DRAFT
Rapide 1.0 Architecture 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 Reference Manual

This manual assumes the RAPIDE overview given in the Guide to Rapide, and refers to other LRMs, particularly the Types LRM. It describes the architecture constructs of RAPIDE-1.0. These consist of *(i)* extensions of interface types (Types LRM) to allow structuring into sub-interfaces (called *services*), and to include executable *behaviors* in interfaces, and *(ii)* architectural connections. Two experimental constructs related to architectures are also described here: *bindings* and *maps*. Bindings allow different interfaces to be bound to the same module dynamically while it is executing. Maps define relationships between architectures that permit the behavior of one architecture to be interpreted as (or "mapped" into) the behavior of another architecture at runtime. Both constructs have applications in testing for consistency and conformance.

This manual also explains the dynamic communication model of RAPIDE. This defines the ways in which modules can communicate (i.e., synchronize or pass data) at runtime by means of events and function calls. It gives a unified view of communication involving the use of both communication constructs defined in the Executable LRM and here in the Architectures LRM.

The reader can omit learning about constraints, bindings and maps until the need for them arises. The full range of control structures in the Executable language is only needed when one develops implementations of an architecture (i.e., instances) by associating modules with their component interfaces.

The core constructs of RAPIDE architectures are interfaces and connections. Maps and Bindings are experimental constructs.

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

Contents

1 Overview **1**

1.1 Terminology 2

1.2 Notational Conventions 2

1.3 Visibility 3

2 Interfaces **5**

2.1 Syntax 5

2.2 Semantics 7

2.3 Object Name Constituents 7

2.4 Actions 7

2.4.1 Purpose 8

2.4.2 Syntax 8

2.4.3 Semantics of Actions 8

2.5 Services 9

2.5.1 Purpose 9

2.5.2 Syntax 10

2.5.3 Semantics of services 10

2.6 Behaviors and Transition Rules 12

2.6.1 Purpose 13

2.6.2 Syntax 13

2.6.3 Terminology 13

2.6.4 Static Semantics 14

2.6.5 Semantics of Behaviors 14

2.7 Conformance to Behaviors. 16

2.8 Constraints 17

2.9 Conformance of Behaviors to Interfaces 17

2.10 Examples 17

3 Architectures **23**

3.1 Syntax 23
3.2 Static Semantics 24
3.3 Semantics 25
3.4 Connections 25
3.4.1 Purpose 26
3.4.2 Syntax of connections 26
3.4.3 Static Semantics of Connections 28
3.5 Semantics of Connections 29
3.5.1 Semantics of Object Connections 30
3.5.2 Semantics of Service Connections 30
3.5.3 Semantics of Event Pattern Connections 30
3.5.4 Basic pattern connections 31
3.5.5 Pipe connections 32
3.5.6 Agent connections 32
3.5.7 Connection generators 32
3.6 Constraints 34
3.6.1 Syntax 34
3.6.2 Semantics 34
3.7 Type conformance of architectures 34
3.8 Examples 35

4 Communication Model **43**

4.1 Initiating Communication 43
4.2 Receiving communication 44
4.2.1 Context 44
4.3 Observing Events and Function Calls 44
4.4 Examples 45

5 Maps **49**

5.1 Syntax 49
5.2 Semantics 51
5.2.1 Visibility within map generators 52
5.2.2 Events Observed by a Map 52
5.2.3 Semantics of mapping rules 53
5.2.4 Conformance to Range Constraints 54
5.3 Map Declarations 55

5.3.1 Syntax 55
5.3.2 Semantics 55
5.4 Uses of Maps 55
5.5 Examples 55

6 Bindings **61**
6.1 Syntax 61
6.2 Semantics 62
6.3 Examples 63

Glossary **67**

Index **70**

List of Figures

3.1 Direct connection architecture. 37
3.2 Fan-in connection architecture. 38
3.3 Fan-out connection architecture. 39

List of Tables

Chapter 1

Overview

The constructs of RAPIDE-1.0 for defining architectures are described in this document. An architecture is a module containing a set of modules (objects of interface types), a set of *connection rules* that define communication between the interfaces of the objects, and a set of *constraints*. An architecture is a restriction of the form of module defined in RAPIDE 1.0 Executable Language Reference Manual since it may contain only some of the features allowed in modules. An *interface connection architecture* as described in Guide to Rapide is an architecture in which all the objects are interface objects.

Interface types are described in the RAPIDE 1.0 Types Language. Chapter 2 describes the additional features of interface types introduced for architecture construction, namely, *services* and *behaviors*, and also describes asynchronous *actions*.

Chapter 3 describes the RAPIDE-1.0 *architecture* construct which is a restricted form of the executable language *module* construct. The only constituents of architectures are *declarations*, *connections* and *constraints*. Components or their types are declared in the declarations part. Connections are event pattern connections (also called *connection rules*) for defining communication between components. Connections can also create new components. Architectures are essentially dynamic in several respects: connections can depend upon runtime conditions, the number of component objects can vary, and a connection can be defined between arbitrary numbers of components, by connecting any objects of given types.

Instances of an architecture are constructed by associating modules (which may be either architectures or executable language modules) with (some of) its component interfaces. This is a basic technique for hierarchical architecture refinement and evolution.

Chapter 4 describes the communication model for RAPIDE. This defines which modules can communicate at runtime, and when communication can happen. Communication is either by means of events or by function calls. The communication model gives a unified view of the constructs for communication that are defined in Architectures (Chapter 3) and in the Executable LRM.

Chapter 5 describes the *map* construct for defining relationships between architectures. Chapter 6 describes the *binding* construct for relating architectures to interfaces. Both of these constructs use the same underlying event pattern construct that are used for connections in architectures. Maps and bindings are considered to be experimental constructs for supporting hierachical design methodologies and for runtime comparison of architectures.

The architecture language, as a whole, is a small extension of the RAPIDE type language. The new architecture constructs rely heavily on event patterns (RAPIDE 1.0 Event Pattern Language Reference Manual), e.g., event patterns are used in behavior rules, connections in architectures, and in bindings and maps. The reader may start building executable architectures with a knowledge of interfaces and architectures defined in this manual, together with some knowledge of the Types and Pattern languages, the standard types, and declarations of objects as defined in the executable language.

Each chapter provides syntax, semantics and examples. In particular, syntax descriptions indicate by means of the italic nonterminal notation (below) where productions of other RAPIDE framework languages are included.

1.1 Terminology

This section contains terminology used in more than one chapter. Other terminology is defined in each chapter, and summarized in the Glossary.

module: an object resulting from a call to a module generator; thus an object of an interface type.

interface object: an object resulting from the elaboration of a name declaration where the type is an interface type (see Section 2.6.5).

object: a module (including an architecture, or binding), an interface object, or a function.

architecture: a module resulting from a call to an architecture generator.

component, of an architecture: an interface object or a module belonging to the *context* (see Chapter 4) of the architecture. These include, for example, *(i)* an object of an interface type denoted by an object name that is declared in the declaration part of an architecture, or *(ii)* an object of an interface type that is declared in the declaration part of an architecture and the object is created by executing a creation rule of the architecture, *(iii)* an object which is linked to the architecture during execution.

instance, of an architecture: a module that differs from an architecture only in that an interface object that is a component of the architecture has been given a module denotation in the instance.

parent, of a component: the architecture that declares the component, or declares its type and creates the component.

observer, of a module: (1) the module itself, (2) the parent module, (3) any module which is *linked* to the module in question (see Chapter 4).

