden* (as defined below) in an inner declarative region by a local declaration. A declaration may be indirectly visible in a nested region where it is hidden, as described below.

2. A visible declaration of an interface type expression that has an extended scope as described in the Section 2.2 is indirectly visible in the extended part of its scope.

3. A parameter declaration as described in the Section 2.2 is directly visible in (function, module generator, or type constructor) calls in the extended part of its scope, and is not visible anywhere else in the extended part of its scope.

A global declaration is hidden by a local declaration in the immediate scope of that local declaration if the local declaration declares the same name and *conflicts* with the global declaration. Two declarations that give similar denotations to names, or otherwise serve to declare names with similar roles (e.g., labels), are said to *conflict* if they both belong to one of the five conflict classes listed below:

1. *Declarations of names that denote objects:* Object declaration, Object name declaration, object parameter declaration, module generator declaration, module name generator declaration, enumeration literal declaration of an enumeration type, placeholder declaration, service declaration, case expression declaration, map domain declaration.

2. *Declarations of names that denote types:* Type declaration, Type name declaration, type constructor declaration, type constructor name declaration, type parameter declaration.

3. *Declarations of names that denote patterns:* Action name declaration, Pattern macro declaration, pattern parameter declaration, exception name declaration.

4. *Declarations of Labels:* Label declarations occur as part of a labeled statement or a labeled constraint.

5. *Declarations of names that denote Maps:* Map generator declaration, map declaration.

Wherever a declaration is directly visible, the names it declares may be used (e.g., in expressions) without any prefix.

Wherever a declaration is indirectly visible the names it declares must be qualified as follows.

- **In an extended scope:** If the extended scope is the scope of an object declaration, then the visible names declared in the interface type expression in the object declaration must be prefixed with the object name using the *selection* operation.

  For example, if N is declared in a visible declaration of an interface type expression which is part of the declaration of an object, O, then O.N refers to the visible constituent of O denoted by N.

- **Where hidden in a part of an immediate scope:** The names of a global declaration must be referred to in the scope of a nested local declaration that hides the global declaration by prefixing them using scope name notation with the name (e.g., label) of the declarative region associated with the global declaration.

  For example, if N is declared in a global region which has a label L, and it is hidden in a nested region by a local declaration of N, then within the nested region L :: N (Section 2.2.1) refers to the denotation of the global declaration.

  If there is no name for the region associated with the global declaration, then its denotation for the names it declares cannot be used within the region where it is hidden.

2.3.1 Examples

Example: *Selection*

```
type Driving_Record is interface
provides
    Name : String;
end interface;

MyRec : Driving_Record is ...;
A_Name : String is MyRec.Name;
```

Commentary:

In the declaration of MyRec Driving_Record is a type expression. The declaration of name Name is associated with this expression; it denotes a visible String constituent of MyRec. It is indirectly visible in its extended scope — that is, in the scope of MyRec — using the selection operation. MyRec.Name denotes the String constituent of MyRec.

□

Example: *Hiding and using scope names*

```
module Stack() return Stack_Type is
    Index : var Integer;              -- (1)
initial
    Inner: declare
```

```
        Index : var Integer;          -- (2)
    do
        ... Index                     -- refers to (2)
        ... Stack::Index              -- refers to (1)
        ... Inner::Index              -- refers to (2)
        ... Stack::Inner::Index       -- refers to (2)
    end;
    Index                             -- refers to (1).
    ...
end Stack;
```

Commentary:

The declaration of Index in the inner declarative region (called Inner) hides the outer declaration of Index. The scope name, Stack, which is the name of the scope associated with the outer declaration, may be used to refer to the outer declaration where it is hidden.

In the inner region, the do statement, the expression Index refers to the inner declaration. The expression, Stack::Index, refers to the outer declaration. Inner::Index and Stack::Inner::Index both refer to the name declared in the Inner region.

□

Example: *Visibility of parameters in calls*

```
function Withdraw( A : Account; D : Dollar );
...
Withdraw( A is Account1, D is 100.00 );
```

Commentary:

Within the call to function Withdraw, the parameters A and D are directly visible.

□

2.4 Static Visibility of Predefined Types and Separately Compiled Declarations

A RAPIDE declaration that is not nested in any declarative region is called an *outermost* declaration. The outermost declarations comprising a RAPIDE program form an implicit declarative region. For maximum visibility, the predefined types (Predefined Types LRM) are treated as if they are explicitly declared at the beginning of the outermost declarations of a RAPIDE program. The declarations of predefined types and the outermost declarations are visible according to the visibility rules (Section 2.3) applied to the implicit outermost declarative region.

In addition, separately compiled declarations may be added to the outermost declarative region of a RAPIDE program, including for example, the predefined Input/Output module. How this is done is defined by a library management system for composing sets of separately compiled RAPIDE declarations that reside in external files.

The separately compiled and outermost declarations of a RAPIDE program may be declarations of:

- types (Types LRM Chapter 6),
- type constructors (Types LRM Chapter 6),
- objects,
- functions (Executable LRM Chapter 6),
- module generators (Executable LRM Chapter 9),
- architecture generators (Architectures LRM Chapter 3),
- map generators (Architectures LRM).

Example: *Visibility to Predefined Declarations*

B : Boolean **is** False;

Commentary:

Boolean and False are predefined declarations defined in the Predefined Types LRM. Users may assume the existence of such predefined declarations.

□

2.5 Overloading of Declarations

A name may be *overloaded* by being declared in more than one declaration associated with the same declarative region, providing the declarations differ so that any use of the name can uniquely determine which declaration is being referred to. Rules defining the conditions under which overloading is allowed, and for determining the denotation of a use of an overloaded name (called *overload resolution*) are described in this Section.

2.5.1 Rules for Overloading

A name may be overloaded by being declared in two declarations associated with the same declarative region if the declarations belong to different conflict classes (see Section 2.3). For example, the same name may be declared as a type name, an object name, a label, etc. in a single declarative region.

In addition, a name may be overloaded by conflicting declarations in the following cases:

- An object name may be overloaded by being declared in two or more object or object name declarations that do not declare objects of the same type [2].
- Type names may not be overloaded.
- Map names may not be overloaded.
- Label names may not be overloaded.
- Pattern names may be overloaded in the following cases (see Patterns LRM):

  Two action name declarations may use the same name if the inferred event types are not equivalent.

  Two pattern macro declarations may use the same name if they do not have the same profile as described in Patterns LRM.

[2] Different type declarations may denote the same type e.g., if the declared types are subtypes of each other,

2.5.2 Overload Resolution

When a name is declared in two or more object or object name declarations associated with the same declarative region, overload resolution to determine the denotation of a use of the name is defined as follows:

1. There is a unique supertype among those types of the declarations of the name that are compatible with the given use of the name. The denotation given by the declaration with the unique supertype is chosen.
2. There is no unique supertype. The use of the name is said to be unresolvable. This will lead to a semantic error.

2.5.3 Examples

Example: *Overloading and Resolution*

```
function Put( I : Integer );
function Put( S : String );
...
Put( 10 );
```

Commentary:

Function Put is overloaded in this declarative region. The call to Put is resolved because its parameter, the integer literal 10, makes the call legal for only one of the functions.

□

Example: *Overload Resolutions – Unambiguous and Ambiguous*

```
Pt : Point is ...;
Pt : Colored_Point is ...;              -- Colored_Point is a subtype of Point.
Good_Point : Colored_Point is Pt;       -- legal
Bad_Point : Point is Pt;                -- illegal, ambiguous
```

Commentary:

Assume Colored_Point <: Point (i.e., Colored_Point is a subtype of Point).

Two objects may be declared with the same name in the same declarative region as long as they have distinct types. Thus, the two declarations of Pt are an allowed overloading. The declaration of Good_Point is legal because only the type Colored_Point satisfies it, so the denotation of the initializing object Pt is uniquely determined. The declaration of Bad_Point is illegal, because either denotation of Pt could be used, so there is no unique resolution.

□

Chapter 3

Declarations

This chapter defines *declarations*, which associate a name with an entity in the language; declarations give denotations to names.

Object declarations associate a name with an object. Some special cases of object declarations are covered in more detail in other chapters. Functions are dealt with more fully in Chapter 6. The traditional programming language concept of "variable" is implemented in RAPIDE using objects of reference types. Chapter 16 of the RAPIDE 1.0 Predefined Types and Objects Reference Manual describes reference types and reference objects, and how they implement the semantics of variables in more detail.

3.1 Declarations

A *declaration* associates a name with an entity in the language, and causes the associated entity to be *elaborated*. The name is said to *denote* the declared entity.

There are many kinds of entities that may be declared in RAPIDE; they are listed in the table in Figure 3.1.

Elaboration causes the entity to become available in the computation, and the associated name provides a way to refer to the entity. Wherever the language requires an entity of a particular kind, a name denoting an entity of that kind may appear.

3.2 Object Declaration

3.2.1 Purpose

An object declaration associates a basic name with an object of a given type. Lexically, a basic name is an identifier, an operator symbol as described in Chapter 4, or a placeholder as described in Section 2.2 of the RAPIDE 1.0 Event Pattern Language Reference Manual.

3.2.2 Syntax

Declaration	Location in LRMs
Object declaration	Chapter 3.2
Object name declaration	Chapter **??**
Object parameter declaration	Types, Chapter 5.2.2
Module generator	Chapter 9
Module name generator	Chapter **??**
Enumeration literal	Predefined, Chapter 11
Placeholder declaration	Patterns, Chapter 2.4.9
Service declaration	Architectures, Chapter 2.5
Case expression declaration	Chapter 5.6
Map domain declaration	Architectures, Chapter 5
Type declaration	Types, Chapter 3.2
Type name declaration	Chapter **??**
Type constructor declaration	Types, Chapter 6.1
Type constructor name declaration	Chapter **??**
Type parameter declaration	Types, Chapter 5.2.2
Action name declaration	Architectures, Chapter 2.4
Pattern macro declaration	Patterns, Chapter 2.5
Pattern parameter declaration	Patterns, Chapter 2.5
Exception name declaration	Chapter 8
Labeled statement	Chapter 5.1
Labeled constraint	Constraints, Chapter 1.2
Map generator declaration	Architectures, Chapter 5
Map declaration	Architectures, Chapter 5.3

Figure 3.1: Declarations and Where They Are Defined

object_declaration ::=
  basic_name_list ':' *type_expression* [**is** *expression*] ';'
  basic_name_list **is** *expression* ';'

basic_name_list ::=
  basic_name { ',' *basic_name* }

basic_name ::=
  identifier
| *placeholder*
| *operator_symbol*

3.2.3 Semantics

[1] **Straighten all this out: obj vs name decl, for w/no type.** The elaboration of an object declaration proceeds as follows:

1. The type expression is first elaborated. This establishes the type of the object.
2. If present, the expression is evaluated, and the name denotes the result of the expression evaluation.

3. If a basic name list appears in the object declaration, step 2 is repeated once for each basic name in the list. The basic names in the list are elaborated in the order in which the names appear in the list.

3.2.4 Examples

Example: *Object Declarations*

```
LIMIT      : INTEGER is 10000;          -- an integer object
LOW_LIMIT  : INTEGER is LIMIT/10;       -- another integer object
PI         : FLOAT is 3.14159;          -- a floating point object
VARLIMIT   : ref(INTEGER) is refto(25); -- an integer "variable" initialized
                                        -- to refer to the integer object 25
S1, S2     : STACK is NEW_STACK(LIMIT);
                                        -- The expression new_stack(LIMIT)
                                        -- is evaluated twice. S1 denotes
                                        -- the result of the first evaluation,
                                        -- and S2 denotes the result of the
                                        -- second evaluation.
```

□

3.3 Forward Declarations

Forward declarations allow the mutually recursive definitions of types and objects.

3.3.1 Syntax

object_declaration ::=
  basic_name_list ':' *type_expression* **is forward**] ';'

function_declaration ::=
  function *basic_name* '(' [*formal_parameter_list*] ')'
[**return** *type_expression*] **is forward** ';'

type_declaration ::=
  type *identifier* **is forward**] ';'

An object or type declaration that uses **forward** is called a "forward declaration;" any other object or type declaration is called a "full declaration."

3.3.2 Semantics

For each forward object declaration, there must be exactly one full object declaration with the same basic_name and with an equivalent type, which follows the forward declaration in the declarative region in which it occurs. The elaboration of a forward object declaration creates an object; its corresponding full declaration provides the value of the object.

For each forward type declaration, there must be exactly one full type declaration with the same identifier, which follows the forward declaration in the declarative region in which it occurs. The elaboration of a forward type declaration creates a type; however, information about the type is only visible in the current declarative region following the full declaration.

3.3.3 Examples

Example: *Recursive Types*

```
type Male is forward;

type Female is interface
    Husband_Of : Male;
end;

type Male is interface
    Wife_Of : Female;
end;
```

□

Example: *Recursive Functions*

```
function Skip( L : List ) return List is forward;

function Take( L : List ) return List is
    if L.Empty() then return Empty_List
    else return Append( L.Head(), Skip(L.Tail()) );
end;

function Skip( L : List ) return List is
    if L.Empty() then return Empty_List
    else return Take(L.Tail());
end;
```

Commentary:

> Take and Skip are mutually recursive function definitions. Take returns the odd-numbered elements of a list. It is defined by appending the head of the list to the result of applying its tail to Skip. Skip returns the even-numbered elements of a list by calling Take on its tail.

□

3.4 Object Name Set Declarations

3.4.1 Syntax

object_set_declaration ::=
  basic_name_list '(' { *set_index* } ')' ':' *type_expression* ';'

set_index ::=
  range
| *enumeration_type_expression*

3.4.2 Semantics

TBD

3.4.3 Examples

Example:
Commentary:

blah blah

□

3.5 Blocks

A *block* is a construct which includes declarations and statements. The following program constructs create blocks:

1. Do statement.
2. Function.
3. Module constructor.

Control enters a block when:

1. Control reaches the do statement.
2. Control is transferred to the function.
3. The module constructor is executed.

The following may cause control to leave a block:

1. Control reaches the end of the block.
2. Return statement.
3. Exit statement.
4. Interrupt.
5. Exception.

Chapter 4

Names and Expressions

Expressions are syntactic constructs used to describe certain kinds of entities in the language. Whenever the language definition requires an entity of a particular kind somewhere, an expression of this same kind may appear.

The simplest kind of expressions are *names*. Names denote program entities without requiring any evaluation.

This chapter describes names, expressions in general, and *object expressions*, which denote objects.

4.1 Names

Syntax

name ::=
  basic_name | *scoped_name*

basic_name ::=
  identifier | *operator_symbol*

scoped_name ::=
  scoped_name '::' *basic_name*

Semantics

A declaration introduces a *basic name*, and associates it with the declared entity.

A name appearing on the left hand side of a `'::'` in a scoped name must denote a declarative region. The name appearing on the right hand side of the `'::'` must be a constituent of that declarative region. See the Guide to the RAPIDE 1.0 Language Reference Manuals for the visibility rules associated with name scopes.

In the cases where an entity is not directly visible because the visibility rules state that a name is hidden, a *scoped name* may be used to refer to the entity.

4.2 Expressions

A *name* is legal syntax for each of the five namespaces of RAPIDE. Additionally, each namespace defines its own syntax for expressions. For each namespace, the following shows where the legal expressions of that namespace are defined:

- Objects: Object expressions are defined in the rest of this chapter.
- Types: Type expressions are defined in Chapter 3.1 of the RAPIDE 1.0 Types Language Reference Manual. Additionally, predefined type expressions are defined in the RAPIDE 1.0 Predefined Types and Objects Reference Manual.
- Patterns: Pattern expressions are defined in Chapter 2 of the RAPIDE 1.0 Event Pattern Language Reference Manual.
- Labels: The only legal label expression is a name.
- Maps: Map expressions are defined in Chapter 5 of the RAPIDE 1.0 Architectures Reference Manual.

4.3 Object Expressions

Object expressions in RAPIDE denote objects (functions or modules).

The rest of this chapter defines the rules applicable to different forms of object expressions, and their evaluation. Since object expressions are very similar to the traditional notion of expressions, the word "object" will be omitted in all further discussion, unless there is the possibility of ambiguity.

4.3.1 Syntax

expression ::= *relations*

```
relations ::=
        relation
|       relations positive_logical_operator relation
|       relations negative_logical_operator relation
|       relations identifier relation
|       relations other_operator relation
```

```
relation ::=
        simple_expression
|       relation relational_operator simple_expression
|       if_expression
|       discrim_expression
```

```
simple_expression ::=
        term
|       unary_adding_operator term
|       simple_expression binary_adding_operator term
|       prefix_operator term

term ::=
        factor
|       term multiplying_operator factor

factor ::=
        primary
|       primary '**' primary
|       not primary
|       abs primary
|       '$' primary
|       new type_expression

primary ::=
        attributed_expression

attributed_expression ::=
        indexed_expression
|       attributed_expression ''' name '''

qualified_expression ::=
        type_expression '''
              '(' expression ')'

indexed_expression ::=
        selected_expression
|       function_call

function_call ::=
        indexed_expression general_aggregate

selected_expression ::=
        name_expression
|       indexed_expression '.'  name_expression

name_expression ::=
        literal
|       name
|       paren
|       exist_placeholder
|       univ_placeholder
```

```
general_aggregate ::=
        '(' [ aggregate_component_list ] ')'
|       '[' [ aggregate_component_list ] ']'
```

```
aggregate_component_list ::=
        aggregate_component_list_element
              { ',' aggregate_component_list_element }
```

```
aggregate_component_list_element ::=
        type_or_object_expression
|       named_association
|       range_expression
```

```
named_association ::=
        identifier is type_or_object_expression
|       default is type_or_object_expression
```

```
paren ::=
        '(' expression ')'
```

```
if_expression ::=
        if expression then expression else expression end if
```

Much of the syntax of expressions is devoted to resolving the precedence and association of infix and prefix operators.

4.3.2 Semantics

An expression denotes an object of a particular type. The type of an expression is the type of the object denoted by the expression. The evaluation of an expression is the process of determining the object which the expression denotes. There are two operations which can be performed during expression evaluation:

1. Selection
2. Function call

All expression evaluation can be reduced to repeated application of these two steps.

4.3.3 Selection

Syntax

```
selected_expression ::=
        name_expression
|       indexed_expression '.'  name_expression
```

Semantics

The left hand side of the "." in a selection must denote an object of an interface type. The right hand side of the "." denotes the corresponding constituent of the interface type.

See the Guide to the RAPIDE 1.0 Language Reference Manuals for the visibility rules associated with selection.

Examples

Example: *Selection*
□

4.3.4 Function Call

Syntax

```
function_call ::=
        indexed_expression general_aggregate


general_aggregate ::=
        '(' [ aggregate_component_list ] ')'
|       '[' [ aggregate_component_list ] ']'
```

Semantics

The semantics of function call is described in Chapter 6.

4.3.5 Operators

Infix and prefix operators are syntactic sugar for function application. The association and precedence of operator symbols are predefined.

Syntax

other_operator ::=
  all combinations of the following operator symbols
    taken in groups of 1, 2, or 3 that are not defined elsewhere:
      ! @ # $ % ^ & * − = + | < > ? / ~

positive_logical_operator ::=
        and | **andthen** | **or** | **orelse** | **xor**

```
negative_logical_operator ::=
        nand | nor


relational_operator ::=
        '=' | '/=' | '<' | '<=' | '>' | '>=' | '/=='
|       '=='


binary_adding_operator ::=
        '+' | '-' | '&' | '~'


unary_adding_operator ::=
        '+' | '-' | '~'


multiplying_operator ::=
        '*' | '/' | mod | rem


prefix_operator ::=
        'not' | 'abs'
```

Several operators are also defined by the syntax rule for *factor*:

```
factor ::=
        primary
|       primary '**' primary
|       not primary
|       abs primary
|       '$' primary
|       new type_expression


operator_symbol ::=
        other_operator
|       '**'
|       not
|       abs
|       '[' ']'
|       positive_logical_operator
|       negative_logical_operator
|       relational_operator
|       binary_adding_operator
|       multiplying_operator
```

Semantics

An infix or prefix operator expression may have one of two meanings in terms of selection and function application:

expression1 operator expression2 ::=
    expression1.operator(expression2)
| *operator(expression1,expression2)*

operator expression ::=
    expression.operator()
| *operator(expression)*

If exactly one of the two meanings is legal in RAPIDE, the expression is legal. If neither meaning is legal, or both are legal, the expression is illegal.

Unary operators are right associative. All other operators are left associative. Figure 4.1 summarizes the precedence of the operator symbols. Each box contains operators of equal precedence. An operator in an earlier box has higher precedence than an operator in a following box.

The "typical meaning" column describes the function typically associated with the operator in the predefined interface types. Note, however, that an interface could associate these operators with any function.

Note that arbitrary identifiers may be used as binary infix operators of the lowest precedence.

4.3.6 Object Literals

Object literals are syntactic constructs that denote objects either of a predefined types, or a type derived from a predefined type constructor.

Syntax

```
literal ::=
|       numeric_literal
|       string_literal
|       character_literal
|       set_literal
|       function_literal
|       module_literal
|       architecture_literal
```

Semantics

Numeric, string, character, and set literals are defined in Chapters 6, 8 , 7, and 15, respectively, of the RAPIDE 1.0 Predefined Types and Objects Reference Manual. Function literals are defined in Chapter 6. Module literals are defined in Chapter 9. Architecture literals are defined in Chapter 3 of the RAPIDE 1.0 Architectures Reference Manual.

Operator	Typical Meaning	Example
NOT	logical negation	not expr
ABS	absolute value	abs expr
\$	dereference	\$ expr
[]	array index	[]
**	exponent	**
*	multiplication	expr * expr
/	division	expr / expr
MOD	modulo	expr mod expr
REM	remainder	expr rem expr
IN	set inclusion	expr in expr
+	unary minus	- expr
-	unary plus	+ expr
+	addition (plus)	expr + expr
-	subtraction (minus)	expr - expr
&	string concatenation	expr & expr
=	equal	expr - expr
/=	not equal	expr /= expr
==	identity	expr == expr
/==	not identity	expr /== expr
<	less than	expr < expr
<=	less than or equal	expr <= expr
>	greater than	expr > expr
>=	greater than or equal	expr >= expr
NAND	logical nand	expr nand expr
NOR	logical nor	expr nor expr
AND	logical and	expr and expr
ANDTHEN	short circuit and	expr andthen expr
OR	logical or	expr or expr
ORELSE	short circuit or	expr orelse expr
XOR	exclusive or	expr xor expr
other operator symbols	none	expr *op* expr

Figure 4.1: Operator Precedence Summary

Examples

Example: *Literals*

```
"This is a string"                      -- a string literal
(a is 10, b is c + d)                   -- a record literal
10                                      -- an integer literal
'c'                                     -- a character literal
{1,2,3}                                 -- a set literal
function(a, b : T) return T2 is         -- a function literal
begin
   -- statements
end
```

□

4.3.7 Attributes

Syntax

```
attributed_expression ::=
        indexed_expression
|       attributed_expression ‘’’ name ‘’’
```

Semantics

Attributes are basic operations on entities provided by the implementation. A summary of the attributes defined by the RAPIDE language appears in Appendix A.

Examples

Example: *Attributes*

declare
F : **function**() **return** Integer **is** ... ;
when F'Call **do**
−− statements
end;

Commentary:

The declaration of a function implicitly declares two actions, which are designated by the 'Call and 'Return attributes.

□

4.3.8 Qualified Expressions

A qualified expression is used to state explicitly the type of an object expression.

Syntax

```
qualified_expression ::=
        type_expression ‘’’
              ‘(’ expression ‘)’
```

Semantics

The object expression must be a subtype of the type expression. The type of the entire expression is the type expression.

When the type of an expression is not known from the context, a qualified expression can be used to state the type explicitly. Explicit qualification may be used to specify the type of an aggregate, to disambiguate overloaded enumeration literals and function calls, or to constrain an object such that only the constituents of the supertype are visible.

Examples

Example: *Qualified enumeration literals*

```
type MASK is enum FIX, DEC, EXP, SIGNIF end;
type CODE is enum FIX, CLA, DEC, TNZ, SUB end;

PRINT : function(v : MASK) is ... ;
PRINT : function(v : CODE) is ... ;

PRINT (MASK'(DEC));   --   DEC is of type MASK
PRINT (CODE'(DEC));   --   DEC is of type CODE
```

Commentary:

The identifier DEC is used as an enumeration literal of types MASK and CODE. By using a qualified expression to explicitly state which usage is intended, overloading can select the proper PRINT function.

□

4.3.9 Self

self is an implicit constituent of the **provides** part of an interface type. It always denotes the object of which it is a constituent.

Thus, **self** always denotes an interface object expression.

4.3.10 If-expression

The if-expression evaluates to one of two nested expressions, depending on the truth value of its condition.

Syntax

```
if_expression ::=
       if expression then expression else expression end if
```

Semantics

The first expression must be of type Boolean. Either the **then** expression must be a subtype of the **else** expression or vice versa. The type of the if-expression is the supertype of the two expressions.

An if-expression is evaluated by evaluating the first expression. If its value is True, the **then** expression is evaluated and returned as the value of the expression (the **else** expression is not evaluated). Otherwise the **else** expression is evaluated and returned as the value of the expression (the **then** expression is not evaluated).

The predefined functions "ANDTHEN" and "ORELSE" are defined in terms of the if-expression as follows:

A ANDTHEN B *is equivalent to*
if A **then** B **else** False **end if**

A ORELSE B *is equivalent to*
if A **then** True **else** B **end if**

Examples

Example: *If-expression*

if Denom /= 0 **then** Num/Denom **else** 0 **end if**

Commentary:

If the value of Denom is anything but zero, the value of this expression is Num/Denom; otherwise the value is zero. In the latter case, Num/Denom is not evaluated.

□

Chapter 5

Statements

This chapter describes the statements in the RAPIDE language. A statement may be one of the following:

labeled_stmt ::=
  [*label*] *stmt*

stmt ::=
  function_call_stmt
| *do_stmt*
| *case_stmt*
| *return_stmt*
| *next_stmt*
| *exit_stmt*
| *if_stmt*
| *null_stmt*
| *await_stmt*
| *assert_stmt*
| *raise_stmt*

label ::=
  identifier ':'

stmt_list ::=
  labeled_stmt { *labeled_stmt* }

5.1 Labels

The occurrence of a labeled statement declares the associated label. The scope of the label extends from the point of declaration of the label to the end of the enclosing declarative region. The scope of the label also includes any declarative region associated with the labeled statement.

Labels may not be overloaded.

5.2 Conditional Statement

5.2.1 Purpose

A conditional statement selects for execution one or none of the enclosed sequences of statements, depending on the value of one or more corresponding Boolean conditions.

5.2.2 Syntax

if_stmt ::=
  if *expression* **then**
    stmt_list
  { **elsif** *expression* **then**
    stmt_list }
  [**else**
    stmt_list]
  end [**if**] ';'

5.2.3 Semantics

The expressions following **if**and **elsif**must be object expressions of type Boolean.

For the execution of a conditional statement, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final "**else**" as "**elsif** True **then**"), until one evaluates to True or all conditions are evaluated and yield False. If one condition evaluates to True, then the corresponding sequence of statements is executed; otherwise none of the sequences of statements is executed.

Example: *If Statement*

```
if ($Month = December) and ($Day = 31) then
        Month := January;
        Day := 1;
        Year := $Year + 1;
end if;

if Line_Too_Short then
        raise Layout_Error();
elsif Line_Full then
        New_Line();
        Put(Item);
else
        Put(Item);
end if;
```

□

5.3 Do Statement

5.3.1 Purpose

A do statement defines a list of statements, possibly with some local declarations, that may be iterated, or executed in response to matching a pattern. The list of statements is known as the body of the do statement, or as the body of the loop for the looping constructs.

5.3.2 Syntax

do_stmt ::=
  [**declare** *declaration_list*]
  [*iteration_scheme* [**declare** *declaration_list*]]
  do
    stmt_list
  [**handler** *handler_list*]
  end *do_terminator* ';'

iteration_scheme ::=
  when *pattern*
| **loop**
| **for** [*identifier* [':' *type_expression*] **in**] *expression_or_range*
| **for** *expression* **in** *expression* **next** *expression*
| **while** *expression*

do_terminator ::=
  [**while** | **for** | **when** | **loop** | **do**] [*identifier*]

5.3.3 Semantics

If an iteration scheme is present, the iteration scheme defines a rule for repeated execution of the body (statement list part) of the do.

If a declaration list appears following the iteration scheme, the declarations are elaborated once before each iteration of the sequence of statements.

The handler portion of a do statement contains the code for exception and interrupt handling. Handlers are described in Chapter 8.

The following sections describe the different iteration schemes.

Loop

For a do statement with a loop iteration scheme, execution of the list of statements repeats until some mechanism such as an exit statement or exception causes the loop to terminate. The other forms of iteration (when, for, and while) can all be expressed using the basic loop. They are syntactic sugar for commonly occurring forms of loops.

When Statement

A do statement with a when iteration scheme is known as a when statement, *pattern triggered statement*, or *rule*. The semantics of pattern matching is described in the RAPIDE 1.0 Event Pattern Language Reference Manual; the semantics of pattern triggered statements is described in Chapter 7.

For Loops

There are two kinds of for iteration schemes:

1. The first form of for loop is iteration across the "contents" of an object, such as elements of a set, elements in an array, objects in a bag, etc. Objects that support this form of iteration are known as container objects. The general approach to this form of iteration is that each container object exports a module generator for an iteration object to use in iterating over the contents of the container.

   This type of for loop has the following most general form:

```
for <identifier> : <type_expression> in <expression_or_range> do
    <stmt_list>
end;
```

   The expression must be either a legal range or an object expression of type Iterator(T) for some type T. See Chapter 12 of the RAPIDE 1.0 Predefined Types and Objects Reference Manual for a definition of Range and Iterator. Further, T must be a subtype of the given type_expression.

   If the type_expression is not given, it is inferred from the iterator expression. If the identifier expression is not given, the semantics of the loop is the same as if a unique identifier were used.

   The expression is evaluated once to determine the iterator object when control reaches the for loop. The following steps are then repeated:

   (a) The More function of the iterator object is called. If the result is False, the loop is complete.
   (b) Otherwise, the Item function of the iterator object is called.
   (c) The body of the loop is executed with the identifier denoting the object returned by the call to the More function.

   This form of for loop is equivalent to the following:

```
declare
    It : Iterator(<type_expression>) is <expression>;
do
    For_Loop: loop do
        exit For_Loop where not It.More();
        declare
            <identifier> : <type_expression> is It.Item();
        do
            <stmt_list>
        end;
    end;
end;
```

The name It and the label For_Loop are simply chosen for purposes of exposition and have no special significance.

2. The second form of for loop is a general form having three expressions: an initializer, a test, and an increment. Its form is:

```
for <expression1> in <expression2> next <expression3> do
    <stmt_list>
end;
```

All three expressions must be object expressions; the type of the second expression must be Boolean.

The first expression is executed once when control reaches the for loop. The following steps are then repeated:

   (a) The second expression is executed, and if it evaluates to False, execution of the loop is complete.
   (b) Otherwise, The body of the loop is executed.
   (c) The third expression is executed.

This type of for loop is equivalent to:

```
<expression1>;
For_Loop: loop do
    exit For_Loop when not <expression2>;
    <stmt_list>
    <expression3>;
end;
```

While Loops

For a do statement with a while iteration scheme the expression must be an object expression of type Boolean.

The expression is evaluated before each execution of the sequence of statements; if the value of the expression is True, the sequence of statements is executed, and execution continues with the next evaluation of the expression. If the expression evaluates to False the execution of the do statement is complete.

5.3.4 Examples

For examples of pattern triggered do statements, see Chapter 7.

Example: *Do statement with exit*

```
declare
    C : Ref(Character) is Ref_To(Character, ' ');
do
    C := Read_Character();
    exit where $C = '*';
end do;
```

Commentary:

C is a reference to a character. The declaration of C is elaborated only once, regardless of how many iterations of the loop actually take place. The scope of C is the body of the do statement. Execution of the do statement completes when the exit statement forces completion of the loop.

□

Example: *Iterating from 1 to 10*

```
for J : Integer in 1 .. 10 do
    if Buffer(J) /= Space then
        Put(Buffer(J));
    end if;
end do;
```

Commentary:

The value of J in the loop is initially one; it is incremented each time through the loop and exits after the tenth iteration.

□

Example: *Do statement with a label*

```
Summation : loop do
    Sum   := $Sum + Nxt.Value;
    while Nxt /= Head do
        Nxt := Nxt.Succ;
        exit Summation where Is_Nil(Nxt);
    end;
end do Summation;
```

Commentary:

The Summation do statement completes when the exit statement is executed and the condition in the exit evaluates to True.

□

5.4 Exit Statement

5.4.1 Purpose

An exit statement is used to complete the execution of an enclosing do statement (called the do in what follows); the completion is conditional if the exit statement includes a condition.

5.4.2 Syntax

exit_stmt ::=
  exit [*do_name*] [**where** *condition*] ';'

do_name ::=
  identifier

5.4.3 Semantics

An exit statement with a do name is only allowed within the do statement with that label, and applies to that do; an exit statement without a do name is only allowed within a do, and applies to the innermost enclosing do (whether labeled or not).

For the execution of an exit statement, the condition, if present, is first evaluated. Exit from the do then takes place if the value is True or if there is no condition.

5.4.4 Examples

Example: *Exit from loop*

```
loop do
    Get_New_Item(New_Item);
    Merge_Item(New_Item, Storage_File);
    exit where New_Item = Terminal_Item;
end do;
```

Commentary:

The loop is exited when NEW_ITEM = TERMINAL_ITEM evaluates to True.

□

Example: *Exit from nested loop*

```
type Event_Label is enum Mousebutton, Ctl_Alt_Del end enum;
...
Main_Cycle: loop do
    Enable_Input_Events();
    Inner_Cycle: when (?V in Event_Label) Input_Event(?V) do
        Process_Input_Event(?V);
        exit Main_Cycle where ?V = Ctl_Alt_Del;
    end Inner_Cycle;
end Main_Cycle;
```

Commentary:

Execution of the exit statement within loop inner_cycle where the condition evaluates to True will cause both loops to be exited.

□

5.5 Next Statement

5.5.1 Purpose

A next statement is used to complete the execution of the current iteration of an enclosing do statement (called the do in what follows), and begin the next iteration of the do. The completion is conditional if the next statement includes a condition.

5.5.2 Syntax

next_stmt ::=
  next [*do_name*] [**where** *condition*] ';'

do_name ::=
  identifier

5.5.3 Semantics

A next statement with a do name is only allowed within the named do, and applies to that do; a next statement without a do name is only allowed within a do, and applies to the innermost enclosing do (whether named or not).

For the execution of a next statement, the condition, if present, is first evaluated. The current iteration of the do then ends if the value is True or if there is no condition, and the next iteration begins. If there is no subsequent iteration, the do is completed.

5.5.4 Examples

Example:

```
Type Event_Label is enum Mousebutton, Ctl_Alt_Del, Ctl_C end enum;
...
Main_Cycle: loop do
      Enable_Input_Events();
      Inner_Cycle: when (?V in Event_Label) Input_Event(?V) do
          Process_Input_Event(?V);
          exit Main_Cycle where ?V = Ctl_Alt_Del;
          next Main_Cycle where ?V = Ctl_C;
      end Inner_Cycle;
end Main_Cycle;
```

Commentary:

Execution of the next statement where the condition is True will cause execution to continue at the top of the loop main_cycle.

□

5.6 Case Statement

5.6.1 Purpose

A case statement selects for execution one or more of a number of alternative sequences of statements based on an expression. The selection can be based on the type of the expression, or on the object denoted by the expression. By selecting on a type expression, the case statement used to treat an object of a type as an object of a subtype of the type, commonly known as downcasting.

5.6.2 Syntax

case_stmt ::=
  case *case_expression* **of**
      case_body
      [*case_default_part*]
  end [**case**] ';'

case_expression ::=
  [*identifier* ':' *type_expression* **is**] *expression*

case_body ::=
  case_xor_list
| *case_or_list*
| *case_else_list*

case_xor_list ::=
  case_stmt_alternate { **xor** *case_stmt_alternate* }

case_or_list ::=
  case_stmt_alternate { **or** *case_stmt_alternate* }

case_else_list ::=
  case_stmt_alternate { **else** *case_stmt_alternate* }

case_stmt_alternate ::=
  case_choice_list '=>' *stmt_list*

case_choice_list ::=
  case_choice { ',' *case_choice* }

case_choice ::=
  expression
| *type_expression*
| *expression* '..' *expression*

case_default_part ::=
  default '=>' *stmt_list*

5.6.3 Semantics

The execution of a case statement consists of the evaluation of the case expression followed by the execution of the chosen sequence(s) of statements. The case expression is evaluated once when control reaches the case statement.

A case statement has an associated type called the *case type*. If the form of the case expression including the identifier and type expression is used, the type expression is the case type; otherwise, the type of the given expression is the case type.

The type of the identifier, if it exists, is the case type. Also, the expression and all case choices denoting objects in the case statement must be of the case type. In addition, for all case choices with a type expression, the type expression in the case choice must be a subtype of the case type.

Within the scope of the case statement the identifier denotes the same object as the expression.