1.2 Notational Conventions

The syntax in all RAPIDE manuals is given as a series of productions:

```
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
`rhs_1 | ... | rhs_n`	choice

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

The syntax in this manual includes productions of the RAPIDE 1.0 Types Language, productions of the RAPIDE 1.0 Executable Language for expressions and standard types, productions of the RAPIDE-1.0 pattern language, and the productions of the RAPIDE 1.0 Constraint Language. Most of these productions are elided by italic non-terminals indicating their inclusion. Whenever a non-terminal appears in the grammars for two of the framework languages, the productions defining its syntax in one language must be unambiguous with the productions in the other language.

For example, the syntax for interface constituents contains:

```
interface_declarative_item ::=
        ...
|       types_language_type_declaration
|       types_language_type_constructor_declaration
|       name_declaration
        interface_derivation_declaration
```

The italic nonterminal indicates that the RAPIDE 1.0 Executable Language productions for object declarations are included. A roman nonterminal indicates that productions for that nonterminal are given in this manual. These productions must be unambiguous with productions for the same nonterminal in other framework languages, if any. For example, object_name_declaration, indicates that productions for object name declaration are given in this manual. These productions must be unambiguous with productions for the same nonterminal in the RAPIDE 1.0 Types Language. The roman nonterminal, service_declaration, introduces a new nonterminal not occuring in other framework languages.

1.3 Visibility

Scopes and their use to define the static visibility rules for expressions are defined in the RAPIDE 1.0 Executable Language Reference Manual, Chapter 2.

Chapter 2

Interfaces

Interface types are described in the RAPIDE 1.0 Types Language. Additional features of interface types are introduced for constructing architecture, namely, *behaviors, services* and *constraints*. This chapter describes behaviors and services.

An interface type defines a type of object called a *module*. An interface type is defined by an *interface type expression*, which is often called, simply, an *interface*. The modules of an interface type are often said to *have* or *possess* that interface.

The *constituents of an interface* described in RAPIDE 1.0 Types Language Reference Manual are the names of types, objects and actions. These interface constituents are grouped into provides, requires, action, and private interface declarative regions, and are defined by name declarations that give the name and denotation specification of a constituent. The constituents of an interface described in this chapter are constituents of behaviors and services parts. An interface may declare behaviors that model the states and state transitions of modules of that interface type. Behaviors are executable constraints — all modules of the interface type must behave in a way that conforms with the interface behavior (Section 3.7). An interface may also declare constraints w3hose purpose is to specify restrictions on its constituents. Constraints are expressed in the RAPIDE 1.0 Constraint Language. In general they are declarative and are not executable.

The *constituents of a module* are the *types* and *actions* declared in the module, and the *objects* (including *functions*) that are either declared or created in the module. A constituent of a module whose name is declared in its interface is called a *visible constituent*. A constituent of a module whose name is not declared in the module's interface is *hidden* (sometimes also called *internal*). A constituent that is declared in the interface as being required by the module is a *required constituent*.

Each module must *conform* to its interface type. The module must contain the constituents that the interface specifies that it provides, and it may refer to the constituents that the interface specifies that it requires, and assume that those references will be connected to appropriate constituents of other modules by the architecture (see Chapter 3). Conformance of modules is defined in the RAPIDE 1.0 Executable Language Reference Manual; conformance of architectures is defined in Chapter 3.

2.1 Syntax

The following is the syntax of the architecture language interface declarative regions. These productions extend the syntax productions for interface type expressions given in the RAPIDE

1.0 Types Language. An interface type expresson (or interface) may contain the declarative regions defined in the RAPIDE 1.0 Types Language and the regions defined here. Some of the productions in the RAPIDE 1.0 Types Language Reference Manual are duplicated here because the architecture language allows alternate syntax sugar, or because the semantics of the constructs are defined here.

```
architecture_language_interface_declarative_region ::=
|       interface_services_part
|       interface_behavior_part
|       interface_constraint_part
```

```
interface_services_part ::=
        service { service_declarative_item }
```

```
interface_behavior_part ::=
        behavior { behavior_declarative_item } begin { transition_rule }
```

```
interface_constraint_part ::=
        constraint { constraint_language_constraint }
```

```
behavior_declarative_item ::=
|       type_language_type_declaration
|       type_language_type_name_declaration
|       type_language_type_constructor_declaration
|       type_language_type_constructor_name_declaration
|       type_language_action_declaration
|       executable_language_object_declaration
|       object_name_declaration
|       map_declaration
|       type_language_interface_derivation_declaration
```

```
object_name_declaration ::=
        basic_name_list ':'  type_expression ';'
|       function_name_declaration
```

```
function_name_declaration ::=
        function identifier '(' [ formal_parameter_list ] ')' [ return type_expression ]
              [ constraint constraint_language_function_constraint ] ';'
```

Object name declarations allow alternate syntax for function name declarations to that allowed in the RAPIDE 1.0 Types Language.

2.2 Semantics

An interface consists of a sequence of regions and an optional **private** interface. Each region may be a *provides, requires, action, service, behavior* or *constraint* part.

A *service* part declares services that specify constituents of the interface in the sense explained in section 2.5.

A *behavior* part consists of a set of types, objects, and transition rules. The objects model abstract *state*, and the transition rules specify how modules of the interface type react to patterns of observed events by changing their states and generating events. A behavior part is executed as described in section 2.6 to provide a behavior for an object of the interface type whenever no module is associated with that object. A behavior part also defines a constraint on any module of the type, as described in Sections 2.6, 3.7.

A *constraint* part of an interface is a set of constraints on the behavior of modules of the interface type, possibly extending any constraint defined in a behavior part. The set of constraints in a constraint part do not necessarily define a unique execution; their purpose is to bound the set of allowable behaviors.

The interface constituents of a module are visible to its observers, except the private constituents which are only visible to observers that are of the same type (or a subtype of the type).

The semantics of interface constituents are described in the following sections.

2.3 Object Name Constituents

The denotation of a **provides** object constituent named in an interface of an interface object is the object of the same name declared in a behavior of that interface object. The denotation of a **provides** object constituent named in an interface of a module is the object of the same name declared in that module. The denotation of a **requires** object constituent named in an interface is determined by the connection rules of the parent architecture of the object having that interface (see Section 3.4).

A call to a **provides** or a **requires** function in an interface may result in generation of 'call and 'return events upon the start and termination of the call. That is, when an expression f(*params*) is evaluated, an event whose Actn component is f'call, (see section 2.4), whose Pi components are *params*, and whose Actor and Target components are the caller and callee objects is generated as an **in** event of the callee. Similarly, an 'return event is generated when the call terminates. Such events are treated as if they were generated by calls to actions of the interface.

Commentary

> The semantics of interface functions does not require that 'call and 'return events be generated.

□

2.4 Actions

Action name declarations (called *actions* for short) may appear in an action interface of an interface type expression (see RAPIDE 1.0 Types Language Reference Manual).

2.4.1 Purpose

An action in an interface type expression indicates that modules of that interface type have the ability to observe and generate events of particular event types. Events are generated by calls to actions and become observable through the process of *matching* (see below). Event types are defined in RAPIDE 1.0 Predefined Types Reference Manual.

2.4.2 Syntax

```
action_part ::=
        action { action_declaration }


action_declaration ::=
        in { action_name_declaration } ';'
|       out { action_name_declaration } ';'


action_name_declaration ::=
        basic_name_list '(' [ formal_parameter_list ] ')' ';'


action_call ::=
        action_name '(' [ actual_parameter_list ] ')' ';'
```

2.4.3 Semantics of Actions

An action name declaration,

action A (Pi: Ti);

(for i = 1,..., n), denotes an *event type* which is the type of events generated by calls to the action:

```
type E is interface
            Pi: Ti;
            Actor  : root;
            Owner  : root;
            Actn   : String;
        end interface;
```

Root denotes the metatype that is the supertype of all interface types. The Actor of an event identifies the module whose execution of an action call generated the event. The Owner of an event identifies the module whose action was called to generate the event. The Actn is the name of the action that was called. The components (Pi: Ti) of the event are the actual parameters of the action call that generated the event.

Events generated by calls of an action are called *events of that action.* Each action call results in a unique event which can be distinguished from any other event by the =, / = operations of the event type (RAPIDE 1.0 Predefined Types Reference Manual).

An action may be declared in an *action* part of an interface, or in a module. It is called, respectively, an **in**, **out**, or *internal* action of a module. Events of that action are called **in, out**, or *internal* events.

The modifiers, **in, out** indicate which modules can generate events of the action and which modules can receive those events.

An **in** action declaration,

action in A (Pi: Ti);

with formal parameters for i = 1, ..., n, declares an **in** action A of modules of that interface type. If M is a module with that interface, its **in** action A may be called by any observer of M (e.g., its parent architecture) and by M itself. The result of the call is an event which is observed by M.

An **out** action declaration,

action out A (Pi: Ti);

with formal parameters for i = 1, ..., n, declares an **out** action A of modules of that interface type. If M is a module with that interface, it may call its **out** action A. The result of the call is an event which can be observed by any observer of M (e.g., its parent architecture) and by M itself.

Action calls have the same syntax as indexed expressions. A call to action A of module M by an observer uses the qualified name syntax, `M.A(v1, ..., vn)`; a call by M itself may use the syntax `A(v1, ..., vn)` since the name A is directly visible in M, being a constituent of the interface of M; in either case, `v1, ..., vn` must be a type correct list of actual parameters.

The only effect of an action call is the generation of an event of the action. The call does not imply that the caller blocks or awaits any result.

The event resulting from an **in** action call is *visible* in the module having the action. The event resulting from an **out** action call is visible to the observers of the module (which include the module having the action).

Commentary

Architectures can wire up components using their interface actions.

An **in** action of a component allows a parent architecture to generate events of that action that the component can observe. The component can also itself generate events of the action (which it can observe).

An **out** action of a component indicates that the component can generate events of the action which a parent architecture can observe. The parent architecture can connect an *out* action to **in** actions of other components by generating those **in** events whenever it observes the **out** events. Connections define a communication structure between the components (see connections, 3.4).

□

2.5 Services

Service declarations are also called *services*. Services are a syntactic feature for structuring interfaces.

2.5.1 Purpose

The purpose of services is twofold: *(i)* to encapsulate a related set of constituents of the enclosing interface, *(ii)* to allow large numbers of connections between components to be defined by a single connection rule in a parent architecture (Chapter 3).

2.5.2 Syntax

```
service_declarative_item ::=
        service_declaration
        service_set_declaration
```

```
service_declaration ::=
        basic_name_list ':'  [ dual ] interface_type_expression ';'
```

```
service_set_declaration ::=
        basic_name_list '(' { service_set_index } ')' ':'  [ dual ]
                                              interface_type_expression ';'
```

```
service_set_index ::=
```
executable_language_range
executable_language_enumeration_type_expression

A service declares constituents of the enclosing interface. The usual dot notation may be used to refer to constituents declared in a service: i.e., if I contains a service S of type T, and T has a constituent C, then S.C is a constituent of I. The kind of constituent, e.g., **provides** or **requires** object, **in** or **out** action, is determined by the rewrite rules below.

A service set declaration declares a set of services of the enclosing interface each of which has the type of the interface type expression (or its dual). Each service of the set can be named by indexing the service set name with a literal of the range or enumeration type used as the service set index (more generally, the index may be a tuple of literals). E.g., if I contains service set S(a .. b) where a, $a+1$, ..., b is an integer range, then S(a), S(a +1) are names of services in I.

2.5.3 Semantics of services

Services are constituents of the service parts of an interface. Services are declared to be of an interface type or of the **dual** of an interface type (explained below).

The interface type specification in a service declaration may not contain a private part, and it may not contain type name or type constructor name constituents.

An interface containing a service is equivalent to an interface in which the service is replaced by its constituent declarations according to a set of rewrite rules. The rewrite rules qualify the names of individual constituents of the type of the service with the name of the service, and place those constituent declarations in appropriate provides, requires or action parts (possibly new parts) of the enclosing interface.

Modules of an interface type containing services must conform to the expanded interface after the service rewrite rules have been applied. Consequently a conforming module need not contain objects corresponding to its service declarations. It must provide the constituents defined by its services as being provided, and may assume those constituents defined by its services as being required.

If interface I contains a service S of a type T, or a service R of the **dual** of type T, then I may be rewritten as an interface, I', without the services S and R:

rewriting services containing object constituents:
If type T has a **provides** object F, and I contains a service S of type T, then I' has a **provides** object $\mathsf{S.F}$

If type T has a **requires** object G, and I contains a service S of type T,then I' has a **requires** object $\mathsf{S.G}$.

If type T has a **provides** object F and I contains a service R of the dual of type T, then I' has a **requires** object $\mathsf{R.F}$.

If type T has a **requires** object G and I contains a service R of the dual of type T, then I' has a **provides** object $\mathsf{R.G}$.

rewriting services containing action constituents:
If type T has an **in** action A and I contains a service S of type T, then I' has an **in** action $\mathsf{S.A}$.

If type T has an **out** action B and I contains a service S of type T, then I' has an **out** action $\mathsf{S.B}$.

If type T has an **in** action A and I contains a service R of the dual of type T, then I' has an **out** action $\mathsf{R.A}$.

If type T has an **out** action B and I contains a service R of the dual of type T, then I' has an **in** action $\mathsf{R.B}$.

In each rewrite case above, a behavior or constraint in T is replaced by a new constraint of I' in the constraint interface of I'. Names in the new constraint are obtained by replacing the old names in the old behavior or constraint by the new names. A behavior is replaced by a new constraint that constrains the executions of I' to satsify the behavior as defined in Section 3.7.

rewriting services containing service constituents:
If type T has a service Q of type T' (which may be a dual of another type) and I contains a service S of type T, then I' has a service $\mathsf{S.Q}$ of type T'.

If type T has a service Q of type T' (which may be a dual of another type) and I contains a service S of a dual of type T, then I' has a service $\mathsf{S.Q}$ of the dual of type T'.
The dual of the dual of a type is the type.

By recursive application of the rewrite rules, an interface may be rewritten as an equivalent interface without services.

Commentary

> An implication of the rewrite semantics is that services are a notational mechanism for structuring interfaces. They are not expressions that denote objects although they have similar syntax.
>
> Note that the rewrite rules for objects apply to functions. Also, services do not contain type name or type constructor name declarations in their provides or requires parts.
>
> A service and its dual have the same behaviour and constraint sections. The semantics of constraints in a service is the same as the semantics of constraints in an interface. A behavior in a service acts as a constraint on the constituents of the service in manner defined in Section 3.7.

□

Example

The declaration