Case Choices. Each case choice consists of either a list of objects (denoted by expressions) or a single type expression. If a type expression appears in a case choice, it must be the only item in the case choice.

If the case choice denotes a list of objects, one of the "=" or "==" functions on each object is called with the object denoted by the case expression as its argument. The "==" function (predefined object identity, see Chapter 22 of the RAPIDE 1.0 Predefined Types and Objects Reference Manual) is used only if the object does not define a binary "=" function. If the result of the function is True for at least one of the objects in the case choice list the case alternative is eligible for execution.

If a case choice is of the form "expression .. expression", the expressions must both denote discrete objects; that is, their they must have a common supertype T such that `T <: Discrete(T)`. See Chapter 12 of the RAPIDE 1.0 Predefined Types and Objects Reference Manual for a definition of Discrete. [2] **Or possibly add Range type to the Predef LRM.**

The case choice is the list of objects defined by starting with the first expression, and successively applying the Succ operator until the resulting object is equal (the "=" operator returns True) to the second expression. If the first expression is not less than or equal to the second expression, the list is empty.

If the case choice is a type expression, the alternative is eligible for execution if the object denoted by the case expression has a type equivalent to the type denoted by the type expression. If an identifier was given as part of the case expression, within an alternative selected by a type expression case choice, the type of the identifier is the type of the type expression in the case choice. That is, inside the alternative the type of the identifier changes from the case type to the subtype specified by the alternative.

Case alternatives. If case alternatives are separated by "xor", only one list of statements may be eligible for execution. If more than one alternative is eligible for execution an exception [3] **Which excpetion?** is raised.

If alternatives are separated by "or", multiple alternatives may be eligible for execution. Each alternative is executed in the order in which it appears in the case statement.

If alternatives are separated by "else", multiple alternatives may be eligible for execution. Only the first eligible alternative is executed.

If no alternatives are eligible for execution, and there exists a case default part, the statements in the case default part are executed.

5.6.4 Examples

Example:

```
type Sensor is interface ... end;
type Elevation is interface include Sensor; ... end;
type Azimuth is interface include Sensor; ... end;
type Distance is interface include Sensor; ... end;
Get_A_Sensor : function() return Sensor;

case S : Sensor is Get_A_Sensor() of
        Elevation      => Record_Elevation(S.Value);
    xor Azimuth        => Record_Azimuth(S.Value);
    xor Distance       => Record_Distance(S.Value);
    default            => null;
end case;
```

Commentary:

In this example, Sensor is a general interface type describing sensors. There are three subtypes of Sensor representing particular kinds of sensors: Elevation, Azimuth, and distance. The case expression calls Get_A_Sensor() to return a sensor. Within the case statement, S denotes this sensor. S may in fact also be an object of a more specific (subtype of) Sensor. Each case alternative names a subtype of Sensor. If S is of type elevation, then the first case alternative will be eligible for execution, and within that alternative S has type Elevation.

□

Example:

```
case Today of
        Mon          => Compute_Initial_Balance();
    xor Fri          => Compute_Closing_Balance();
    xor Tue .. Thu   => Generate_Report(Today);
    xor Sat .. Sun   => null;
end case;
```

□

Example:

```
case Bin_Number(Count) of
        1            => Update_Bin(1);
    xor 2            => Update_Bin(2);
    xor 3,4          => Empty_Bin(1);
                        Empty_Bin(2);
    default          => Error;
end case;
```

□

Example:

```
case P : Point is Next_Point() of
```

```
            Point          => Do_Point_Things();
         or Colored_Point  => Do_Colored_Point_Things();
     end case;
```

Commentary:

> In this example, Colored_Point is a subtype of Point. If P is a point, only the first case alternative is executed. (Note that P must be of type Point, since if Next_Point does not return a Point, a static semantic error would result.) If P is also a Colored_Point, the second case alternative is also executed. within the second case alternative is is of type Colored_Point.

□

5.7 Await Statement

The await statement is described in Section 7.3.1.

5.8 Return Statement

5.8.1 Purpose

A return statement is used to complete the execution of the innermost enclosing function.

5.8.2 Syntax

return_stmt ::=
  return [*expression*] ';'

5.8.3 Semantics

A return statement is only allowed within the body of a function, and applies to the innermost (enclosing) such construct.

A return statement for the body of a function may omit an expression if the return type in the function declaration is also omitted (in which case a value of the Root type is returned); otherwise the return statement must include an expression. The value of the expression defines the result returned by the function, and must be a subtype of the return type of the function.

For the execution of a return statement, the expression (if any) is evaluated. The execution of the function is completed.

5.8.4 Examples

Example:

```
return Key_Value(Last_Index);
return;                               -- in a function with no return type
```

□

5.8.5 Discussion

5.9 Assert Statement

5.9.1 Purpose

An assertion statement indicates an invariant that should be True at the point in the computation where the assertion is executed.

5.9.2 Syntax

assert_stmt ::=
  assert *expression* ';'

5.9.3 Semantics

If the expression evaluates to False, an Inconsistent event is generated. If the assertion has a label, then the label will be a parameter of the inconsistent event.

5.9.4 Examples

Example:

```
assert False;                    -- always generate an inconsistent event
assert Account_Balance > 0;      -- frequently generates inconsistent events
```

□

5.9.5 Discussion

5.10 Null Statement

5.10.1 Purpose

The null statement is a statement which does nothing.

5.10.2 Syntax

null_stmt ::=
  null

5.10.3 Semantics

Execution of a null statement has no effect;

5.10.4 Examples

Example: *Null statement*

```
null;
```

□

5.11 Function Call Statement

5.11.1 Purpose

A function call statement invokes the execution of the corresponding function body.

5.11.2 Syntax

function_call_stmt ::= *function_call* ';'
function_call ::= *indexed_expression* *general_aggregate*

5.11.3 Semantics

The call specifies the association of the actual parameters, if any, with formal parameters of the function. See Chapter 6 for a description of function parameter modes and association.

The execution of a function call proceeds as follows:

1. The expression is evaluated to determine the function object to be called.
2. The actual parameters are evaluated and bound to the formal parameters.
3. The function is executed with the associated formal to actual bindings.

The order in which the parameter associations of a function call are evaluated is not defined by the language.

5.11.4 Examples

Example:

```
Traverse_Tree();
Table_Manager.Insert(E);
Print_Header(128, Title, True);
Switch(From_Param is X, To_Param is Nxt);
```

```
Print_Header(128, Header is Title, Center is True);
Print_Header(Header is Title, Center is True,
             Pages is 128);
```

□

5.12 Raise Statement

5.12.1 Purpose

A raise statement is used to generate an exception. See Section 8.3 for details.

5.12.2 Syntax

raise_stmt ::=
  raise [*expression*] [**where** *expression*] ';'

Chapter 6

Functions

There are two kinds of objects in RAPIDE: modules, which implement interface types; and functions, which implement function types. This chapter describes how function objects are defined and called.

Functions are one of the forms of program unit of which programs can be composed. It is a named part of a program that can be invoked from other parts of the program.

A function is invoked by a *function call*. A function call may be either a statement or an expression.

6.1 Function Type Expressions

Function type expressions are defined in Chapter 5 of the RAPIDE 1.0 Types Language Reference Manual. A function type expression denotes a type whose objects are functions.

6.2 Function Literal

6.2.1 Purpose

A function literal denotes an object of a function type.

6.2.2 Syntax

function_literal ::=
  function '(' [*formal_parameter_list*] ')' [**return** *type_expression*] **is**
    declaration_list
  begin
    stmt_list
  end [**function**]

formal_parameter_list ::=
  parameter { ';' *parameter* }

formal_parameter ::=
  type_parameter
| *object_parameter*

type_parameter ::=
  type *identifier_list* ['<:' *type_expression*]

object_parameter ::=
  identifier_list ':' *type_expression* [*default_expression*]

default_expression ::=
  is *expression*

6.2.3 Semantics

The result of evaluating a function literal is a function object.

The default_expression of an object parameter denotes an object to be associated with the formal parameter in the case that no actual parameter appears in a corresponding function call. See Section 6.4 for details on function call and parameter association.

The type of a function literal is determined by constructing a function type expression with the same formal parameter list and return type as the function literal.

The semantics of formal parameters are defined in Section 5.2.2 of the RAPIDE 1.0 Types Language Reference Manual.

6.2.4 Examples

Example: *References to functions*

```
declare
   F      : Ref(function(X, Y : Integer) return Integer);
   A, B   : Ref(Integer);

do
   F :=  function(X, Y : Integer) return Integer is
         begin
            return X + Y;
         end;
   A :=  $F(2,10);
   F :=  function(X, Y : Integer) return Integer is
         begin
            return X - Y;
         end;
   B :=  $F(2,10);
end;
```

Commentary:

The name F is declared as a reference to a function. Initially it refers to nothing. F is used by assigning a function object to it, and then calling it. In this example two different function objects denoted using function literals are assigned to F.

□

Example: *In–line function literal*

```
declare
   X : var Integer := 10;
do
   X := (function(X,Y : Integer) return Integer is
           begin
              return X*Y;
           end)(2,4);
end;
```

Commentary:

The function literal is evaluated, resulting in a function object. The function object is called (see Section 6.4) with actual parameters 2 and 4.. The result is assigned to X.

The type of the function literal is
function(X,Y : Integer) **return** Integer

□

6.3 Function Declaration

6.3.1 Purpose

A function declaration associates a basic name with an object of a function type.

6.3.2 Syntax

function_declaration ::=
  basic_name_list ':' *type_expression* **is** *function_literal* ';'
| **function** *basic_name* '(' [*formal_parameter_list*] ')' [**return** *type_expression*] **is**
    declaration_list
  begin
    stmt_list
  end [**function**] [*identifier*] ';'

6.3.3 Semantics

A function declaration associates a basic name (identifier or operator symbol) with an object of a function type.

The first form of the syntax for a function declaration is the general form, and is the same as for other object declarations.

The second form of the syntax is less verbose in the case of a declaration of a single function constant, and more similar to the style used in Ada and other Algol–like languages. It is semantically equivalent to a declaration of an object of an appropriate function type with the same basic name and an appropriate function literal. The function type is constructed using the formal parameter list and return type of the declaration. The value is determined by constructing a function literal with the same formal parameter list, return type, declaration list, and statement list as in the declaration.

6.4 Function Call

6.4.1 Purpose

A function call occurs as part of an expression or as a statement.

6.4.2 Syntax

function_call ::=
  expression '(' [*parameter_association_list*] ')'

function_call_stmt ::==
  function_call ';'

parameter_association_list ::=
  parameter_association { ',' *parameter_association* }

parameter_association ::=
  [*formal_parameter* **is**] *actual_parameter*

formal_parameter ::=
  identifier

actual_parameter ::=
  expression

If a function call occurs as a statement, it must be terminated by a ';' as are all other statements.

6.4.3 Semantics

A function call proceeds as follows:

1. The actual object parameters are evaluated and associated with formal object parameters as described later in this section.
2. A call event is generated as described later in this section.
3. The declarations in the declaration part are elaborated.
4. The statement list is executed.

Execution of the function terminates when a return statement is executed (in which case the value of the return statement is returned by the function), or when control reaches the end of the list of statements (in which case an arbitrary value of the return type is returned). See Section 5.8 for the definition of return statements.

A return event as described below is generated immediately after execution of the function terminates, but before any return value is made available.

Functions may declare formal type parameters, allowing polymorphism. The meaning of polymorphic functions is given below.

Parameter Association

A parameter association is said to be named if the formal parameter is named explicitly; it is otherwise said to be positional. For a positional association, the actual corresponds to the formal parameter with the same position in the formal part.

Named associations can be given in any order, but if both positional and named associations are used in the same call, positional associations must occur first, at their normal position. Hence once a named association is used, the rest of the call must use only named associations.

For each formal parameter of a function, a function call must specify exactly one corresponding type or object. This type or object may be specified explicitly by a parameter association. For object parameters in the absence of such an association, the object may be specified implicitly by a default expression.

For object parameters, the type of an object denoted by the actual parameter must be a subtype of the type of the corresponding formal object parameter. In the case where the type denotes a formal type parameter, the type of the object must be a subtype of the corresponding actual type parameter.

For type parameters, the type denoted by the actual parameter must satisfy any subtype constraint on the formal type parameter; if no subtype constraint is given, any type may be associated with the formal parameter.

Meaning of Parameters in Functions

The call specifies the association of an object (denoted by the actual parameter expression) with each formal object parameter (if any) of the function. The corresponding object expressions are evaluated at the time of the call. Within the body of the function, the formal then denotes that object.

The order in which the actual object parameters of a function call are evaluated is not defined by the language.

Occurrences of formal type parameters in the body refer to the type passed as the associated actual parameter. In particular, if an object is declared whose type denotes a type parameter, an object of the actual type is created.

Call and Return Event Generation

The event generated as a result of a function call has one parameter for each formal parameter (including type parameters) of the function. The order, type, and name of these parameters are the same as their function counterparts. The call events associated with a function are matched in a basic pattern using the 'Call attribute of the function.

The event generated as a result of a function return has one parameter for each formal parameter of the function. The order, type, and name of these parameters are the same as their function counterparts. The return event includes an additional parameter of the type of the return value. The additional parameter is last and is named Return. The return events associated with a function are matched in a basic pattern using the 'Return attribute of the function.

The call and return events generated by a function call are visible as if an out- or internal-action call had been made at the point of the function call.

6.4.4 Examples

Example: *Return parameters in implicit events*

```
function Sin (X : Float) return Float;
-- A call to Sin produces two events
--     Sin'Call    (X : Float);
--     Sin'Return  (X : Float; Return : Float);

function Push (type T; Item : T);
-- A call to Push produces two events
--     Push'Call    (type T; Item : T);
--     Push'Return  (type T; Item : T; Return : Root);
```

Commentary:

The Sin'Return event has an a parameter named Return of the return type.

Push is polymorphic, and so are its 'Call and 'Return events.

□

Example: *Aliasing of parameters*

```
I : Ref(T) is Ref_To(T, Make_New_T);
P : function(Arg1, Arg2 : Ref(T)) is ...;
...
P(I, I);
```

Commentary:

Within function P, Arg1 and Arg2 are references to the same object, and the function may assign to them.

□

Example: *Constant parameters*

```
I : Integer is 10;
P : function(Arg1 : Integer; Arg2 : Ref(Integer)) is ...;
...
P(Arg1 is I, Arg2 is I);                     -- illegal
P(Arg1 is I, Arg2 is Ref_To(Integer, I));    -- legal
```

Commentary:

Within the call to function P, Arg1 will be treated as an integer constant denoting the same object (10) as I. Any attempt to assign to Arg1 would result in a type error because assignment is not an operation of integer. The association of Arg2 with I in the first call is a type error because I is of type integer and Arg2 is of type Ref(Integer).

□

Example: *Read–only parameters*

```
type Stack(type Element_Type) is interface
   Create  : function (...);
   Top     : function () return Element_Type;
   Push    : function (...);
   Pop     : function () return Element_Type;
   ...
end stack;

type rstack(type element_type) is interface
   include stack(element_type) only (top);
end rstack;

P : function(Arg1 : Rstack(Integer)) is
    begin
        ...
        Do_Something(Arg1.Top());
        ...
    end;
```

Commentary:

The way to assure that a function does not modify a parameter is to write the function declaration so that the function does not have visibility over operations of the type that may modify the passed object. The earlier examples where the type of the parameter is Integer instead of Ref(Integer) are a special case of this in which the assignment operation is excluded. In this example, the function requires only that its argument support the Top operation. So we create a supertype of Stack that includes only the Top operation. This guarantees that the Push and Pop operations of a Stack passed to function P cannot be called by P, and therefore that P cannot modify the state of the passed Stack.

□

Example: *Write–only parameters*

```
type Write_Only_Integer is interface
    include Ref(Integer) except ("$");
end stack;

P : function(Arg1 : Write_Only_Integer);
```

Commentary:

By creating a supertype of Ref(Integer) that excludes "$", we specify a parameter that supports the assignment operation, but does not allow access to the referenced object through dereferencing.

□

Example: *Recursive function calls*

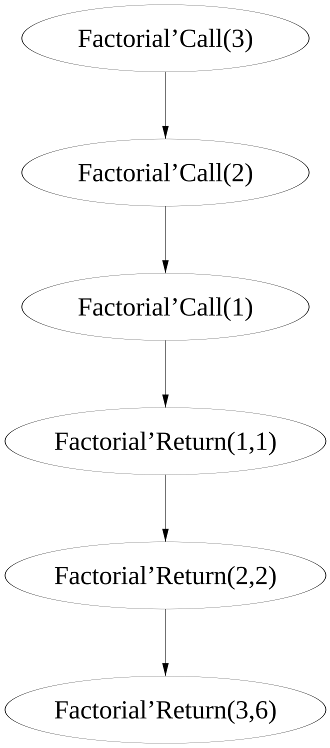


Figure 6.1: Events generated by a recursive function.

```
function Factorial (N : Integer) return Integer is
begin
   if N = 1 or N = 0 then
      return 1;
   else
      return N * Factorial(N - 1);
   end if;
end Factorial;
...
X := Factorial(3);
```

Commentary:

The execution of this call will produce the total order of events shown in Figure 6.1.

□

Chapter 7

Processes: Generating and Reacting to Events

The code comprising a module may include concurrent *processes* (each of which is a separate thread of control).

Processes are constructed from sequential statements, including *action call statements*, which generate events, and *reactive statements*, which react to events. The semantics of statements is primarily covered in Chapter 5. Action calls and reactive statements are defined in this chapter.

Section 7.1 gives the syntax and semantics for a process, and describes how a process affects the order of events it generates. Section 7.2 defines action call statements. Section 7.3 defines the reactive statements. Section 7.4 shows the relationships between several processes in a module. Section 7.5 defines the iterative and conditional generation of processes. Finally, Section 7.6 defines serial modules, in which all processes are synchronized so that only one is executing at a time.

7.1 Processes

A Process is an independently executing thread of control.

7.1.1 Syntax

The syntax of processes is

process ::=
  statement_list

statement_list ::=
  statement {*statement* }

7.1.2 Semantics

A process is a list of sequential statements. It is elaborated by the module that contains it, by executing the list of statements. It terminates when its list of statements terminates execution.

Each process is a single thread of control. A process generates an event by executing an action call; each event it generates is dependent on the preceding event generated by the process. Thus all events generated by a process form a linear dependency ordering.

Processes react to events by using one of the reactive statements described in Section 7.3.

Example: *A simple process*

```
for I : Integer in 1..5 do
    Accounts(I).Initialize();
end do;
```

Commentary:

> This process consists of a single **for** statement. It executes when the module containing it is elaborated. It initializes several accounts (using an action call), and then terminates.