```
type I is interface
        service P: interface
                        P1: T1;
                    requires
                        E2: T2;
                    end interface;;

                Q: dual interface
                            P3: T3;
                        requires
                            E4: T4;
                        end interface;
        end interface;
```

is rewritten to (and understood to represent):

```
type I is interface
            P.P1: T1;
            Q.E4: T4;
        requires
            P.E2: T2;
            Q.P3: T3;
        end interface;
```

Commentary

A service of a type T may be thought of as analogous to a *plug* of some standard type, and a service of the dual of T is analagous to a *socket* of that type. One of the powerful features of services is that they may be connected by a single connection rule (chapter 3). Such a connection is equivalent to a set of connections between each pair of dual constituents with the same name, one in each of the services (see Section 3.5.2).

Services may be viewed as introducing *context* into interfaces because an interface with a service (or plug) "expects" to be connected to other interfaces that have the dual service (or socket).

Services also allow communication between modules to be defined with the same object-oriented notions and non-flat namespaces that are used in defining modules. This is achieved by building up the types of services (plugs/sockets) for a given domain using object-oriented methods.

□

2.6 Behaviors and Transition Rules

The *behavior* part of an interface consists of a set of types, objects, and transition rules. The objects model abstract *state*, and the transition rules specify how modules of the interface type react to patterns of observed events by changing their states and generating events.

2.6.1 Purpose

A behavior part is executed to provide a behavior for an object of the interface type whenever the object is declared without a denotation (i.e., no module is associated with that object). When an object of the interface type is given a denotation, a behavior part in the interface defines a constraint to which that denotation must conform as described below.

2.6.2 Syntax

```
behavior_part ::=
        behavior { behavior_declarative_item } begin { state_transition_rule }


state_transition_rule ::=
        [{ pattern_language_placeholder_declaration} ] transition_rule ';'


transition_rule ::=
        {trigger} op [ { state_assignment } ] [ poset_generator ] ';'


trigger ::=
        pattern_language_pattern
|       boolean_expression


op ::=
        '=>' | '||>'


state_assignment ::=
        executable_language_statement ';'


poset_generator ::=
        pattern_language_restricted_pattern ';'
```

2.6.3 Terminology

A state transition rule consists of an optional sequence of placeholder declarations (RAPIDE 1.0 Event Pattern Language Reference Manual) called the *outermost placeholder declarations*, followed by a transition rule.

A transition rule has two parts, a *trigger* and a *body*.

A trigger is either an event pattern (see RAPIDE 1.0 Event Pattern Language) or a boolean-valued expression on the left side of the op symbol (either => or ||>) of a transition rule.

A *body* is the part of the rule on the right of the op symbol (=> or ||>). It consists of an optional set of statements of the RAPIDE 1.0 Executable Language followed by an optional *restricted pattern* which describes a set of posets (called a *poset generator*). .

A transition rule defined using the $\Rightarrow$ symbol is called a *pipe*, one defined using the ||> symbol is called an *agent*.

2.6.4 Static Semantics

Violations of restrictions stated below result in compiletime errors.

Names declared in **provides** or **requires** or **private** parts of the interface are visible in a behavior part; names declared in other behavior parts are not.

The name **self** in a behavior part denotes the object of the interface type that results from the elaboration of an object name declaration that in turn causes the elaboration of the behavior.

The scope of outermost placeholder declarations at the left of a state transition rule extends to the end of the rule. Placeholders in the body of a state transition rule must be declared in the outermost declarations of the rule and must appear in patterns in the trigger in such a way that they are guaranteed to be bound by any match of the trigger. Placeholders declared locally in the trigger of a transition rule have a scope within the trigger determined by the pattern language rules (RAPIDE 1.0 Event Pattern Language Reference Manual).

The action names of basic event patterns in a trigger must be *(i)* names of actions declared in the interface (**in**, **out** or **private** actions), or *(ii)* names of **out** actions of objects declared in the behavior part containing the rule, or *(iii)* names of the form F'call or F'return where F is the name of a function declared in the interface or the name of a function required by an object declared in the behavior part containing the rule.

The action names of basic event patterns in a *poset generator* must be (**in**, **out** or **private**) actions of the interface or **in** actions of objects declared in the behavior part. The *actor* part of the basic event patterns (see section 2.4) in the *poset generator* must be **self**, or else it must be omitted (in which case **self** is the default actor). Furthermore, patterns in a poset generator are *restricted patterns*: they may not contain the $\Leftrightarrow$ operator.

Commentary

> As a consequence of the restrictions on visibility of names and actions in separate behavior parts, behavior parts cannot share abstract state or actions of their local objects.

□

2.6.5 Semantics of Behaviors

If an interface type is the type in an object name declaration then a denotation for the object is created from the behavior part of the interface. [1] That denotation is an object having the execution behavior defined below. In effect, an implicitly declared module generator is called. The call executes sequentially as follows:

1. Create a new module.
2. Generate the predefined Start action of the module.
3. Elaborate the declarations of the behavior part.
4. Elaborate the transition rules of the behavior part.
5. Return the new module.

[1] A name declaration does not assign a denotation for the declared object.

The denotation of an object name in the **provides** interface of an interface object the object of the same name declared in the behavior, if any (see Section 2.9).

The elaboration of a transition rule makes it ready for matching and execution.

The execution semantics of behaviors is as follows.

Boolean triggers. First, if any of the boolean triggers are true, then one of the transition rules with a true trigger is chosen arbitrarily and its body is executed. This process is repeated until none of the boolean triggers is true, at which time the pattern triggered transitions are processed as described below.

Event observation. Whenever an action of an interface object is called the resulting event (see section 2.4) is received by the object and becomes observable by its transition rules.

Next, the execution selects one of the events it has received. An event is selected at most once. If no events are available for selection, the execution waits for one. Events are selected in a sequential order, one by one, which is consistent with their temporal and causal orderings. The execution attempts to *match* the triggers of the rules, according to the semantics of matching given in the RAPIDE 1.0 Event Pattern Language Reference Manual. An event may take part in triggering a given rule at most once, although it may take part in triggering several rules.

The selected event may *trigger* one or more transition rules. A rule R triggers if some subset of the events thus far selected by the module but not yet used in triggering R match its pattern. If more than one set of selected events can trigger a rule, an earliest (in time and causal order) and largest is chosen according to the definition given in the RAPIDE 1.0 Executable Language Reference Manual. If R triggers, the binding of placeholders used to match its pattern trigger is applied to its body. The resulting instance of R's body is then ready for execution.

The set of rule body instances that are ready for execution are then executed one by one in some order.

Guarded pattern triggers are matched according to the RAPIDE 1.0 Event Pattern Language Reference Manual, Section 2.5. Namely, the guards in the pattern triggers of a behavior are evaluated whenever an interface event is received. Their Boolean values are associated with the event. In matching a guarded pattern, first the pattern is matched with the selected events and then the guard is evaluated. The value of the guard that is associated with the *last* selected event used in matching the pattern is taken as the value of the guard when the pattern is matched. If a guard has an unbound placeholder when it is evaluated, a runtime error results, except in the case that the predicate of the guard is an = operator; in this case the placeholder is assigned the object on the other side of the = operator, if there is one.

Transition rule execution. Executing an instance of a rule body consists of *(i)* executing the statements of the body, *(ii)* then generating a poset of new events, and *(iii)* adding the new poset to the execution.

Poset generation. The new poset must match the instance of the *poset generator*. In general the generated poset will be a partially ordered set of events conforming to the causal relationships defined by the poset generator. The denotation of **self** is the actor of all the new events. If the instance of the poset generator can match more than one new poset, then one of them is chosen arbitrarily.

Dependency order of generated events. When the new events are added to the already existing events, their dependencies on existing events are as follows. They depend on all of the events in the match of the trigger. They also depend on events as a result of assignment and dereference operations in executing the rule (see RAPIDE 1.0 Predefined Types Reference Manual chapter 16). For example, new events may depend on events that triggered the last rules to assign to objects that are referenced in executing the rule (e.g., in a guard in the trigger or in a parameter expression on the right side of the rule). In addition, if the rule is a *pipe*, then all of the generated events depend on all events generated by any previous triggering of the rule. If the rule is an *agent* rule, events generated on distinct triggerings of the rule are independent unless otherwise constrained to be dependent (e.g., if events generated on one triggering contribute to another triggering).

Execution of the behavior then continues with evaluating boolean triggers first, and then selecting events, as above.

Below is a schema for the above ordering of activities:

```
loop
   while (one or more boolean triggers are true) do
       choose one of the true triggers and
          execute its rule's body;
   end while;
   select an event E;
   for all pattern triggers triggered by E loop
      execute the trigger's rule's body
   end for;
end loop;
```

The execution of the behavior of an object *terminates* whenever the execution is waiting to select an event (i.e., there are no more events for selection), all objects declared by the behavior have terminated, and all observers of the object have terminated.

Commentary

> The behavior feature is designed to allow interfaces to define a high-level abstract behavior for modules of their type. A single transition rule can express the reaction of a concurrent/distributed system to an input since the rule's output pattern can define a complex poset of events with various dependencies and timing. Transition rules thus provide a simple high level reactive programming feature for defining behaviors in terms of events and their dependencies.

□

Commentary

> As a consequence of event observation, transition rules execute one at a time rather than concurrently. The task of programming rules that share state is thereby simplified. However, triggering of rules is nondeterministic since independent events may be selected in any order. Also, the order in which transitions are executed when more than one trigger is matched by the same poset is nondeterministic. Thus, the semantics of interface behaviors is may be viewed as nondeterministic finite state automata computing on posets.

□

2.7 Conformance to Behaviors.

When an object of an interface type is declared by an object declaration, in which case the object is given a module denotation, the behavior part of the interface is not elaborated to an executable form.

A module of an interface type is constrained by the behavior in that interface. Its own executions must be consistent with the interface behavior (see Section 3.7).

2.8 Constraints

Constraints in an interface specify restrictions on the behavior of modules of that interface type. That is, the poset behavior that results from executing either the interface transition rules or modules of the type, must satisfy the constraint (see RAPIDE 1.0 Constraint Language Reference Manual).

Commentary

> A constraint can specify, for example, that transition rules trigger only in certain orders, in certain states, etc. Constraints may be supported in different ways by various tools. For example, executions can be checked and violations detected (runtime checking or post-mortem tools), or transitions may be allowed to execute only if a constraint won't be violated (safe execution), or transition rules can be proved consistent with constraints (verification).

□

2.9 Conformance of Behaviors to Interfaces

A behavior part must conform with other parts of the interface as described in Section 2.6.

A behavior part of an interface should, but is not required to, conform with the **provides** and **constraint** parts as follows. For each provides type name declaration it should contain a definition of that type. For each provides function name declaration it should contain a function (see RAPIDE 1.0 Executable Language Reference Manual) with that name satisfying that denotation specification (e.g., is a subtype of the function type specified in the name declaration). For each provides module name declaration it should contain a module (see RAPIDE 1.0 Executable Language Reference Manual) with that name satisfying that denotation specification (e.g., is a subtype of the interface type specified in the name declaration). Finally, the state transition rules in the behavior part should generate poset executions that do not violate the constraints (see RAPIDE 1.0 Constraint Language Reference Manual).

Commentary

> Conformance requirements for interface behaviors are intentionally weak. Behaviors are intended for rapid prototyping and are allowed to be incomplete in many respects. A provided function name, for example, may be omitted or stubbed. Warnings may be issued by a compiler or constraint checker or other tools if these criteria are not met.
>
> Testing for inconsistency between behaviors and constraints is expected to play an important role in developing interfaces for complex modules. It is expected that these conformance requirements for behaviors will be changed as a result of experience with RAPIDE tools and prototyping processes in the future.

□

2.10 Examples

Example: *Automobile controls*

```
type AutoControls is
interface
    function Speedometer ( ) return MPH;
    function Gas_Gauge ( ) return Gallons;
action
    in Steering_Wheel(A : Angle),
         Accelerator(P : Position),
         Brake(P : Pressure);
    out Warning_Light(S : Status);
end AutoControls;
```

Commentary:

Instruments such as Speedometer and Gas_Gauge are specified as **provides** functions. An observer of an AutoControls module, such as its parent architecture, can call these functions of the module. An AutoControls module must conform to the interface by supplying functions with these names that have appropriate types. Another module, a user say, in the same parent architecture could have **requires** functions that are connected to the Speedometer or Gas_Gauge by the parent architecture. When the user calls its requires functions it would obtain the return results of Speedometer or Gas_Gauge. So, in effect, they *output* readings when asked.

Controls such as Steering_Wheel, Accelerator and Brake are specified as **in** actions. An AutoControls component can observe (and react to) events generated by calls to these actions. A user component could call its **out** actions containing position, pressure or angle data, which would be connected by the parent architecture to these **in** actions. The connections would pass on the data in the **in** events so that the AutoControls component can observe and react to the data.

An AutoControls module can generate Warning_Light events with appropriate status data. These events are specified by the **out** action and are generated asynchronously. Hopefully, the parent architecture will do something useful with them, but the execution of AutoControls module does not await a response.

Subtyping rules (RAPIDE 1.0 Types Language Reference Manual) allow an AutoControls module to be replaced in a parent architecture by a module of a subtype. Such a subtype module may provide some new constituents, but it must also provide all the AutoControl **provides** functions as well. New provided functions for Speedometer and Gas_Gauge can differ only as allowed by the subtype rules. For example, Gas_Gauge could return a smaller range of readings in Gallons so that any other component working by calling the previous Gas_Gauge could still use the new readings. A new Steering_Wheel action could work on a wider range of angles but not a smaller range, so the new car could turn the same corners as the old car. A subtype must not require more functions than before. Adding actions of either mode does not affect subtyping. So, for example, a subtype module must have a Warning_Light.

□

Example: *Specification of concurrent behavior*

```
...
behavior
  Speedometer ( ) > 55  ||>
    Accelerator(0)  ||  Brake(High)  ||  Warning_Light(On);
  ...
```

Commentary:

This transition rule could be part of the behavior of the AutoControls interface. It is triggered by a Boolean condition and specifies a reaction consisting of the generation of three independent events. Two of these are **in** events which will be observed by the AutoControl module itself. The other is an **out** event which may cause other events to be generated according to the parent architecture's connections. This transition is an *agent* rule, which means that the events generated by any triggering of the rule are independent of those generated by any other triggering.

□

Example: *Automobile interface*

```
type Automobile is interface
service  Controls : AutoControls;
-- ...other parts
behavior
     CurrentSpeed : MPH;
     CurrentGas   : Gallons;
     ...

   function Controls.Speedometer ( ) return MPH is
   begin
        return CurrentSpeed;
   end;

  ...
end interface;
```

Commentary:

An automobile's interface contains a service of type AutoControls called Controls. This means that the interface contains the constituents declared in its Controls service. This allows the set of those constituents of an automobile to be connected by a single connection rule (Chapter 3) to a dual set of consituents in other components whose interfaces have a *dual* service of type AutoControl.

The example illustrates how the behavior part can conform to the interface. It provides a function, Controls.Speedometer, that implements (or is a denotation for) the function name declaration in the service. Gas_Gauge could be implemented similarly.

□

Example: *A driver interface*

```
type Driver is interface
-- ...other parts.
service  ControlPanel : dual AutoControls;
behavior
   ...
     ControlPanel.Speedometer ( ) > 55  ||>
                 ControlPanel.Accelerator(0);
```

```
        (?S in Status) ControlPanel.Warning_Light (?S)  ||> ...;
      ...
    end Driver;
```

Commentary:

The driver interface contains a service called ControlPanel that is of a type that is a dual of AutoControls. By applying the rewriting rules of section 2.5 the driver interface is equivalent to:

```
type Driver is interface
-- ...other parts.
requires
    function ControlPanel.Speedometer ( ) return MPH;
    function ControlPanel.Gas_Gauge ( ) return Gallons;
action
    out ControlPanel.Steering_Wheel(A : Angle),
         ControlPanel.Accelerator(P : Position),
         ControlPanel.Brake(P : Pressure);
    in ControlPanel.Warning_Light(S : Status);
behavior
-- Behavior from control panel service.
   ControlPanel.Speedometer ( ) > 55  ||>
      ControlPanel.Accelerator(0)  ||  ControlPanel.Brake(High)
      ||  ControlPanel.Warning_Light(On);

-- Other behavior defined in driver interface.
      ControlPanel.Speedometer ( ) > 55  ||>
                  ControlPanel.Accelerator(0);
      (?S in Status) ControlPanel.Warning_Light (?S)  ||> ...;
    ...
end Driver;
```

The dual constituents of **provides** functions are **requires** functions and conversely, and the dual constituents of **in** actions are **out** actions and conversely. This means that driver modules can communicate through ControlPanel by calling its **provides** functions to get instrument readings (i.e., treating them as **requires** functions), and by generating events of the ControlPanel **in** actions (i.e., treating them as **out** actions). The dual transition rule is the same. As shown here, the transition rule calls the Speedometer function and outputs Accelerator events. Also, ControlPanel.WarningLight is an **in** action of the driver; it can observe warning light events that are sent to it.

As we shall see in Chapter 3, drivers and automobiles can be connected together by connecting their AutoControl type services, which are duals, with a single connection rule.

□

Example: *An RS-232 service*

```
type  RS232 is interface
               action
                  in RXD ( );      --Receive Data.
                     CTS ( );      --Clear to Send.
                     DSR ( );      --Data Set Ready.
```

```
                DCD ( );        --Data Carrier Detect.
             ... ;
           out
                TXD ( );        --Transmit Data.
                RTS ( );        --Request to Send.
             ... ;
         end RS232;

type Computer is
         interface
         service   S1, S2 : RS232;
             ...
         end Computer;

type Modem is interface

         service   S : dual RS232;
             ...
         end Modem;
```

Commentary:

RS-232 is a common interface used between computers and modems. It defines some 20-odd signals, some of which travel from the computer to the modem, others from the modem to the computer. Here RS−232 is an interface type with **in** and **out** actions corresponding to the signals.

Using services, the RS−232 interface is defined once. Our computer interface declares some RS−232 services and the modem interface declares a **dual** RS−232 service. A computer and a modem can then be connected in an architecture by a single connection between an RS−232 service of the computer and a dual RS−232 service of the modem. In effect, the one connection between the services defines over 20 connections between the constituents of the computer and modem interfaces.

Note that a more ambitious interface would contain a behavior part defining RS−232 protocols. Also, the actions can be reordered into more traditional groups of related **in** and **out** actions.

□

Example: *An interface with a service set*