□

Example: *A reactive process*

```
when (?I in Integer) Call(?I) do
    Answer(?I);
end do;
```

Commentary:

> This is a process consisting of a **when** statement (see Section 7.3.2). In brief, the above process repeatedly waits for a Call event, and generates an Answer event in response to it.

□

7.2 Action Call Statements

This section presents the syntax and semantics of calls to actions. Action declarations are presented in Chapter 2.4 of the RAPIDE 1.0 Architectures Reference Manual.

7.2.1 Purpose

The purpose of an action call is to generate an event.

7.2.2 Syntax

action_call ::= [*expression* '.'] *identifier*
'(' *parameter_association_list* ')' [*timing_clause*] ';'

An action call is syntactically similar to a function call. An action call always contains an action name, which may be qualified by selecting from a module expression, and is applied to a (possibly empty) list of parameters. An action call may be modified by a timing clause. Action calls are statements, not expressions, since they do not yield an object.

7.2.3 Semantics

The expression at the beginning of an action call, if any, must denote a module. If no expression is given, then **self** (see Section 9.7) is implicitly named. The identifier must be that of an action of the named module. The rules for parameter association and overload resolution are the same as those for function calls (see Section 5.11).

An action call statement is executed by the module that most immediately encloses it.

The execution of an action call without a timing clause evaluates the name and any parameters, in some order. Then the call generates a module of the predefined type constructor Event(), called an event. The event generated has the type of the event inferred by the action declaration. The event is generated by the process that executes the action call, except in the case of connections (see the RAPIDE 1.0 Architectures Reference Manual).

An event is said to be *of the* called action. We sometimes simply say that calling action A generates an A event.

An event is called an **in**-event if it is of an **in**-action. Likewise **out**- and **internal**-events are of **out**- and **internal**-actions, respectively.

7.3 Reactive Statements

Reactive statements are used to respond to the events available to a process. The primary reactive statement is the *await statement*; the *when statement* is a commonly used do statement that is based on the semantics of the await statement.

7.3.1 The Await Statement

Syntax

The syntax of the await statement is

await_stmt ::=
  await
    pattern ['=>' *stmt_list*
    [*pattern_choices*]
    [*else_part*]
  end [**await**] [*identifier*]] ';'

pattern_choices ::=
  pattern_choice { **or** *pattern_choice* }

pattern_choice ::=
  pattern '=>' *stmt_list*

else_part ::=
  else *stmt_list*

Semantics

An await statement includes one or more patterns that may be matched by events; when events are *available* (see below) that form a match of a pattern, a (possibly null) body of statements associated with the pattern, called its *alternative*, is executed. We can think of the match of a pattern as *triggering* the execution of its alternative.

The execution of an await statement with no else part

await *pattern*Δ0 => *statement_list*Δ0
or *pattern*Δ1 => *statement_list*Δ1
or *pattern*Δ2 => *statement_list*Δ2
...
or *pattern*Δ*n* => *statement_list*Δ*n*
end await

waits for a *triggering match* (defined below) of one or more of the patterns $pattern_0$, ..., $pattern_n$. When available events form a match for pattern $pattern_i$, the corresponding alternative $statement\_list_i$ is executed. If more than one pattern of an await statement is matched by available events, an arbitrary choice is made among the triggering matches. If there is no such match, the process containing the statement blocks until a match is available.

The execution of an await statement with an else part

await *pattern*Δ0 => *statement_list*Δ0
or *pattern*Δ1 => *statement_list*Δ1
or *pattern*Δ2 => *statement_list*Δ2
...
or *pattern*Δ*n* => *statement_list*Δ*n*
else *statement_list*
end await

looks for a triggering match of one or more of the patterns $pattern_0$, ..., $pattern_n$. If available events form a match for pattern $pattern_i$, the corresponding alternative $statement\_list_i$ is executed. If more than one pattern of an await statement is matched by available events, an arbitrary choice is made among the triggering matches. If there is no match for any of the patterns, the statement_list following the **else** is executed and the statement completes.

An await statement with an else part is called a *non-blocking reactive statement*. All other reactive statements are called *blocking reactive statements*.

Once a triggering match is available and the alternative corresponding to it has been chosen, the events of the match are made unavailable to that process.

Triggering Matches. Not every match as defined in the RAPIDE 1.0 Event Pattern Language Reference Manual will trigger the execution of an await statement alternative; for example, once an event has participated in a match that triggers the execution of an alternative, that event will not participate in any subsequent match in the process. The matches that may trigger await alternatives are called the *triggering matches* of the process. The triggering matches of a particular process are defined as follows.

At any point during the execution of a process, there is a set of events that may participate in a triggering match. This set consists of every event visible to the module that: (1) has been generated; (2) if it was generated with a timing clause, the appropriate clock has reached the end time of the event; and (3) has not participated in a previous match in the process that caused the execution of an alternative. The set comprises the *available* events of the process.

Note that since availability is on a per process basis, an event that is unavailable to one process may still participate in the match of a pattern of another process that has not already used it.

In the matching of any particular pattern at a particular point in the execution of the process, there may be more than one set of available events that match the given pattern. Briefly, a triggering match must be an *earliest maximal* match for the pattern among available events. This is determined as follows.

The set of matches of a pattern among available events may be ordered by an *earlier* ordering ("<<"), defined as:

For two computations C_1 and C_2 that match the pattern and use available events,

$$
\begin{array}{ll}
C_1 << C_2 \quad \equiv & \\
& \mathsf{if}\ C_1 = \{\} \quad \vee \quad C_2 = \{\}\ \mathsf{then\ false} \\
& \mathsf{elsif}\ (\exists S_1 \subseteq C_1) S_1 \mid\mid C_2\ \mathsf{then}\ C_1 - S_1 << C_2 \\
& \mathsf{elsif}\ (\exists S_2 \subseteq C_2) S_2 \mid\mid C_1\ \mathsf{then}\ C_1 << C_2 - S_2 \\
& \mathsf{elsif}\ (\exists S) S \subseteq C_1 \wedge S \subseteq C_2\ \mathsf{then}\ C_1 - S << C_2 - S \\
& \mathsf{elsif}\ (\forall e_2 \in C_2)(\exists e_1 \in C_1) e_1 < e_2\ \mathsf{then\ true} \\
& \mathsf{else\ false}
\end{array}
$$

Among the minima of the ordering determined by "<<", the maximal matches (*i.e.* those that are not strict subsets of any other match in the ordering) form the set of earliest maximal matches.

Two properties of an earliest maximal match C are that, with respect to all other matches of the pattern C' that use available events:

- there does not exist an event e' in C' such that for all events e in C e' precedes e in the dependency ordering, and
- one cannot replace an event $e \in C$ with an event $e' \in C'$ such that e' precedes e in the dependency ordering and have the new C still match

Dependency Order of Generated Events. Two rules define how an event generated by a process is related in the dependency order to other events: (1) the event follows the event generated previously by the process, and (2) the event follows all events comprising the most recent triggering match.

Scope of Existential Placeholders. If the outermost operator of an await statement pattern is a placeholder pattern, the scope of declared existential placeholders extends to the associated alternative; the alternative may contain occurrences of these placeholders. A substitution value that has been obtained by a triggering match of the pattern is used as the value of the placeholder in the alternative. No other placeholders are defined in an await alternative.

Since the body must be executed exactly once per triggering match, this means that all existential placeholders occurring in an executed alternative must have exactly one substitution value in each match for this pattern.

We can think of an await statement alternative as a parametrized list of statements, such that upon observing a triggering match, an instance of the alternative is executed, where the placeholder occurrences have been replaced by the substitution values determined in matching.

Discussion

The statement

```
await    pattern;
```

is semantically equivalent to the statement

```
await pattern =>
    null;
end await;
```

An await statement may itself contain nested await statements in its alternatives or else part.

Examples

Example: *A "teller" process*

```
loop do
    await (?D in Dollar, ?A in Accnt) Deposit(?D,?A) =>
        -- Handle deposits
    or    (?D in Dollar, ?A in Accnt) Withdraw(?D,?A) =>
        -- Handle withdrawals
    or    (?D in Dollar, ?A in Accnt, ?N in Check_No)
          Issue_Check(?D,?A,?N) =>
        -- Handle checks
    end await;
end loop;
```

Commentary:

This single thread of control handles all kinds of customer events, responding to each request with the appropriate action statements (omitted here for brevity). In the first alternative, which handles deposits, occurrences of ?D and ?A will have the values determined by the pattern match that caused its execution. However, an occurrence of ?N, for example, in this alternative would be illegal since no definition of that placeholder is visible.

□

Example: *An await statement with an else part*

```
await Issue_Check =>
        -- Make a report
else
        -- Default execution
end await;
```

Commentary:

The else part provides a non-blocking await. In this statement, if an Issue_Check event is available, that alternative will match and be executed. If no Issue_Check events have arrived, the statements of the else part will be executed.

□

7.3.2 When Statements

When statements are a form of the do statement (see Section 5.3). They model "rules" that fire repeatedly upon observation of a triggering match.

Syntax

A when statement is a syntactic form of the do statement:

```
when_statement ::=
  [ declare declaration_list ]
  when pattern [ declare declaration_list ]
  do
    stmt_list
  [ handler handler_list ]
  end when ';'
```

Semantics

A when statement is a reactive compound statement. The pattern of a when statement is called its *trigger*; the statement list of a when statement is called its *body*.

The when statement

```
    declare
        <decls1>
    when <pattern>
    declare
        <decls2>
    do
        <statements>
    handler
        <handlers>
    end when;
```

is equivalent to:

```
    declare
        <decls1>
    loop do
        await <pattern> =>
            declare
                <decls2>
            do
                <statements>
            handler
                <handlers>
            end do;
        end await;
    end loop;
```

A when statement waits for a match of its trigger and then executes its body. This cycle loops repeatedly; an **exit** statement may be used to exit the when statement.

Occurrences of placeholders in the body of a when statement have the substitution values determined by the match of the trigger.

Discussion

A when statement waits for something — a particular pattern of events — to happen, and when it occurs the when statement reacts with the execution of a list of statements. Upon termination of the execution of the list of statements the when statement again waits for a particular pattern of events to occur.

When statements may also be considered as *rules* that fire (execute their bodies) when triggered.

Examples

Example:

```
when (?D in Dollar, ?A in Accnt) Deposit(?D,?A) do
    -- Credit the receiving account
    Accounts[?A] := $(Accounts[?A]) + ?D;
end when;
```

Commentary:

This "teller" when statement is triggered by a Deposit event, and responds to it by increasing the appropriate account by the deposited amount.

If the computation of Figure 7.1 were the events available to the process, the statement would fire twice, once for each Deposit event of the computation. An execution of the body uses the substitution values for ?D and ?A obtained from the match. As a result of the two body executions, the value of Accounts[123] would be 1000 greater than its original value, and the value of Accounts[124] would be 235 greater.

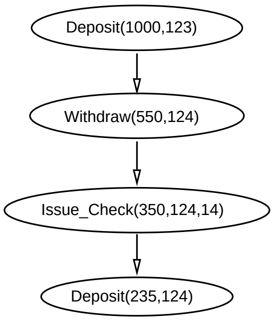


Figure 7.1: A computation available to a process

Now consider the computation of Figure 7.2. Because there are independent Deposit events (as with the receipt of deposits from independently acting persons), the first attempt at matching the pattern

Deposit(?D,?A)

could result in a match on *either* of the two events Deposit(1000,123) or Deposit(950, 123). Since they are independent, the process may trigger on either event first.

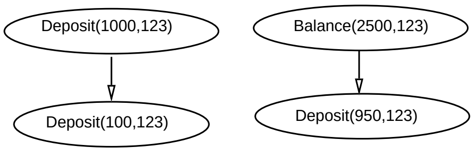


Figure 7.2: Another computation for the teller process

That is, the first match could give ?D either the value 1000 or 950. In a sequence of matches we could get any one of the following sequences of values for ?D in successive process executions:

$$< 1000, 100, 950 >, < 1000, 950, 100 > \text{ or } < 950, 1000, 100 >$$

Because of earliest maximal matching, the teller process cannot trigger on events out of their causal order, and therefore it cannot match the \$100 event before the \$1000 one.

□

Example: *An await statement nested in a when statement*

```
when (?Msg in Msg_Type, ?Data in Data_Type)
        Begin_Transmission(?Msg,?Data) do
    Transmit(?Msg,?Data);
    await Acknowledge(?Msg);
end when;
```

Commentary:

This when statement is a simple "sender" that waits for acknowledgement of a message before transmitting the next one. Note that occurrences of ?Msg and ?Data in the process body, including those in the await statement, have the values determined by the latest match of the trigger. In particular, the **await** statement will wait for an Acknowledge event with a parameter equal to the second parameter of the Begin_Transmission.

□

Example: *A self-triggering clock*

```
when Start or Tick do
    Tick() pause 1;
end when;
```

Commentary:

Reactive statements in a process may be triggered by events that the process itself has generated. In this example, a single Start event will cause the process to fire, which will generate a Tick event one time unit later. This Tick event will again trigger the when statement, which will generate another Tick event after one time unit. The result is a self-triggering clock process that repeatedly generates Tick events every time unit.

□

7.4 Multiple Processes

7.4.1 Syntax

The syntax of a set of processes in a module definition is

process_part	::=	**parallel** *process_list* | **serial** *process_list*
process_list	::=	*process_or_generator* [|| *process_or_generator*]
process_or_generator	::=	*process* | *process_generator*

Process generators are discussed in Section 7.5.

7.4.2 Semantics

This section defines the semantics of a process part preceded by **parallel**. For the semantics of a process part preceded by **serial** see Section 7.6.

Each process in a process part is executed concurrently with and independently of other processes, unless clocked (see Chapter 10) or explicitly synchronized. No assumption is made about the relative speeds of different processes in a module. A module terminates when all its processes have terminated. If a process generates a concurrent module, then that module does *not* have to terminate for the process to terminate.

The execution of processes may be synchronized (1) through the generation and matching of events, (2) through references to shared variables (reference objects), or (3) through references to time in a clocked module. Synchronization leads to dependencies between the events involved.

Events of a particular module are made available to all processes in identical order; this order is consistent with the orderly observation principle. Each process performs its pattern matching independently of other processes.

Process State. At any point during the execution of the module, all processes are either *suspended* or *active*. Initially all processes are in the active state and begin executing.

A process makes a transition from the active state to the suspended state when it encounters a blocking reactive statement for which there is no triggering match in the available events. A suspended process makes a transition to the active state when a triggering match becomes available.

7.4.3 Examples

Example: *Relative speeds of processes*

```
V : var Integer := 0;
...
      when E do V := $V − 1; end do;
||
      when E do V := $V + 1; end do;
```

Commentary:

Processes may execute at arbitrary speeds.

Consider a computation consisting of a sequence of events of action E. Since neither of the processes need wait for the other, we cannot derive from the above code that the value of V always will be in the range −1 . . 1, since *e.g.* the second process may handle any number of E events while the first one reacts to a single E event.

If lock-step execution is desired then one can introduce explicit synchronization.

□

Example: *Explicit synchronization of processes*

```
V : var Integer := 0;
action Sync(Who: Integer);
...
      Sync(1); Sync(2);
||
      when E and Sync(1) do V := $V − 1; Sync(2); end do;
||
      when E and Sync(2) do V := $V + 1; Sync(1); end do;
```

Commentary:

Here neither process can get ahead of the other, and so the value of V always remains within the range −1 . . 1. Note that we let an initializing process generate the first Sync events to start the system.

□

7.5 Process Generators

A designer will often wish to define a number of identical processes, or processes that differ only slightly. Also, it may be that a process should only be defined in a module under certain conditions. *Process generators* allow this.

7.5.1 Syntax

The syntax of process generation is

process_generator ::=
  generation_scheme **generate**
    generator_body
  end *generate_terminator* ';'

generation_scheme ::=
  for *identifier* ':' *type_expression* **in** *expression* '..' *expression* |
  if boolean_*expression*

generate_terminator ::= **for** | **if**
generator_body ::= *process* | *process_generator*

7.5.2 Semantics

For generation scheme. The type_*expression* of the **for** generation scheme must be determinable at elaboration time. The *identifier*, called the *loop parameter*, may occur in the generator body, and should be considered a value of the type_*expression*. See Section 5.3 for a definition of the iteration values of the **for** scheme.

Upon elaboration of the module in which it is defined, a **for** scheme does the following for each iteration. First, all occurrences of the loop parameter in the body are replaced by the current value of the loop parameter. Then, if the generator body is a process, a process is created and elaborated. If the generator body is another *process_generator*, then that *process_generator* is executed.

If generation scheme. The Boolean expression of the **if** generation scheme must be an elaboration time constant. If its value is True, then: if the generator body is a process, that process is created and elaborated; if it is a *process_generator*, the *process_generator* is executed.

7.5.3 Examples

Example: *A "pipeline" of processes*

```
        action Next_Part( K : Integer );
        Pipeline_Length : Integer is 10;
        ...
    parallel
            -- Initialize and pass to pipeline
            Next_Part(1);
        ||
            for J : Integer in 1..Pipeline_Length-1 generate
                when Next_Part(J) do
                    -- Do some work
                    Next_Part(J+1);
                end do;
            end for;
        ||
             -- Get from end of pipeline
             when Next_Part(Pipeline_Length) do
                 -- More work
             end do;
```

Commentary:

A pipeline of ten processes is created by the process generator. These processes are identical except where the loop parameter J occurs. When the individual processes are created, occurrences of J are replaced by the value of the loop parameter for that iteration; for example, the first process created has the definition:

```
when Next_Part(1) do
    -- Do some work
    Next_Part(1+1);
end do;
```

Thus each process waits for a Next_Part event from its predecessor, does some work, and then sends a Next_Part event to its successor in the pipeline.

□

Example: *Conditional generation of a process*

```
module Bank_Teller( Audit_Trail : Boolean ) Return Teller is
    ...
parallel
    ...
    ||
        if Audit_Trail generate
            when (?T in Trans_Type) Transaction(?T) do
                Report_Transaction(?T);
            end do;
        end if;
end module;
```

Commentary:

A process that monitors transactions and reports them to some other agent is only created if the Audit_Trail parameter of the module is TRUE.

□

7.6 Serial Modules

RAPIDE allows the programmer to specify that a module is *serial*; that is, that all processes of the module are synchronized such that only one process is executing at any given time.

7.6.1 Syntax

process_part ::= **serial** *process_list*

The fact that a module is serial is indicated by the keyword **serial** preceding the process definitions, in place of the keyword **parallel**. A module may not have both a **serial** and a **parallel** part.

7.6.2 Semantics

In a serial module, execution alternates between all processes and function calls from other objects; the execution of each process or call is synchronized such that only one is executing at a time. The events generated by a serial module form a linear dependency ordering. Processes still consider events for matching in the same manner as in parallel modules.

At any point during the execution of the module, all processes and function calls are either *suspended*, *ready*, or *active*. Initially all processes are in the ready state. At most one process or call is active at a time in the module; this is the process or call that is being executed.

The transitions between states that may occur for a specific process or call are shown in Figure 7.3. Conditions for these transitions are as follows:

- *ready* → *active*: the process or call is chosen to continue execution.
- *active* → *ready*: the process or call reaches a *switching point* (defined below).
- *active* → *suspended*: the process or call reaches a statement that cannot be immediately executed — for example, a timed statement with a non-zero timing expression, or a blocking reactive statement for which the available events do not form a triggering match.
- *suspended* → *ready*: a condition for the process or call to continue has been achieved — for example, a timed statement where the required amount of time has elapsed, or a reactive statement where a triggering match is now available.

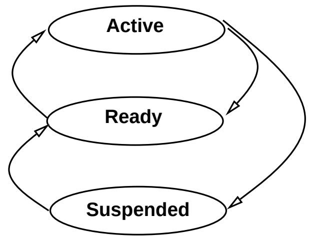


Figure 7.3: State transitions for processes in a serial module

The following statements define the *switching points* of a module:

- An **await** or **when** statement.
- A statement that has a **pause** or **delay** timing clause; specifically, an action call with a timing clause, or a timed statement.
- The completion or termination of a function call. [4] **This may be too liberal, since doing just about anything calls a function (such as getting a Ref object value).**

If a switching point is a statement, the process or call is switched to the ready state immediately before the statement is to be executed. When next made active, the process or function continues, first executing the statement. In the case of the completion or termination of a call, the call returns and another ready process or call must be made active, if one exists.

Events generated by a process or call that has been made active follow in the dependency ordering the events generated by the previously active process or call; thus all events of the module are ordered.

Child Modules of Serial Modules. If a module has an (immediate) child module and both are serial, all processes and calls in the modules are synchronized, and so all events generated by both modules form a linear dependency order.

A serial module that has a child which is not serial is not synchronized with the child.

The rules for children of serial modules apply transitively, so that if a serial module has a child module that is serial, and the child itself has a child module that is serial, all three modules are synchronized. However, if a serial module has a child module that is not serial, and the child has a child module that is serial, there is no synchronization.

Example: *A serial stack module*

```
module Stack_Mod( type T ) return Stack is

    Length : var Natural := 0;

    type Stack_Type is
    union
        Bottom : Triv;
        Elem : record
                    Item : T;
                    Nxt : Stack_Type;
                end record;
    end union;

    Stack_Impl : var Stack_Type := Stack_Type'(Bottom, Unit);

    function Push( Val : T ) is
    begin
        Length := $Length + 1;
        Stack_Impl := Stack_Type'( Elem, (Val, $Stack_Impl) );
    end;

    function Pop() return T is
    begin
        raise Underflow() where $Length = 0;

        Length := $Length − 1;
        declare
            Rtn_Val : T is $Stack_Impl.Elem.Val;
        do
            Stack_Impl := $Stack_Impl.Elem.Nxt;
            return Rtn_Val;
        end;
    end;

    ⋮

serial
    null;
end;
```

Commentary:

This serial module generator returns a stack object. It contains no processes; the serialization means that all function calls will be executed serially, and since the functions contain no switching points, no call will be interrupted before it completes. This insures the integrity of the the state values during the execution of the call.

In contrast, consider a non-serial version of this module. If two independent Pop calls were executed on a stack with Length one, a pathological order of access of the module's state could lead to indeterminate behavior.

□

Example: *A resource monitor*

```
type Resource_Monitor( Initially_Available : Natural ) is
interface
action
    in Request( N : Natural );
    out Provide( ID : Natural );
    in Free( N : Natural );
end interface;

module Resource_Mon( Initially_Available : Natural )
                        return Resource_Monitor(Initially_Available) is
    action Available( Req : Natural );
    Num_Free : var Natural := Initially_Available;
    Request_ID, Next_Request : var Natural;
initial
    Num_Free := Initially_Available;
    Request_ID := 1;
parallel
    when (?N in Natural) Request( ?N )
    declare
        My_Request_ID : Natural is $Request_ID;
    do
        Request_ID := $Request_ID + 1;
        while ?N < $Num_Free do
            await Available(My_Request_ID);
        end while;
        Provide($Request_ID);
        Next_Request := My_Request_ID + 1;
        Num_Free := $Num_Free - ?N;
    end;
  ||
    when (?N in Natural) Free( ?N ) do
        Num_Free := $Num_Free + ?N;
        Available($Next_Request);
    end;
end;
```

Commentary:

This example is a monitor for resource allocation. Users of the module Resource_-Mon request some number of resources by generating a Request event. When enough resources have been made available, a Provide event is generated by the monitor. The internal action Available uses an identification number to identify which caller is next in line to get resources.

□

Chapter 8

Exceptions and Interrupts

8.1 Purpose

The Purpose of Exceptions and Interrupts is to facilitate the smooth handling of unusual or unexpected program behavior.

The concepts of exceptions, interrupts and events are closely intertwined. Each can be viewed as a message. An event is a passive message that may be ignored or observed an arbitrary amount of time after it was generated. An exception is an event whose generation has the immediate side effect of causing the thread of control that generated it to jump to a handler (see below). An interrupt is a event whose matching has the side effect of causing the thread of control that matched it to jump to a handler.

One pertinent difference between interrupts and exceptions is in their point of view, that is, who decides if an event is an interrupt or an exception. The existence of an exception is a decision made by its generator. The existence of an interrupt is a decision made by its observer. This asymmetry results in an an asymmetric syntax for exceptions and interrupts. To generate an exception, an explicit **raise** statement is used. An exception is handled by naming it in a handler. To generate an interrupt, a normal action call statement is used. The generator of an "interrupt" has no guarantee that the receiver will view it as such. In a manner of speaking, all interrupts are maskable. By creating a handler that names an event, a block makes that event an interrupt for itself.

8.2 Syntax

exception_declaration ::=
exception *name* ['(' [*formal_parameter_list* ')'] ';'

raise_stmt ::=
raise [*name* ['(' *parameter_association_list* ')']
[**where** *boolean_expression*]] ';'

The rules for *parameter_association_list* are the same as for function calls; see Chapter 6.

handler ::= **handler**

is *handler_choice*
{ **is** *handler_choice* }
[**else** *stmt_list*]

handler_choice ::= *pattern* '=>' *stmt_list*

Interrupts are, syntactically, simply events generated from action declarations that are interpreted as interrupts. The syntax for interrupt generation is therefore the action call statement.

The names that can be used as basic patterns in a handler_choice are the visible exceptions and actions. The visibility rules for exceptions are the same as those for actions.

A handler may appear in any block. The definition of a *block* may be found in Section 3.5.

8.3 Semantics

Exception Declarations. An exception declaration is syntactically almost identical to an action declaration, and the two declarations both serve the same purpose: they are a means for generating events. The semantics of formal parameters are defined in Section 5.2.2 of the RAPIDE 1.0 Types Language Reference Manual.

An exception declaration may only be called by a raise statement, and patterns referring to exception declarations may only occur in a handler (both defined below).

Raise Statements. A **raise** statement is used to generate an event that causes control to transfer to a handler (defined below).

The name and parameters may be omitted in a raise statement that occurs in a handler; otherwise the name must be present, and must denote an exception.

When a raise statement is executed, if the Boolean expression after the **where** is False, execution continues normally. If the Boolean expression is True, or if there is no such expression, then the actual parameters of the raise statement, if any, are evaluated, and an exception is said to be *raised*.

When an exception is raised, an *exception event* is said to be generated; the parameters of the event are the actual parameters of the raise statement. An exception event causes control to be transferred immediately to a handler.

Handlers. A handler consists of a sequence of handler choices with an optional **else** part. A handler choice is a pattern followed by a list of statements. The basic patterns of the pattern may denote either action declarations or exception declarations.

For any given exception or interrupt only a single handler pattern will match on it. That handler and in particular that handler pattern will be said to have *handled* the exception or interrupt. How that handler is chosen depends on whether an exception or an interrupt has occurred, and will be discussed in the following sections.

Once the handler has been chosen, the handling pattern within the handler is chosen using the earliest maximal matching defined for the await statement (see Section 7.3). If the handler occurs in a reactive statement, the events that participated in the match will become unavailable to the process. The list of statements in the handler choice is then executed.

The **else** part of a handler is chosen for execution only if an exception has been raised which no handler choice can handle. If that condition occurs, the sequence of statements following the **else** is executed. If the handler occurs in a reactive statement, the exception event is made unavailable.

8.3.1 Handling Exceptions

The raising of an exception causes an event to be generated. If that event, plus other exception and interrupt events available to the block that contains a handler, form a match for one of the patterns among the handler's choices, we say that the handler can immediately handle the exception.

If an exception is raised in a block which is the sequence of statements of a **do** statement or a function containing a handler, and the handler can immediately handle the exception, execution of the sequence of statements is abandoned and control is transferred to the handler. The sequence of statements associated with the handler pattern that handled the exception is executed and execution continues as if the end of the block had been reached normally.

Otherwise, for an exception raised in:

- a function's sequence of statements where the function does not handle the exception then the same exception is raised again at the point of the call of the function.
- a **do** statement's sequence of statements where the **do** statement is in a sequence of statements and does not handle the exception, the same exception is raised again immediately after the **do** statement.
- a process, and the exception is not handled in the process, that process is terminated. If the enclosing module's handler can handle the exception then the sequence of statements associated with the handling pattern is executed. The other processes of the module continue to execute normally. If the module's handler cannot handle the exception, the module terminates. If the parent module of the terminated module has not terminated then the exception is raised in that module as though it had been raised in a process of that module that did not have a handler. If the parent module has already terminated, no further action is taken for the exception.
- one of a handler's sequence of statements, the exception is treated as though it was raised in that handler's block and the handler did not handle it (regardless of whether the handler could have handled the exception).

A handler with an **else** part can handle any exception.

There are no predefined exceptions. Predefined interfaces may, however, define exceptions thus achieving the functionality of predefined exceptions in other languages.

8.3.2 Handling Interrupts

Any action referred to in a handler is considered an interrupt. A handler that refers to an interrupt becomes *active* when its block is entered. It becomes inactive when its block is exited. See Section 3.5 for a discussion of when a block is entered or exited.

When a handler becomes active its patterns become available for matching. Only events that are generated after the handler becomes active (and before it becomes inactive) may participate in a match of its patterns.

An interrupt triggers a handler's choice when the interrupt event, plus other exception and interrupt events available to the block that contains a handler, form a match for one of the patterns among the handler's choices.

If an interrupt triggers a handler's choice, the computation in that handler's block is abandoned and control is transferred to the handler. The sequence of statments associated with the handler choice that matched the interrupt is executed and execution continues as if the end of the block had been reached normally.

A single process can cause multiple handlers to be active at one time. If more than one of the handlers activated by a particular process could handle a particular interrupt then the most recently activated of such handlers will handle it.

The special pattern **any** (See Section 18 of the RAPIDE 1.0 Predefined Types and Objects Reference Manual) may be used to allow any event to serve as an interrupt.

A process may interrupt itself by generating a pattern of events that trigger its handler. There is no requirement that interrupt events come from "outside" the block.

8.4 Examples

Example: *Using exceptions*

```
exception Stack_Memory_Error(available : integer);
...
function Initialize_Stack(size : integer) is
begin
   raise Stack_Memory_Error(available_memory)
         where available_memory < size;
   ...
end Initialize_Stack;
...
when start do
   Initialize_Stack(1000);
handler
   is Stack_Memory_Error(?I) =>
      Put_Line("Stack could not be allocated");
      Put("Available memory is only");
      Put_Line(?I);
end when;
```

Commentary:

TBD.

□

Example: *Using Interrupts*

```
   ...
   action in Launch_Missile();
   action in Abort();
   action out Launch_Aborted();
   action out Missile_Launched();
   action out Abort_Ignored();
```

```
...
loop do
   await Launch_Missile =>
      do
         pause 10;
         Missile_Launched();
      handler
         is Abort =>
            Launch_Aborted();
      end;
   handler
      is Abort =>
         Abort_Ignored();
   end await;
end loop;
```

Commentary:

TBD.

□

Chapter 9

Modules

Objects are the basic program-structuring facility of RAPIDE. Objects are of interface and function types. An object of a function type is called a function, and an object of an interface type is called a *module*. An object may encapsulate a number of related parts of a large program, including any number of sub-objects (objects contained within an object), or it may be as small as a single datum with a few associated operations. A module interface, or interface for short, describes a required set of constituents of a module, and the way that these constituents may be accessed by other objects. For example, if a module contains a private data structure and several public access functions, then the interface will list the names of the access functions and give the types of each function parameter and result.

Every module definition must name an interface for the module. The most common form of module definition is a module *generator*. A module generator may be *called* to create a module.

The relationship between a module and its interface is that the interface is a type and the module is an element of this type. This is analogous to the relationship between the type Integer and the object 3. In fact, we may regard the type Integer as an interface, and 3 as a module with this interface.

The concepts of module and interface are general enough that all of the predefined types may be regarded as special cases of interfaces, provided by tools. Tools also predefine modules of the predefined types. In this view, 3, for example, is a predefined module containing private data together with integer operations addition, subtraction, and so on. The integer expression 3+4 is considered an abbreviation for the function call `3.+(4)`, which is a call to the addition function of 3, with 4 as the parameter. Of course, the actual implementation of integers and integer expressions is essentially the standard one; we present this general view of the language simply to emphasize the uniform treatment of all types. In addition, the interface to each predefined type is a convenient place to put all the constraints (written in the constraint sublanguage) of the type.

In this chapter we present the syntax and semantics for defining, creating, initializing, and finalizing modules.

9.1 Module definitions

9.1.1 Purpose

Modules of a particular interface type may be structured in many different ways, and differently structured modules of the same type may coexist within a single program.

Modules are created using *module literals*, *module generators*, and *allocator functions*. A module literal defines the structure of a module, and its evaluation results in a module being created. A module generator declares a function that returns a unique module every time it is called. A module generator defines the structure of the modules it returns. Each call to a module generator creates a new module. Allocator functions are defined in Section 9.8.

[5] **The type of a module generator.**

Module literals and module generators both define the structure, or internal implementation, of modules. Thus we call module literals and generators *module definitions*.

9.1.2 Syntax

```
module_generator ::=
        module identifier '(' [ parameter_list ] ')' return interface_expression
        is
           module_body
        end [ module ] [ identifier ] ';'

module_literal ::= module
                      module_body
                   end [ module ]

module_body ::= { module_constituent }
                [ initialization_part ]
                [ process_part ]
                [ handler_part ]
                [ finalization_part ]

initialization_part ::= initial [ '(' parameter_list ')' ]
                            statement_or_derivation_list

statement_or_derivation_list ::= ( statement | module_derivation )
                                 { ( statement | module_derivation ) }

process_part ::= ( parallel | serial ) process_definition_list

process_definition_list ::= process_definition { '||' process_definition }

process_definition ::= [ label ':' ] statement_list
                     | [ label ':' ] process_generator
                     | module_derivation { module_derivation }

handler_part ::= handler ( handler | module_derivation )

finalization_part ::= final statement_or_derivation_list

module_constituent ::= module_derivation
                     | declaration
                     | placeholder_declaration
                     | connection
```

Module derivations are presented in Section 9.3. Process generators are presented in Section 7.5.

9.1.3 Examples

Example: *A simple module*

```
type Counter is interface
   Increment, Decrement : function();
   Value : function() return Integer;
end Counter;

C : Counter is module
            State : var Integer := 0;
            function Increment() is
               begin State := $State + 1; end;
            function Decrement() is
               begin State := $State − 1; end;
            function Value() return Integer is
               begin return $State; end;
         end module;
```

Commentary:

The declaration of C introduces a name designating a Counter module, and the module it designates is the one defined by the subsequent module literal. The literal provides functions for each of the function names declared by the interface type Counter, and in addition defines an integer reference constituent, which maintains the state of the counter.

□

Example: *Module generator with parameters*

```
module New_Counter (Initial_Value : Integer) return Counter is
   State : var Integer := Initial_Value;
   function Increment() is ...
   function Decrement() is ...
   function Value() return Integer is ...
end New_Counter;

module Zero_Counter () return Counter is
   State : var Integer := 0;
   function Increment() is ...
   function Decrement() is ...
   function Value() return Integer is ...
end Zero_Counter;

C : Counter is New_Counter(0);
```

Commentary:

New_Counter and Zero_Counter designate generators that, when called, create Counter modules. The declaration of C calls the first generator. Elaboration of C's declaration executes the call. The call creates a Counter module with State initially zero. One could equally well have declared C as:

```
C : Counter is Zero_Counter();
```

□

9.2 Type and module conformance

9.2.1 Purpose

An interface type identifies constituents required of modules of the type. Constituents defined in an interface are accessible outside of modules of the type. Also, a module may define additional constituents not found in its interface.

9.2.2 Semantics

The type of a module is determined when the module is created. The type of a module created by a module generator is the type specified after the keyword **return** in the module generator, following parameter association. The type of a module created by a module literal is the type of the literal, determined in one of two ways:

1. If the literal appears as the expression on the right-hand side of the keyword **is** in an object declaration, then the type of the module literal is the type on the left-hand side of the keyword **is**.
2. If the literal is the expression in a qualified expression (see Section 4.3.8), then the type of the literal is the type given in the qualified expression.

These are the only allowed uses of a module literal.

Lastly, the type of a module returned by an allocator function is that returned by the corresponding module generator call; see Section 9.8.

Conformance of module constituents to type constituents. An interface type may contain name declarations in its **provides** part. All modules of such a type must provide corresponding full declarations.

For object name declarations, the corresponding object may be provided by an object declaration in a module definition, or by a tool. A module definition that does not contain a corresponding object declaration may still create modules of the interface type, if a tool can supply the needed declaration. Tool-supplied objects must satisfy all their constraints. If a tool cannot determine such an object, then all module definitions of the type must do so. (Tool-supplied objects cannot be used when module definitions provide them.)

These rules of constituents provided by module definitions or tools also apply to types and module generators declared in interface types.

The full declaration corresponding to a name declaration must *satisfy* the name declaration according to the following rules:

- For an object name declaration, the object declaration must have the same name as the object declaration, and its type must be a subtype of the name declaration's type.
- For a module generator name declaration N and a module generator declaration G, conformance is analogous to function subtyping: for function types F1 and F2 whose parameters and return type are the parameters and return type of G and N respectively, G satisfies N iff they have the same name and F1 <: F2. A corollary of this definition is that all legal calls to N will be legal calls to G.
- For a type name declaration, the type declaration must have the same name as the type name declaration. If the name declaration has a subtype constraint, the type of the corresponding declaration must be a subtype of the constraint.
- For a type constructor name declaration N and a type constructor declaration T, conformance is also analogous to function subtyping: for function types F1 and F2 whose parameters are the parameters of T and N respectively, and where both F1 and F2 return Root, T satisfies N iff they have the same name and F1 <: F2. A corollary of this definition is that all legal calls to N will be legal calls to T.