```
type Application_Interface is
interface
action
    in   Begin_Call (X : Tid);
         Commit_Call (X : Tid);
         ... ;
   out   Begin_Return (X : Tid);
         Commit_Return (X : Tid);
         ... ;
end;

type Resource_Interface is
```

```
interface
action
    in   Prepare_Call (X : Tid);
         Commit_Call (X : Tid);
         RollBack_Call (X : Tid);
         . . .  ;
    out  Prepare_Return (X : Tid);
         Commit_Return (X : Tid);
         . . .  ;
end;

type Transaction_Manager (NumRMs : Integer) is
interface
    . . .
service
    Application : Application_Interface;
    Resources (1 .. Num_RMs) : dual Resource_Interface;
    . . .
end;
```

Commentary:

A Transaction Manager object, TM say, has an interface type that results from instantiating the type constructor shown for some value of NumRMs. The TM interface consists of a single service called Application of type Application Interface, and a set of NumRMs copies of services of the dual of the Resource Interface type. Each service in the set can be named using the notation, TM.Resources (1), ..., TM.Resources (NumRMs). Although this notation resembles expressions, service names are not expressions. These service names do not denote objects, they denote services (which are sets of constituents) of the TM interface. For example, TM.Resources (1) is the service containing the **out** action, TM.Resources (1).Prepare_Call, etc.

□

Chapter 3

Architectures

An architecture is a module containing a set of modules (called *components*), a set of *connection rules* that define communication between the components, and a set of *constraints.* An architecture is a restricted form of module as defined in RAPIDE 1.0 Executable Language Reference Manual since it may contain only some of the features allowed in modules.

The purpose of architectures is to provide a construct for defining a module as a network of communicating components (see the Guide to Rapide). It encourages a style of building modules by gradual instantiation of an interface connection architecture. [1] That is, initially, all of the components will be interface objects [2] if the architecture does not give module denotations to the objects that it declares or creates. Such an architecture can be executed if the interface types of its components contain executable behaviors (see Section 2.6). When the behavior of an interface connection architecture is deemed satisfactory, a process of gradual instantiation and testing as described in the Guide to Rapide can be followed. However, architectures do not enforce this style of system development, and object connection architectures can also be modelled in RAPIDE.

3.1 Syntax

```
architecture_generator_declaration ::=
        architecture identifier '(' [ formal_parameter_list ] ')'
              [ return type_expression ] is
              { architecture_declaration_item }
        connect { connection }
        [ constraint { architecture_constraint } ]
        [ initial { executable_language_statement } ]
        end [ architecture ] [ identifier ] ';'


architecture_literal ::=
        architecture
              { architecture_declaration_item }
        connect { connection }
        [ constraint { constraint_language_constraint } ]
        [ initial { executable_language_statement } ]
        end [ architecture ]
```

[1] See Guide to Rapide

[2] see terminology, Section 1.1

```
formal_parameter_list ::=
        type_language_formal_parameter_declaration_list
```

```
architecture_declaration_item ::=
        type_language_type_declaration
|       type_language_type_constructor_declaration
|       type_language_object_name_declaration
|       executable_language_module_decl_item
|       map_declaration
```

Commentary

> Note that the syntax productions for *architecture declaration item* given here are redundant; all items from the types language are contained in *module decl item* but are mentioned explicitly for emphasis later on.

□

Terminology

See Section 1.1.

Architectures and their instances (below) are referred to collectively as *architectures*.

A *component* of an architecture is an object of an interface type that belongs to the *context* (see Section 4.2.1) of the architecture. Note that context can vary at runtime.

The term *interface component* is used to refer to an interface object that is a component of an architecture.

An architecture that results when an interface component of a given architecture is assigned a module is called an *instance* of the previous architecture.

A *static* architecture declares all components by object and object name declarations.

A *dynamic* architecture is an architecture having a varying number of components, or connections that vary at runtime. For example, an architecture may declare the interface types of components and rely on creation rules (Section 3.4) to define when, during execution, components of those types are created.

3.2 Static Semantics

Violations of restrictions stated below result in compiletime errors.

The names visible in an architecture generator are the interfaces of the predefined types (RAPIDE 1.0 Predefined Types Reference Manual), names of constituents of the architecture's interface, names declared in the architecture's declarative part, names declared in the file containing the architecture generator declaration, and names made visible by selection (e.g., names that are constituents of the interfaces of names that are visible).

3.3 Semantics

An architecture generator is a function that, when applied to actual parameters returns an architecture of the declared return type. If there is no **return** *type_expression* clause, the return type is Root, the type with the empty interface. The function call notation is used for architecture generator calls.

Architectures are created in two ways:

1. By evaluation of a call to an architecture generator.
2. By evaluation of an architecture literal.

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 RAPIDE 1.0 Executable Language Reference Manual, section 2.5). After parameter association, the call executes sequentially as follows:

1. Create a new module – the new architecture.
2. Generate the predefined Start event of the architecture.
3. Elaborate the declaration items of the new architecture.
4. Elaborate the connections of the new architecture.
5. Return the new architecture.

Architecture literals are evaluated in this same order.

As a result of the creation sequence, the predefined Start event of an architecture causally precedes events generated by the elaboration of the architecture's components. Further, the Start event causally precedes events generated by the connections of the architecture.

An architecture terminates whenever *(i)* all of the objects that it can observe have terminated and *(ii)* all of its observers (Section 1.1) have terminated.

Commentary

> The initial state (i. e., local objects) of each architecture generated by calling the same generator may be distinct. For example, if an architecture generator contains an object declaration which includes a call to a module generator for that object, then the object declaration is elaborated (calling the module generator and forming a *new* object) each time the architecture generator is called.
>
> Connections, like state, are elaborated for each new architecture. For example, if an architecture generator defines three connections, then two calls to the generator will elaborate six connections.
>
> An architecture does not contain exception handlers at its outer scope, as do modules (RAPIDE 1.0 Executable Language Reference Manual).

□

3.4 Connections

The *connection* part of an architecture contains connection rules and creation rules.

3.4.1 Purpose

A *connection rule* may connect the architecture's components to each other or the architecture's interface to its components (or both). A connection between components defines equivalences between calls to functions of components, or dependencies between events generated and received by components. A connection between the architecture's interface and its components defines equivalences between calls to function name constituents of the interface and calls to functions of components (and conversely), or it defines dependencies between events of the interface's actions and events of component's actions. A connection rule may depend upon runtime conditions, and may connect varying numbers of components.

Creation rules define conditions that lead to creation of new components.

3.4.2 Syntax of connections

```
connection ::=
        [ { pattern_language_placeholder_declaration } ] connection_rule_list
|       [ { pattern_language_placeholder_declaration } ] connection_generator


connection_rule_list ::=
        guarded_expression_generator_list to connection_set ';'
|       service_list to connection_set ';'
|       pattern_list to connection_set ';'
|       pattern_list '=>' connection_set ';'
|       pattern_list '||>' connection_set ';'


guarded_expression_generator_list ::=
        guarded_expression_generator { ',' guarded_expression_generator }


guarded_expression_generator ::=
        expression_generator [ where boolean_expression ]


expression_generator ::=
        executable_language_selected_expression [ '(' parameter_list ')' ]


parameter_list ::=
        parameter { ',' parameter_list }


parameter ::=
        pattern_language_placeholder
|       expression


pattern_list ::=
        pattern { ',' pattern }
```

```
service_list ::=
        service_name { ',' service_name }
|       service_set_declaration


connection_generator ::=
        generation_scheme generate
               connection
               { connection }
        end [ generate ] ';'


generation_scheme ::=
        for indentifier ':'  type_expression in expression '..'  expression
|       for indentifier ':'  type_expression in expression
|       for expression ':'  expression next expression
|       if expression