9.2.3 Examples

Example: *Tool-dependent declarations*

```
type T is interface
   I : Integer;
   F : function();
end T;

module G() return T is end;

X, Y : T is G();
```

Commentary:

The expression X.I is legal since X is of type T and I is declared by T. The declaration of I specifies that it names an Integer object. Module generator G, however, does not associate an object with I. In the absence of such an explicit association, a tool must associate any Integer object with X.I, or the generator G is illegal. A tool may supply different objects for X.I and Y.I.

Tool-supplied objects are also associated with X.F and Y.F. Possible values include:

```
function() is begin null; end

function() is
   exception Tool_Dependent();
begin
   raise Tool_Dependent();
end
```

□

9.3 Derivation

9.3.1 Purpose

A module derivation is a mechanism for reusing parts of module definitions in other modules. Module derivations are syntactically similar to interface derivations.

9.3.2 Syntax

module_derivation	::=	**include** *module_generator_call*
		[(**only** | **except**) *derived_name_list*]
		[**replace** *renaming_list*] ';'
module_generator_call	::=	*function_application*
derived_name_list	::=	*derived_name* { ',' *derived_name* }
derived_name	::=	*identifier*
	|	*type_expression* ''' '(' *identifier* ')'
renaming_list	::=	*renaming* { ',' *renaming* }
renaming	::=	*derived_name* **by** *identifier*

9.3.3 Semantics

A module generator call is used to identify the module definition to derive from. The effect of a module derivation (sometimes simply called a derivation in this chapter), however, does not include execution of the call. The module definition being derived from is referred to as the *source* module definition (or simply source). A module definition may contain multiple module derivations, and thus multiple sources.

Module derivations may appear in any of the five parts of a module definition: constituent list, initialization statements, process list, handler, finalization statements. The meaning of a derivation is dependent on the part in which it appears: derivations in a constituent list can only derive constituents of the source, derivations in initialization statements can only derive the (entire list of) initialization statements of the source, derivations in a process list can only derive processes, derivations in a handler can only derive entire handlers, and derivations in finalization statements can only derive the (entire list of) finalization statements of the source. Derived items become part of the module definition that contains the derivation, at the point of the derivation. The lexical order of items derived by a derivation is the same as their order in the source of the derivation.

A derivation may include a list of derived names to specify the items of the source to be included or excluded. A derived name is either an identifier or a qualified expressions. Qualified expressions may be used to differentiate between overloaded identifiers. All derived names must denote an item that the derivation may derive. Derivations in initialization, finalization, or handler parts may not contain a derived name or renaming list.

If the reserved word **only** is used, then only the items named in the associated derived name list are derived. If the reserved word **except** is used, then all the items of the source that can be derived by the derivation, but are not named in the derived name list, are derived. If neither **only** nor **except** is used, all items that may be derived are derived.

Derived names in a renaming list that are on the left-hand side of the keyword **by**, must name items derived by the module derivation. A derivation containing a renaming list is elaborated exactly as if it did not contain the renaming list. However, once this part of the elaboration is done the following substitution is performed:

1. All occurrences of identifiers getting a designation at the outer-most level resulting from the elaboration of the derivation, and which occur immediately to the left of the keyword **by** in a renaming, are replaced by the identifier that occurs immediately to the right of **by**.
2. All occurrences of identifiers which occur immediately to the left of the keyword **by** in a renaming and which refer to a name substituted according to (1) are replaced by the identifier that occurs immediately to the right of **by**.

The substitution is simultaneous for all renamings. The elaboration of a derivation may not entail a recursive attempt to elaborate the derivation.

9.3.4 Examples

Example: *A complete derivation*

```
module Zero_Counter() return Counter is
    include New_Counter(0);
end Zero_Counter;
```

Commentary:

> This form of derivation reuses only the constituents of the source module definition. Since the source contains only constituents (no initialization, processes, etc.), we effectively derive the entire source. Here we derive the declarations of State, Increment, Decrement, and Value.

□

▬[6] **need more meaningful, non-contrived examples**

Example: *Initialization, finalization, and process derivation*

```
type Root is interface end;

module M() return Root is
   action A(I : Integer);
initial
   A(0);
parallel
   P1: A(1);
|| P2: A(2);
final
```

```
   A(100);
end M;

module N() return Root is
   include M();              -- D1
initial
   A(3);
   include M();              -- D2
parallel
   P3: A(4);
|| include M() only (P2)     -- D3
      replace (P2 by P4);
final
   include M();              -- D4
   A(200);
   include M();              -- D5
end N;
```

Commentary:

N defines modules equivalent to the following. Items are labeled with the derivation resulting in their declaration.

```
module
   action A(I : Integer);     -- D1
initial
   A(3);
   A(0);                      -- D2
parallel
   P3: A(4);
|| P4: A(2);                  -- D3
final
   A(100);                    -- D4
   A(200);
   A(100);                    -- D5
end
```

□

Example: *Illegal derivation*

```
module M() return Counter is
   include New_Counter(0) only (Increment, Value);   -- Illegal!
end M;
```

Commentary:

This module definition is illegal since it is equivalent to the following definition in which State is not declared:

```
module M() return Counter is
   function Increment() is
      begin State := $State + 1; end;
```

```
    function Value() return Integer is
      begin return $State; end;
  end M;
```

□

9.4 Module creation and initialization

9.4.1 Semantics

Modules are created in three ways:

1. By evaluation of a call to a module generator.
2. By evaluation of a module literal.
3. By evaluation of a call to an allocator function. (See Section 9.8.)

Evaluation of a generator call begins with parameter evaluation and association. The rules for parameter evaluation and association, as well as rules for overload resolution of generator calls, are the same as those for function calls (see the Guide to the RAPIDE 1.0 Language Reference Manuals). After parameter association, the call executes sequentially as follows:

1. Create a new module.
2. Call the predefined Start action of the module.
3. Elaborate the constituents of the module.
4. Execute the new module's initialization statement list, if any.
5. Elaborate the processes (if any) of the new module.
6. Return the new module.

Module literals are evaluated in this same order.

As a result of this initialization sequence, the state (e.g., local objects) of each module may be distinct. For example, if a module definition contains an object declaration whose value is determined by a generator call, then the object declaration is elaborated (calling the generator and forming a new module) each time the module definition is evaluated.

The parameters following the keyword **initial** (if any) in a module definition are called the *initialization parameters* of the module definition. Initialization parameters may not include type parameters, and all initialization parameters must have default associations. The execution of a module's initialization statements begins by associating actual parameters with initialization parameters. Modules created by a module generator or module literal will always use the default associations. Modules created by an allocator function are discussed in Section 9.8. After parameter association, the initialization statement list is executed.

Processes, like state, are elaborated for each new module. For example, if a module generator defines three processes, then two calls to the generator will elaborate six processes.

As a result of this creation sequence, the Start event (see Section 9.5) of a module causally precedes events generated by the elaboration of the module's constituents. Further, the Start event causally precedes events generated by initialization statements, processes, and finalization statements of the module.

The *parent* of a module m is the innermost enclosing module of the generator call, literal, or allocator call that creates m. The parent relationship is used in defining the semantics of mode-out events, serial modules, connections and clocks. If module p is the parent of module c, then c is a *child* of p. Not all modules have a parent. A module has at most one parent, and may have many children.

9.4.2 Examples

Example: *Module generator calls*

```
C1, C2 : Counter is New_Counter(0);
C3     : Counter is New_Counter(0);

function P (X : Counter) is ... end P;
...
P(New_Counter(0));
```

Commentary:

Here we call New_Counter four times: to obtain the objects C1, C2, C3, and the X parameter of P.

□

Example: *Start event*

```
module M() return Root is
   action A (I : Integer);
   function F() return Integer is
      begin  A(1);  return 3;  end F;

   I : Integer is F();
initial
   A(2);
parallel
   A(3);  ||  A(4);
end M;

X, Y : Root is M();
```

Commentary:

The process elaborating the declarations of X and Y executes two module generator calls. Each call will generate events Start, A(1) and A(2). Each new module then generates A(3) and A(4). For each module, the Start event precedes A(1), A(1) precedes A(2), and A(2) precedes A(3) and A(4). Since two modules are generated,

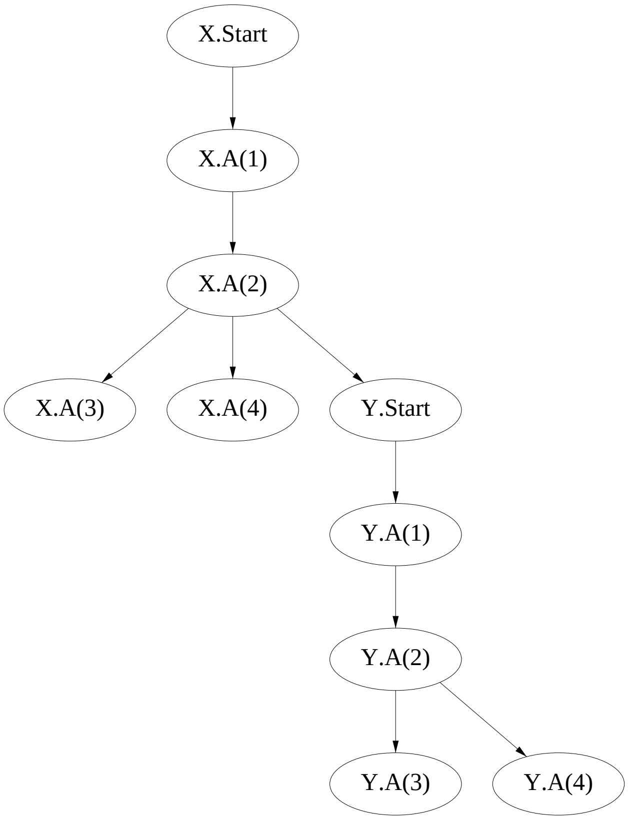


Figure 9.1: Generation and execution of two concurrent modules.

two Start events, two A(1) events, etc., will be generated. The Start, A(1), and A(2) events of X precede the corresponding events of Y since the elaboration (and initialization) of X precedes that of Y.

The events generated, together with their ordering, are depicted in Figure 9.1. For explanatory purposes, action names in the figure have been prefixed by the name of the module that generated the events.

□

Example: *Initialization*

```
type Log_File is interface
   function Write (S : String);
end Log_File;

module Make_Log_File(Name : String is "log_file") return Log_File is
   F : File is Create(Out_File, Name);

   function Write (S : String) is
   begin
      F.Put(S);
```

```
      end Write;
   initial
      F.Put_Line("This is my log file.");
      F.New_Line;
   end Make_Log_File;
```

Commentary:

A module may contain an initialization part that is executed as part of module's creation. Modules generated by Make_Log_File contain no processes. However, Make_Log_File does contain an initialization statement list that is used to set-up new modules' state.

□

Example: *Initialization deadlock*

```
   type T is interface
      action out A();
   end T;

   module M() return T is
   initial
      await A;
   end M;
```

Commentary:

A process calling M is suspended until an A event is observed. But this suspension takes place before a name for the new module exists (i.e., during execution of a generator call). Thus, no A events *can* be generated. The calling process will be suspended forever.

□

9.5 Start action

9.5.1 Purpose

Start events are used to instrument the initialization of a module.

9.5.2 Syntax

```
action out Start();
```

9.5.3 Semantics

The action Start is implicitly declared as an out-action in every interface type. The Start action of a module is implicitly called during creation of the module (see Section 9.4). Start may not be called explicitly.

9.6 Finish action

9.6.1 Purpose

Finish events are used to instrument the finalization of a module.

9.6.2 Syntax

action Finish();

9.6.3 Semantics

The action Finish is implicitly declared in every module generator and module literal. The Finish action of a module is implicitly called when the module is finalized (see Section 9.9). Finish may not be called explicitly.

9.7 Self

9.7.1 Purpose

The name Self allows the module to refer to itself.

9.7.2 Syntax

Self : T;

9.7.3 Semantics

Self is implicitly declared as a provided object of every interface type. The type of Self is the same as the type in which it is declared. A module definition may not provide an object declaration for Self; a tool-supplied object declaration is always used.

9.8 Allocator function

9.8.1 Purpose

Allocator functions allow a module m to create new modules with structure similar to m. [7]

It's not clear to me that this section is needed. It introduces parameters to the initial part, and a New function which is similar but not identical to the predefined polymorphic New. WM 8/23/95

9.8.2 Syntax

function New (...) **return** T;

9.8.3 Semantics

An allocator function, named New, is implicitly declared as a constituent of every module definition. It is declared as the last constituent of the module definition.

When appearing in a module generator, the parameter list of an allocator function is that formed by appending the initialization parameters to the parameter list of the generator, and the result type of the function is that given in the module generator. When appearing in a module literal, the parameters of an allocator function are the initialization parameters, and the result type is the same as the type of modules returned by the literal.

An allocator function may only be called by the module in which it is declared. To enforce this, interface types may not declare functions named New. Also, modules may not supply a function (object declaration) for an allocator function. A tool-supplied function is always used.

The effect of calling an allocator function declared in a module generator is equivalent to using the corresponding actual parameters of the allocator as actual parameters of the generator, using the corresponding actual parameters of the allocator as actual initialization parameters, and then executing a call to the generator. The effect of calling an allocator function declared in a module literal is to do the same association of initialization parameters, and then again evaluate the literal. In both cases the allocator returns the newly created module.

9.8.4 Examples

Example: *An allocator function*

```
module Stacks (type Item) return Stack(Item) is
   Items : array (Integer) of Item is Null(array (Integer) of Item);
   Last  : var Integer := 0;

initial (I : array (Integer) of Item is Copy(Items);
         L : Integer is $Last)
   Items := I;
   Last  := L;
   ⋮
```

```
end Stacks;
```

Commentary:

The following allocator function is implicitly declared in this module generator:
[8] **need Copy for arrays**

```
function New (type Item;
              I : array (Integer) of Item is Copy(Items);
              L : Integer is $Last) return Stack(Item);
```

Example calls to the allocator are:

```
New(Item)
New(Item, (1 .. 2 => Items(1)), 2)
```

[9] **aggregates for arrays**

In this example the default initial state of modules returned by the allocator function is the state of the module from which the allocator is called. Thus, by default, the allocator behaves like a copy operation. Alternatively, we could have made the default initial state the same as the initial state given in the module constituents, an empty stack:

```
initial (I : array (Integer) of Item
             is Null(array (Integer) of Item);
         L : Integer is 0)
```

□

Example: *A polymorphic allocator function*

```
module Make_Stack (type Element) return Stack(Element) is
  ⋮
end Make_Stack;
```

Commentary:

The allocator function implicitly declared here is the following:

```
function New (type Element) return Stack(Element);
```

Thus a Stack(Integer) module could call its allocator function to create a Stack(Boolean) module. Namely, the type of a module returned by an allocator function need not be the same as the type of the module that called the allocator function.

□

Example: *Spawning using an allocator*

```
type Sieve_Node is interface
   action out Check(N : Integer);
end Sieve_Node;

module Start_Sieve () return Sieve_Node is
  Me    : var Integer := 0;
  Child : ref(Sieve_Node) is Nil(Sieve_Node);
initial (New_Prime : Integer is 2)
  Me := New_Prime;
  Standard_Output.Put(New_Prime);
parallel
  when (?Next in Integer)Check(?Next) where ?Next mod $Me /= 0 do
    if Child.Is_Nil() then
      Child := New(?Next);
    else
      $Child.Check(?Next);
    end if;
  end when;
end Start_Sieve;

declare
  Start_Node : Sieve_Node is Start_Sieve();
do
  for I : Integer in 3 .. Max_Int do
    Start_Node.Check(I);
  end for;
end;
```

Commentary:

A call to this generator will always create a module with $Me referencing 2. Thereafter, sieve nodes are created by other sieve nodes, using the allocator function. The object referenced by Me of nodes created by the allocator function are other than 2.

Execution of the block-statement will print the positive, prime integers. One Sieve_Node module is created for each prime. Integers are passed from one node to another, to determine if they are prime. When a new prime is found, a new node is created, and the node prints the new prime.

□

9.9 Module states and finalization

9.9.1 Semantics

A module is in one of four states:

1. **running:** At least one of the module's processes has not yet completed.
2. **completed:** All of the module's processes have completed, or the module has no processes.

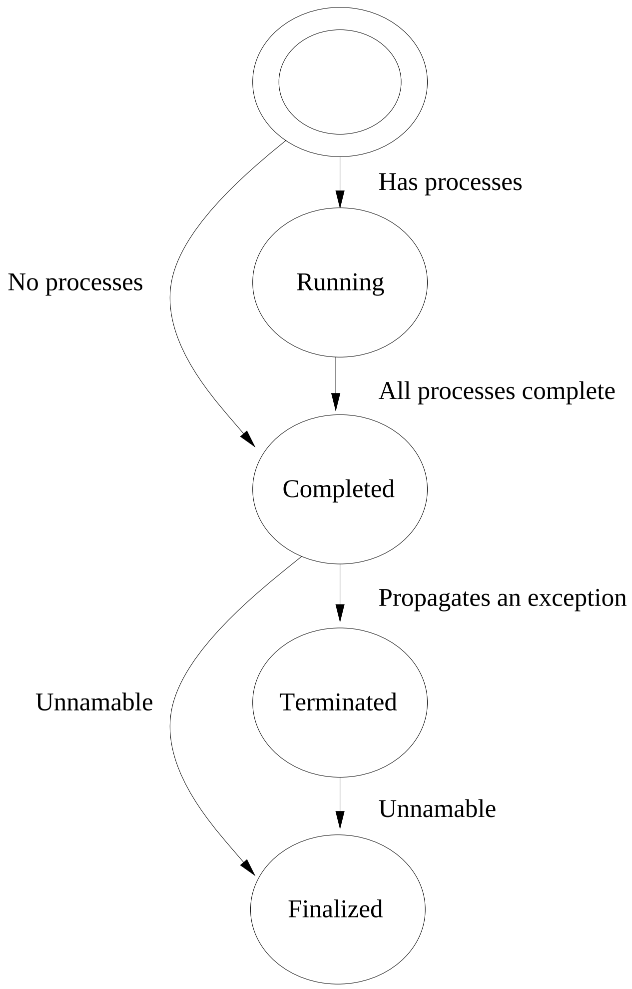


Figure 9.2: States of a module.

3. **terminated:** The module has propagated an exception.
4. **finalized:** All the module's processes (if any) have completed and the module is no longer namable anywhere in the computation.

Figure 9.2 depicts these states in a state-transition diagram. The states are mutually exclusive: a module is in only one state at a time.

Processes *complete* when they finish executing their statement list. Completed processes no longer execute. Exceptions propagated by a process may be handled by the module of the process in the handler given in the module's definition. If a module propagates an exception, the exception is propagated to the module's parent and all modules linked to the module (see Section 7.2). That is, the exception is propagated like an out-event. When a module propagates an exception, all its processes (if any) become completed and the module becomes terminated. If a module propagates an exception during elaboration of its constituents or during execution of its initialization, then processes are not elaborated and the module is unnamable since the newly created module has not yet been returned. Thus, the module is finalized.

The statement list in the finalization part (if any) of a module is executed when the module becomes finalized, provided that the module has not become finalized due to propagation of an exception from elaboration of constituents or initialization. Then, the implicitly-defined action Finish of the module is called. The Finish event causally follows any events generated by the finalization part. The execution of the finalization part, together with the call to Finish, is referred to as the finalization of the module.

Modules that are namable are not finalized. The loss of a name for a module can thus cause finalization. Suppose the loss of a name were an event. Then, all events generated by the finalization of a module causally follow all losses of names for the module.

The boolean-valued attributes Running, Completed, and Terminated are defined for all modules. These attributes return True when the module is in the corresponding state.

9.9.2 Examples

Example: *Finalization of a passive module*

```
module Log_File(Name : String is "log_file") return ... is
   F : File is Create(Out_File, Name);

   function Write (S : String) is
   begin
      F.Put(S);
   end Write;
initial
   F.Put_Line("This is a log file.");
   F.New_Line;
final
   F.New_Line;
   F.Put_Line("All done.");
   F.Close;
end Log_File;

function P() is
   Log : ... is Log_File();
begin
   Log.Write("Hello log.");
end P;
```

Commentary:

A process executing P will generate a module named Log, call the Write function of Log, and then make Log unnamable by exiting its scope. When the module designated by Log becomes unnamable, it is finalized. Thus the return-event (see Section 6.4.3) of the call to Write will causally precede finalization of Log. However, the return-event of a call to P will be causally independent of the finalization of Log. [10] **Counter intuitive; make p'return -> finalization.**

□

Example: *Finalization of an active module*

```
type T is interface
   action out Stop();
end T;

module M() return T is
   action B();
initial
   B();
parallel
   when B do
      B();
   handler is
      Stop => exit;
   end when;
final
   ⋮
end M;

function F() is
   X, Y : T is M();
begin
   X.Stop();
end F;
```

Commentary:

Sending the Stop event to X completes the process of X. The process of Y, however, never completes. X and Y become unnamable when F is finished executing. Then, X becomes finalized and executes its finalization part. Y never becomes finalized and never executes its finalization part.

□

Example: *Forever namable*

```
function F() is
   X : interface
         function Value() return Integer;
       end is
       module
         function Value() return Integer is
            begin  return 42;  end Value;
       end;

   Y : interface end is
       module
       parallel
         while True do
            Standard_Output.Put(X.Value);
         end while;
       end;
```

```
begin
   null;
end F;
```

Commentary:

An execution of F will create two modules, X and Y. Though the scope of X and Y is exited when F returns, Y does not finalize since it has an executing process. Since X is namable by Y, X does not finalize either. Neither module ever finalizes.

□

Chapter 10

Clocks and Time

10.1 Introduction to RAPIDE's Timing Model

The purpose of the RAPIDE timing model is to allow designers to describe and analyze time sensitive prototypes. This includes prototypes which use timing information as an analysis tool (e.g., for determining performance) as well as prototypes of systems with timing constraints on their response time.

A clock is a monotonically increasing counter. The rate at which it increases is not related to any physical unit, but rather is controlled by a computation. This kind of clock provides what is often referred to as *simulation*-time, as opposed to real-time.

We refer to a clock's counter as time, and an increase in time as a tick. Clocks tick when there are no more events to be generated at the current time. An event may be timed with respect to one or more clocks. Further, events have start and finish times, modeling event duration.

A single computation may contain multiple clocks. Clocks may be totally independent or related. Clocks can be related by (1) relating the time of a clock to a global, common metric, (2) relating the time of a clock to a global, common metric and specifying a period and accuracy, or (3) directly relating the time of one clock to the time of another clock.

Independent clocks, called basic clocks, are defined in Section 10.2. Clocks related to a common metric are synchronous clocks and are defined in Section 10.5. Regular clocks (see Section 10.6) are synchronous clocks with a specified period and accuracy. Lastly, slaved clocks, in Section 10.7, are defined in terms of another clock. The predefined types Clock, Synchronous_Clock, Regular_Clock and Slaved_Clock specify these classes of clocks. The subtyping relationships between these clock types is shown in Figure 10.1. The complete specification of the predefined clock types is given in Chapter 17 of the RAPIDE 1.0 Predefined Types and Objects Reference Manual.

Sections 10.3, 10.4, and 10.8 discuss how a computation controls a clock. Section 10.3 shows how time is used to schedule events, Section 10.4 shows how time is used to suspend processes and Section 10.8 presents the conditions under which a clock can tick.

10.2 Basic Clocks

10.2.1 Purpose

The purpose of a basic clock is to provide the minimal functions common to all clocks.

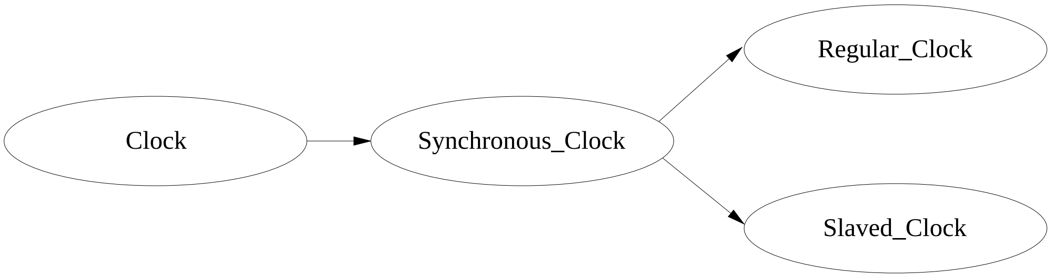


Figure 10.1: Clock types subtype relationships (A $\rightarrow$ B means A >: B).

10.2.2 Syntax

```
type Clock <: interface
   type Ticks <: Natural;

   function Now    ()            return Ticks;
   function Start  (E : event()) return Ticks;
   function Finish (E : event()) return Ticks;
   function Length (E : event()) return Ticks;
end Clock;

module Make_Clock () return Clock;
```

10.2.3 Semantics

Basic clocks are predefined by the type Clock. Each Clock defines a type Ticks. Ticks are time, or the values of a clock's counter.

The function Now returns the current value of a clock's counter.

The function Start returns the start time of an event. If the event is not *related* to the clock, the exception Timing_Error is propagated. An event e is related to a clock c if and only if:

- e is generated by the parent module of c,
- e is observable by the parent module of c, or
- e is generated using a timing clause that names c.

(Parent modules are discussed in Section 9.4 and timing clauses in Section 10.3.)

The Finish function returns the finish time of an event and may also propagate Timing_Error. For all events E and Clock modules C, C.Start(E) is less than or equal to C.Finish(E). Length simply returns the difference between the finish and start times of an event (C.Length(E) = C.Finish(E) − C.Start(E)) and may propagate Timing_Error as well. Note that type Ticks is used for both absolute time (e.g., result of functions Now, Start, and Finish) and relative time (e.g., result of function Length).

Lastly, time is consistent with the causality relationship. For any two events E1 and E2, and Clock C, if E1 causally precedes E2 then C.Finish(E1) is less than or equal to C.Start(E2).

10.3 Timing clauses

10.3.1 Purpose

The purpose of a timing clause is to model the time taken by events. A timing clause determines the start and finish times of an event, and may affect the scheduling of an event. Timing clauses may only be applied to action calls.

10.3.2 Syntax

timing_clause ::= **pause** *timing_expression*
| **delay** *timing_expression*
| **in** *timing_expression*

timing_expression ::= *expression* | *type_expression*

10.3.3 Semantics

A timing clause includes an expression. The expression may denote a single object, or a type. When an object is given, the expression must be of type C.Ticks for some Clock module C. If a type is used, then it must be a subtype of C.Ticks for some Clock C. C is called the named clock of the timing clause.

An example timing clause that uses a type is:

```
C : Clock is Make_Clock();
action A();
...
A() in C.Ticks range 3 .. 5;
```

The execution of an action call with a timing clause begins by evaluating names, parameters, and the timing expression in some order. First, consider timing clauses that specify a single object rather than a type. Let t denote the object.

Consider timing clauses that use **pause**. Execution of the statement takes t Ticks relative to the named clock. That is, if the call begins when the Now function of the named clock yields n, then the call completes when Now() yields $n + t$. The event is generated and observable at time $n + t$. The start time of the event is n and the finish time is $n + t$.

An action call that includes **delay** is equivalent to the same call, substituting **pause** for **delay**, with the following exception. The action call generates an event with a start time s and a finish time f. Any event observable at time t relative to the named clock, where $s <= t <= f$, is made unavailable for matching by the process executing the timing clause. The event being generated is thus always made unavailable to the generating process.

A call that includes **in**, and a timing clause that evaluates to object t where t is zero, is equivalent to the same call without a timing clause. For $t > 0$, the execution of the statement takes zero time, but after t Ticks (Now()+t) the process generates the event with the previously evaluated names and parameters. The causal order of the event is the same as if it were generated at the correct time (Now()+t), without using a timing clause. Events generated by a process at the same time, using action calls that include **in** and a non-zero time, are causally ordered in the order that the actions calls were executed, if any; further, such events are the first events generated by a process at that time.

The start and finish times of events generated by action calls without timing clauses, and by action calls with **in** timing clauses, are the same: the time the event is generated. In any case, events are generated and observable at their finish times.

A timing clause that specifies a subtype is equivalent to specifying an arbitrary object of the subtype. That is, tools select an arbitrary object. If the type defines no objects, then no event is generated and the predefined exception Timing_Error is raised.

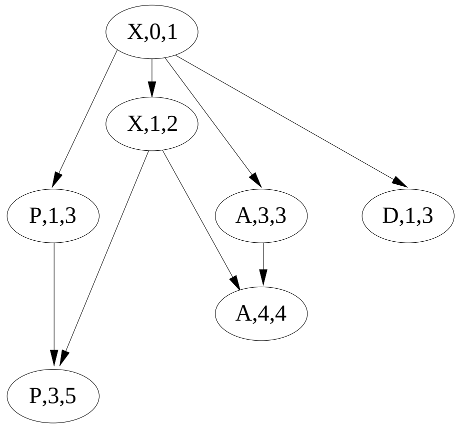


Figure 10.2: Timing clause example.

10.3.4 Examples

Example: *Three kinds of timing*

```
module
   C   : Clock is Make_Clock();
   Two : C.Ticks is 2;
   X, P, D, A : action();
parallel
   X() pause C.Ticks(1);
   X() pause C.Ticks(1);
|| when X do  P() pause Two;  end when;
|| when X do  D() delay Two;  end when;
|| when X do  A() in    Two;  end when;
end module
```

Commentary:

This program generates the computation shown in Figure 10.2. Events are labeled with their action, start time, and finish time. The graph has been arranged so that the vertical position of events is consistent with their finish times.

Consider if 100 X events were generated, one tick apart. One hundred P events would be generated but the last one would finish at time 201. The pausing process gets further and further behind the X events. Only 34 D events would be generated, since two-thirds of the X events are lost (made unavailable) by the delaying process. The last D event would finish at time 102. Lastly, one hundred A events would also be generated, the last one finishing at time 102.

□

Example: *Timing clauses with* **in**

```
module
   action A (I : Integer);
   C : Clock is Make_Clock();
process
   A(1) in C.Ticks(1);
   A(2) in C.Ticks(1);
   A(3);
end module;
```

Commentary:

This example will generate an A(3) event at time zero, an A(1) event at time 1, and an A(2) event at time 1. The events will be causally ordered as A(3) -> A(1) -> A(2).

□

Example: *Critical paths*

```
module
   Flight_Lands, Baggage, Rental_Car, Hotel : action();
   C : Clock is Make_Clock();
initial
   Flight_Lands();
process
     when Flight_Lands do
        Baggage() pause C.Ticks'(2);
     end when;
||   when Flight_Lands do
        Rental_Car() pause C.Ticks'(1);
     end when;
||   when Baggage and Rental_Car do
        Hotel() pause C.Ticks'(3);
     end when;
end;
```

Commentary:

Flight_Lands is generated at time 0 (on clock C), [11] **C ticks before flight-lands?** Baggage at time 2, and Rental_Car at time 1. Thus, the last process is triggered at time 2, and Hotel is generated at time 5. Figure 10.3 graphically depicts this computation in two ways. The upper graph uses ellipses to show the length of events. The time-line in the upper graph gives the start and finish times of each event. The lower graph labels edges with the length of the event at the head of the edge. From a graph such as this, it is easy to determine, for example, that if we shorten the length of Baggage, Hotel will happen earlier.

□

10.4 Timed statements

10.4.1 Purpose

The purpose of timed statements is to allow time to pass without generating an event.

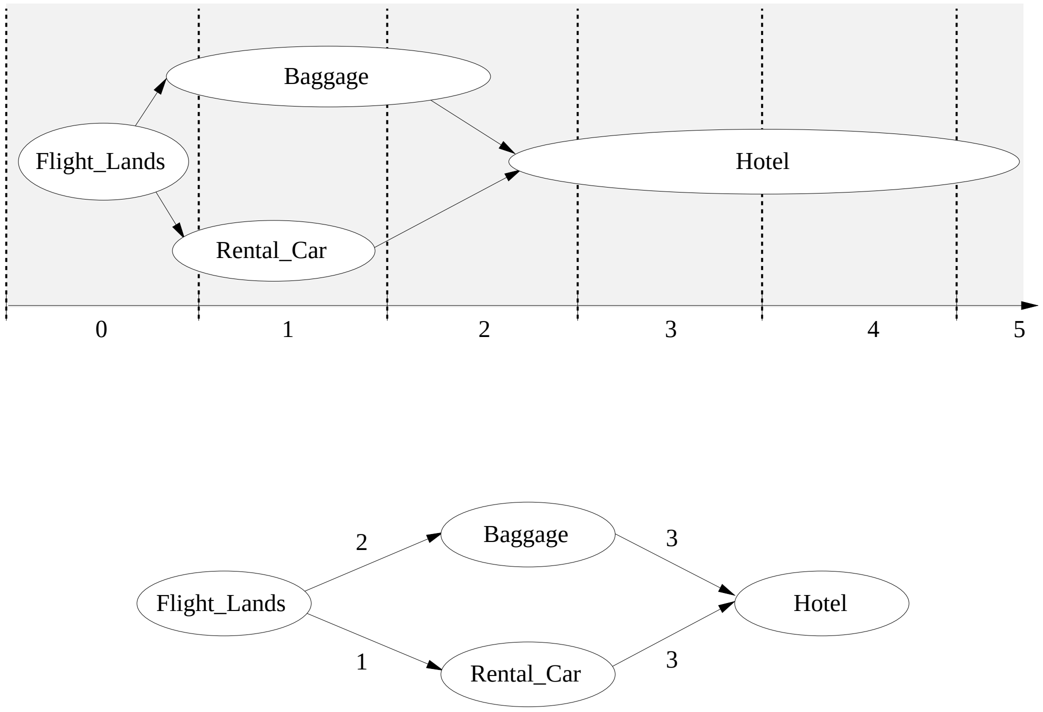


Figure 10.3: Timed events and critical paths.

10.4.2 Syntax

timed_statement ::= **pause** *timing_expression* ';'
| **delay** *timing_expression* ';'

10.4.3 Semantics

The effect of executing a timed statement is the same as executing an action call with an identical timing clause, except that no event is generated.

10.5 Synchronous clocks

10.5.1 Purpose

The purpose of synchronous clocks is to relate the time of a clock to a common metric. By relating two clocks to the same metric, the time of the two clocks become related; we say the clocks are *synchronized*.

10.5.2 Syntax

```
type GST <: Float;

type Synchronous_Clock <: interface
   include Clock;

   function GST      (T : Ticks)        return GST;
   function Distance (T1, T2 : Ticks)   return GST;
   function Ticks    (Distance : GST)   return Ticks;
end Synchronous_Clock;

module Make_Synchronous_Clock () return Synchronous_Clock;
```

10.5.3 Semantics

GST (Global Standard Time) defines a unit of measure that the time of synchronous clocks can be converted to and from. Thus given two synchronous clocks, GST can be used as an intermediate measure for determining the time on one clock given a time on the other clock.

The function GST converts a clock's time to the common metric GST. GST is a one-to-one mapping. For Ticks objects n and m, GST(n) is less than GST(m) if and only if $n < m$.

The Distance function simply returns the absolute value of GST(T1)−GST(T2). Distance(0,t) is the same as GST(t). Distance(n,$n+1$) need not equal Distance($n+1$,$n+2$) for all n.

The Ticks function converts from GST to a clock's time. It returns the time closest to the given GST parameter. That is, if Ticks(g) returns t then GST(t) is closer to g than any other

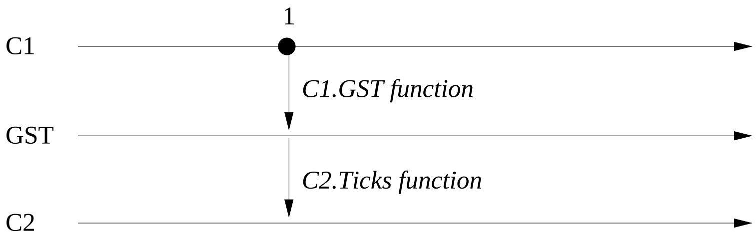


Figure 10.4: Mapping between two synchronous clocks using GST.

clock time. Ticks is a many-to-one mapping. Also, Ticks(GST(t))=t; the Ticks function is the inverse of the GST function.

If the functions GST or Distance are called with times in the future (times greater than Now()), then the exception Timing_Error is propagated. Similarly, Timing_Error is propagated when Ticks(g) is called where GST(Now())$< g$.

Lastly, the value of GST(t) is "determined" when Now() reaches t, and cannot change thereafter. That is, from time t on, GST(t) always returns the same value.

10.5.4 Examples

Example: *Multiple synchronized clocks*

```
module
   action A();
   C1, C2   : Synchronous_Clock is Make_Synchronous_Clock();
   C2_Time : ref(C2.Ticks) is Null(C2.Ticks);
process
   A() pause C1.Ticks'(1);
   await (?E in event()) ?E@A do
      C2_Time := C2.Ticks(C1.GST(C1.Start(?E)));
   end await;
end;
```

Commentary:

An A event is generated at time 1 relative to C1. [12] **C1 could tick before action call.** The GST function of C1 maps the time of the event onto the GST time-line. Then, the Ticks function of C2 maps from GST to the C2 time-line. C2_Time will contain the time, on clock C2, when the A event was generated. Figure 10.4 depicts these conversions.

□

10.6 Regular clocks

10.6.1 Purpose

Regular clocks are synchronous clocks in which the period of the clock, relative to GST, is specified. Further, the accuracy of a regular clock's period is specified.

10.6.2 Syntax

```
type Accuracy is record
   Kind      : enum Interval, Ratio end enum;
   Measure  : Float range 0.0 .. 1.0;
end Accuracy;

type Regular_Clock <: interface
   include Synchronous_Clock;
   Period      : Float;
   Precision  : Accuracy;
end Regular_Clock;

module Make_Regular_Clock (Period      : Float;
                           Precision : Accuracy) return Regular_Clock;
```

10.6.3 Semantics

Period must be greater than zero. Period relates the Ticks time-line of a regular clock to the GST time-line. For example, a period of 1.0 specifies that 1 on the Ticks scale maps to 1.0 on the GST scale, a period of 2.0 specifies that 1 on the Ticks scale maps to 2.0 on the GST scale, and a period of 0.5 specifies that 1 on the Ticks scale maps to 0.5 on the GST scale. Given a totally accurate regular clock R, R.Period$*t$ =R.GST(t) for all R.Ticks t.

Accuracy is defined in one of two ways: as an interval or as a ratio. For an interval accuracy measure a and period of 1, GST(t) $= t \pm (1-a) * t$. More generally, for any unequal Ticks t_1 and t_2, period p, and interval accuracy measure a:

$$a \leq \frac{GST(t_1) - GST(t_2)}{p*(t_1 - t_2)} \leq 2 - a$$

A ratio accuracy measure of a defines the regular clock as:

$$a \leq \frac{GST(t_1) - GST(t_2)}{p*(t_1 - t_2)} \leq 1/a$$

In the case of both interval and ratio accuracy, 1.0 means totally accurate and 0.0 means totally inaccurate.

Figure 10.5 plots various relationships between Ticks of a regular clock and GST. In Figure 10.5(a), accuracy is fixed at 1, and clocks of three different periods p are plotted. In Figure 10.5(b), period is 1 and an interval accuracy a is used. For a regular clock R, the shaded

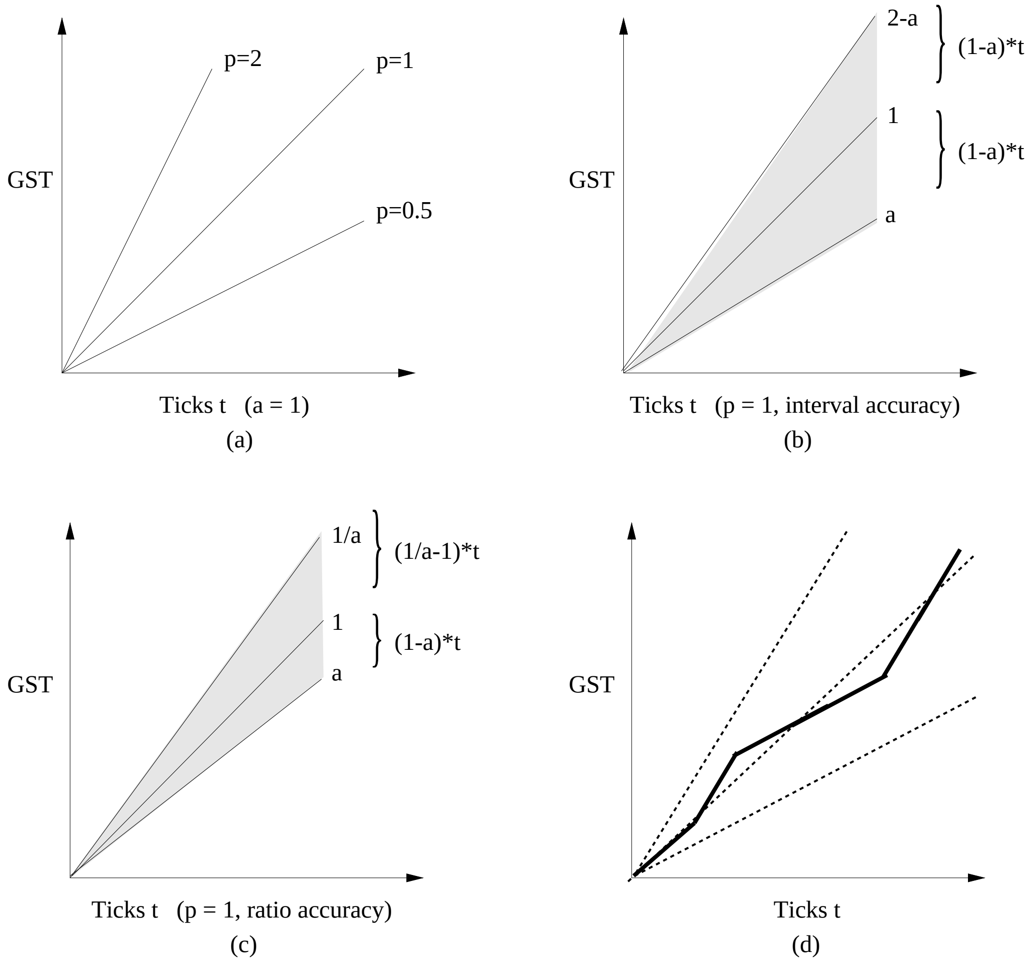


Figure 10.5: Period and accuracy of regular clocks.

area indicates the range of values that may be returned by R.GST(t). The range increases as t increases. The range increases at an inverse proportion to the accuracy, so for lower accuracies, the shaded area would become wider faster. The lines in Figure 10.5(b) are labeled by their slopes. This figure shows that for a given Ticks t and period of 1, the range of GST(t) is symmetric about t.

Figure 10.5(c) shows a ratio accuracy a with a period of 1. Although the shaded area is symmetric about the Ticks=GST diagonal, the range of GST(t) is not symmetric about t: there are more values above t.

Lastly, in Figure 10.5(d) dashed lines show the bounds of GST(t), with the middle dashed line again being accuracy of 1. The thick, solid line indicates one possible GST function given these bounds. It shows that GST/Ticks may vary over time. However, the slope of any segment of the function must remain in the specified bounds.

10.7 Slaved clocks

10.7.1 Purpose

Synchronous_Clock, and its subtype Regular_Clock, relate Ticks directly to GST. A slaved clock relates its Ticks to Ticks of another clock. The other clock is called the master of the slaved clock.

10.7.2 Syntax

```
type Slaved_Clock <: interface
   include Synchronous_Clock;
   Master    : Synchronous_Clock;
   Period    : Float;
   Precision : Accuracy;
end Slaved_Clock;

module Make_Slaved_Clock (Master    : Synchronous_Clock;
                          Period    : Float;
                          Precision : Accuracy) return Slaved_Clock;
```

10.7.3 Semantics

Period again must be greater than zero. A slaved clock is defined in terms of its master. For master M, period p, interval accuracy measure a, and Ticks t_1 and t_2 of a slaved clock S, the slaved clock's GST function is defined as follows:

$$a \leq \frac{M.GST(t_1) - M.GST(t_2)}{p * (S.GST(t_1) - S.GST(t_2))} \leq 2 - a$$

Similarly, for ratio accuracy measure a, GST is defined as follows:

$$a \leq \frac{M.GST(t_1) - M.GST(t_2)}{p * (S.GST(t_1) - S.GST(t_2))} \leq 1/a$$

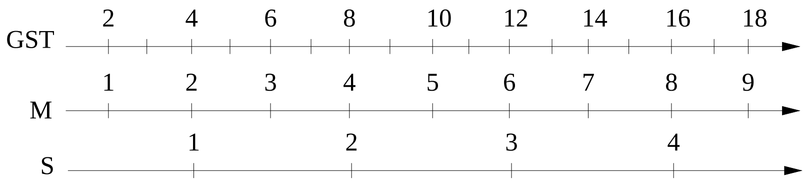
10.7.4 Examples

Example: *A slaved clock and its master*

```
Exact : Accuracy     is (Interval, 1.0);
M     : Regular_Clock is Make_Regular_Clock(2.0, Exact);
S     : Slaved_Clock  is Make_Slaved_Clock(M, 2.0, Exact);
```

Commentary:

M is perfectly accurate, with a period of 2.0 relative to GST. S is a slave of M, and is also perfectly accurate. M is twice as fast as S. The relationships between the GST, M and S time-lines are as follows:



□

Example: *Accumulative error*

```
type Channel (type Message) is interface
    action in Receive(M : Message);
    action out Send (M : Message);
private provides
  C                    : Synchronous_Clock;
constraint
  match ( ?R in event(M : Message), ?S in event(M : Message),
          ?M in Message)
      ( ?R@Receive(?M) -> ?S@Send(?M) where
          C.GST(C.Start(?S)) - C.GST(C.Finish(?R)) < 10.0 )^(~ *);
end Channel;

module M (type Message) return Channel(Message) is
  C     : Regular_Clock is Make_Regular_Clock(1.0, Accuracy'(Kind is Interval, Measure is 0.9));
parallel
  when (?M in Message) Receive(?M) do
    Send(?M) in C.Ticks'(8);
  end when;
end M;
```

Commentary:

The constraint in the parameterized type Channel specifies that for each Receive event, a Send event will be generated. The distance between the Receive and corresponding Send must be less than 10 on the GST scale.

The module generator implements this by generating a Send event 8 C ticks after a Receive event. However, C is only 90% accurate. Suppose a Receive event is observable at every value of C.Ticks (i.e., a Receive event at time 1, a Receive event at time 2, etc.). Figure 10.6 illustrates how the implementation can vary from its specification over time. C.Ticks is plotted on the x-axis and GST on the y-axis. The solid line represents the upper bound of the specification: $t + 10$ for C.Ticks t. The diamonds in the error bounds represent perfect accuracy of the implementation: $t + 8$. At time 12 on clock C, the error bounds reach the constraint. That is, if the above action call is executed at time 12, then a Send event is generated at GST time $12 + 8 \pm (1 - 0.9) * (12 + 8)$, or 20 ± 2. In the worst case, the Send event would be generated at GST time 22: 10.0 GST after the Receive event. In this case the specification is violated by the implementation.

□

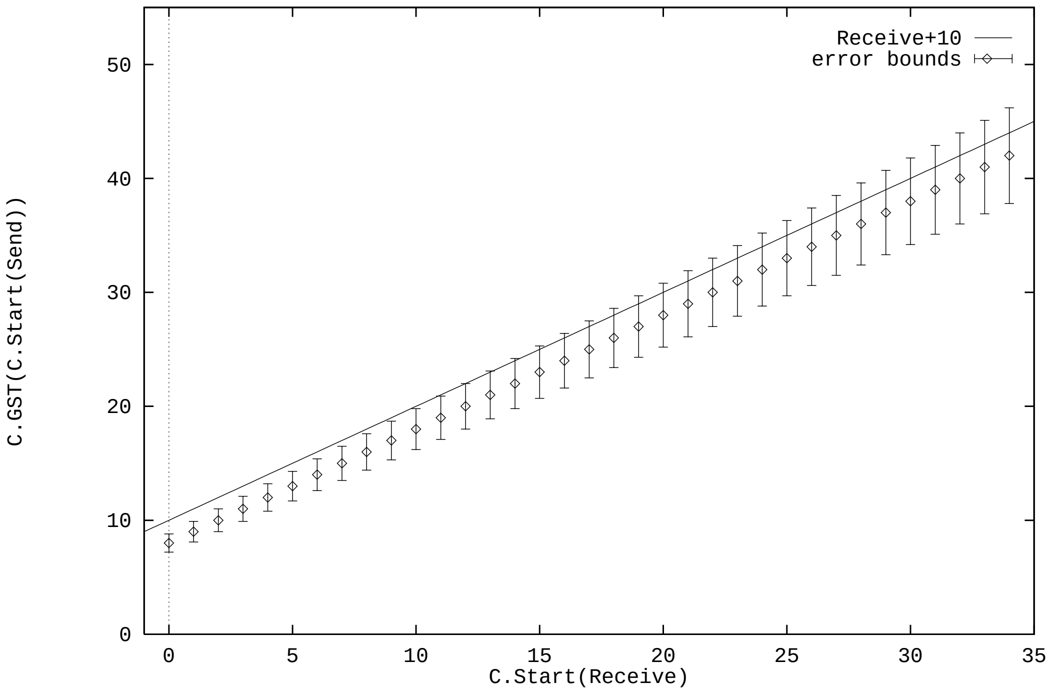


Figure 10.6: Accumulative error in an inaccurate clock.

10.8 Clock Ticking Semantics

For all clocks C, the value of C.Now() is ever increasing. We say that C *ticks* when C.Now() increases. Informally, a clock may tick when there is nothing left to be done at the current time. Specifically, a clock may tick when no subset of the events observable to the clock's parent module are a triggering match of a when-do statement, await-statement, or connection of the parent, and no process of the parent is execution.

Timing clauses and timed statements place constraints on how far a clock may tick. A clock may not tick beyond the next thing to be done. If a timing clause or timed statement being executed names clock C, and evaluates to time t on C, then C may not tick beyond t. Further, if an event has been scheduled at C time t (using a timing clause with **in**), then C may not tick beyond t.

If there is nothing to be done, and nothing scheduled for the future, a clock may tick arbitrarily.

A ramification of these rules is that at the beginning of a computation, a clock may tick arbitrarily. Also, if one process of a clock's parent executes infinitely, the clock never ticks.

10.8.1 Examples

Example: *Multiple Clocks*

```
type T is interface
  action in A();
  action out B();
  action out C();
end T;

M : T is module
    C1, C2 : Clock is Make_Clock();
  parallel
    P1: when A do
      B() pause C1.Ticks'(1);
    end when;
  ||
    P2: when A do
      C() pause C2.Ticks'(1);
    end when;
  end module;
```

Commentary:

Upon creation processes P1 and P2 are waiting for an A event. Since there is nothing to do until an A event, C1 and C2 may tick. Because there are no events scheduled, C1 and C2 may tick an arbitrary amount. Also, since C1 and C2 are not synchronous clocks, they may tick an arbitrary *different* amount; e.g., C1 could tick to 20 and C2 to 30.

Suppose an A event is generated. P1 and P2, in no particular order, match the event and execute the associated action calls. P1 suspends until C1 ticks to C1.Finish(a)+ 1 where a is the triggering event, and P2 until C2.Finish(a) + 1. When the B and C events are generated, there is again nothing left to do. The clocks may again tick an arbitrary, different amount.

□

Example: *Regular Clocks*

```
type T is interface
   action in A();
   action out B();
   action out C();
end T;

M : T is module
      C1 : Clock is Make_Regular_Clock(1.0, Accuracy'(Kind is Interval, Measure is 1.0));
      C2 : Clock is Make_Regular_Clock(2.0, Accuracy'(Kind is Interval, Measure is 1.0));
   parallel
      P1: when A do
         B() pause C1.Ticks'(1);
      end when;
   ||
      P2: when A do
         C() pause C2.Ticks'(1);
      end when;
   end module;
```

Commentary:

In this example, C1 and C2 are totally accurate synchronous clocks. C1 has a GST period of 1, and C2 a GST period of 2. Since there is nothing to do until an A event is generated, C1 and C2 may tick. They may not tick an arbitrary amount since they are related via GST. For example, if C1 ticks 4 then C2 may not tick 4 since this would violate the constraint between GST and regular clocks. Generally speaking, C1 ticks twice as far as C2.

□

Appendix A

Predefined Attributes

This appendix summarizes the definitions appearing elsewhere in the reference manual of the predefined attributes in RAPIDE 1.0.

F'Call	For a function F: An action, events of which represent calls to P. Includes the parameters of the call. See Section 6.4.3.
P'Completed	For a module P: Yields True if P contains no processes, or if the processes of P have completed and P is not terminated; otherwise yields False. See Section 9.9.
F'Return	For a function F: An action, events of which represent the return of a call to P. Includes the parameters of the call and the return value. See Section 6.4.3.
P'Running	For a module P: Yields True if P contains processes that have not completed; otherwise yields False. See Section 9.9.
P'Terminated	For a module P: Yields True if P has propagated an exception; otherwise yields False. See Section 9.9.

Glossary

Overloading!of names: under certain conditions a name may declared more than once in the same declarative region, in which case it is said to be overloaded. See page 3.

await statement A reactive statement including one or more patterns to be matched by available events, and associated alternatives (statement lists) to be executed if a match occurs. See page 58.

block One of several language constructs which may include both declarations and statements. Blocks are formed by do statements, functions, and module constructors. See page 17.

clock A monotonically increasing counter used to schedule and measure events. A module of the predefined type Clock. See page 99.

declaration!associated with a region: a member of a list of declarations specified by the syntax of the declarative region. See page 4.

declaration Associates a name with an entity in the language, and causes the associated entity to be elaborated. The name is said to denote the entity, or be bound to the entity. See page 13.

direct visibility: A declaration is directly visible within its immediate scope unless it is *hidden* in an inner declarative region by a local declaration. See page 8.

elaboration Elaboration causes the entity to become available in the computation. See page 13.

immediate scope: of a declaration starts at the end of the declaration in the declaration list, and extends to the end of the associated declarative region. See page 6.

indirect visibility: A visible declaration of an interface type expression is indirectly visible in the extended part of its scope. See page 8.

object declaration A declaration which associates a basic name with an object of a given type. See page 13.

process An independently executing thread of control. See page 55.

qualification!of a name: notation whereby a hidden declaration of a of a name may be referred to. See page 5.

reactive statement A statement used to respond to events available to a process. See page 57.

scope name: the name of the declarative region associated with a declaration is called the scope name of the declaration. See page 7.

scope, of a declaration: region of program text within which the declaration is visible. See page 6.

scope, of a name: the scope of its declaration. See page 6.

static visibility: compiletime visibility of names defined in terms of declarative regions. See page 3.

tool Something that enforces language rules, or generates computations from programs. See page 83.

triggering match A match of a pattern that triggers the execution of an await statement alternative. See page 58.

when statement A reactive statement; form of a do statement. See page 61.

Bibliography

[Tea94a] Rapide Design Team. *The Rapide-1 Architectures Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94b] Rapide Design Team. *The Rapide-1 Executable Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94c] Rapide Design Team. *The Rapide-1 Pattern Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94d] Rapide Design Team. *The Rapide-1 Predefined Types Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94e] Rapide Design Team. *The Rapide-1 Specification Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea94f] Rapide Design Team. *The Rapide-1 Types Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

[Tea95] Rapide Design Team. *The Rapide-1 Full Syntax Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, August 1995.

Index