connection_set ::=
        connection_set_item { ',' connection_set_item }


connection_set_item ::=
        expression
|       pattern
|       service_name
|       generation_scheme generate
               connection_set
        end [ generate ]


pattern ::=
        pattern_language_pattern
```

Terminology

A connection consists of an optional sequence of placeholder declarations (RAPIDE 1.0 Event Pattern Language Reference Manual) called the *outermost placeholder declarations*, followed by a connection rule list or a connection generator.

A *connection rule* consists of an optional sequence of outermost placeholder declarations, a *trigger* to the left of an operator, an operator (**to**, $\Rightarrow$, $||>$), and a *body* to the right of the operator. The scope of the outermost placeholder declarations extends to the end of the rule.

An *instance* of an expression generator is a legal expression obtained by replacing each placeholder by the same expression (of a type compatible with the placeholder) at all of its occurrences in the generator.

A *connection rule list* is a shorthand notation for a list of connection rules with the same operator. First of all, the left and right sides of a connection rule list may be lists of event pattern triggers and event pattern bodies. Each connection rule in the list consists of the outermost placeholder declarations, a *trigger* that is a member of the list of triggers, the operator, and a *body* that is a member of the list of bodies. Secondly, the left and right sides of a connection rule list may be lists of expression generators (i.e., lists of triggers and bodies respectively). Each connection rule in the list consists of the outermost placeholder declarations, a trigger which one of the expression generators in the list of triggers, the operator, and a body which is one of the expression generators in the list of bodies.

A connection rule with the **to** operator is called a *basic connection*.

A connection rule with the ⇒ operator is called a *pipe connection*.

A connection rule with the ||> operator is called an *agent connection*.

A basic connection rule whose trigger is an expression or expression generator is called an *object connection*. If the trigger expression is a function call, such a rule is also called a *function call connection*.

A basic connection rule between service names is called a *service connection*.

A connection whose body contains a call to a module generator or to the predefined **new** operation, is called a *creation rule* since execution of its body may create objects.

Commentary

> An expression generator designates the set of expressions that are its instances. A pattern denotes the sets of events (posets) that match it (RAPIDE 1.0 Event Pattern Language Reference Manual).

□

3.4.3 Static Semantics of Connections

Restrictions stated below are for a single connection rule. Violations result in compiletime errors.

The scope of outermost placeholder declarations at the left of a connection rule extends to the end of the rule. Placeholders in the body of a connection rule must be declared in the outermost declarations of the rule and must appear in the trigger in such a way that they are guaranteed to be bound by any match or instantiation of the trigger. That is, if the trigger is a pattern, then a placeholder in the body must be bound by any match of the trigger; if the trigger is an expression generator, then a placeholder in the body must be bound by any instantiation of the trigger. Placeholders declared locally in the trigger of a connection rule have a scope within the trigger determined by the pattern language rules (RAPIDE 1.0 Event Pattern Language Reference Manual).

Patterns in the body of a connection rule are *restricted patterns*: *(i)* basic patterns may not contain a performer part, and *(ii)* patterns may not contain the <=> operator.

A parameter of an expression generator must be either a placeholder or an expression that does not contain placeholders.

For a *basic connection rule*, if its trigger is an expression then the body must be an expression; if its trigger is an expression generator then the body must be an expression generator; if its trigger is a service name then the body must be a service name; and if the trigger is a basic event pattern then the body must be a basic event pattern. Optionally the trigger can be guarded. These are the only allowed forms of basic connection.

If the trigger and body of a basic connection rule are expressions, then the trigger must either denote, or be a call to, a **requires** object constituent of a component or a **provides** object constituent of the architecture's interface. Correspondingly, the body must either denote, or be a call to, a **provides** object constituent of a component or a **requires** object constituent of the architecture's interface. The denotation of the body must be of a subtype of the type of the denotation of the trigger, or a subtype of the return type of the trigger, as the case may be. If the trigger and body of a basic connection rule are expression generators, then the same restrictions apply to all instances of the rule.

If the trigger and body of a basic connection rule are service names, the service named in the body must be of the **dual** type (section 2.5) of the type of the service named in the trigger, or of the same type if one (and only one) of the services is a constituent of the interface of the architecture.

In any connection rule in which the trigger and body are patterns, an action name of a basic pattern of a trigger must be an **out** action of a component or an **in** action of the architecture's interface. Correspondingly, an action name of a basic pattern in the rule's body must be an **in** action of a component or an **out** action of the architecture's interface.

The trigger and body of a *pipe connection* or an *agent connection* must be patterns, and may be complex patterns. Optionally, the trigger may be guarded.

When the trigger and body of a connection rule are patterns there are no type requirements between the declarations of actions whose names are action names of basic patterns in the trigger and body. Type correctness of the action calls in patterns with respect to the declarations of the actions follows the same rules as for function calls (see RAPIDE 1.0 Types Language Reference Manual).

Commentary

> Connections are used to wire components together by connecting their dual features, (e.g., **requires** functions to **provides** functions, or **out** actions to **in** actions, or dual services) and to wire the architecture's interface to components by connecting like features.
>
> Patterns in the bodies of connections are restricted patterns so that whenever the trigger is matched the resulting instance of the body is a well-defined (possibly empty) poset.
>
> A basic connection between services is usually *bi-directional* in the sense that constituents from each service will be action names in a trigger of a basic connection defined by the service connection.
>
> Basic connections between objects include as the commonest case, basic connections between functions.

□

3.5 Semantics of Connections

The elaboration of a connection consists of elaborating the generation schemes (if any), and then elaborating each connection rule in the connections that are generated.

All connections of an architecture are elaborated independently.

Once a connection is elaborated it can then observe (as described below) all the events that are received (Section 4.2) by the architecture.

3.5.1 Semantics of Object Connections

The result of elaborating an object connection rule is that the expression (without placeholders) that is an instance of the rule's trigger is given as its denotation the denotation of the corresponding instance of the rule's body. If the trigger is guarded then the guard is evaluated each time an instance of the trigger is evaluated; if the guard is FALSE then the denotation of the trigger instance is undefined, otherwise it is defined as described above. In the case that an expression is an instance of triggers of more than one object connection rule, then whenever that expression is evaluated, *all* of the corresponding instances of the connection rule bodies are evaluated and *one* of their values is chosen arbitrarily as the denotation of that expression.

Commentary

> A call to a **requires** function may be an instance of the triggers of more than one object connection, each connecting it to a call to a **provides** function. The bodies of all these connections are evaluated, with their possible side effects. One result is returned as the value of the call.
>
> A function call connection rule can be viewed as defining *aliases* of calls to a function (be it a **requires** function of a component or a **provides** or **private** function of the architecture's interface) to a **provides** function of a component or a **requires** function of the interface, as allowed by the static semantic restrictions. Aliasing implies synchronization of calls. Thus interface function provide a feature for synchronous communication.

□

3.5.2 Semantics of Service Connections

Elaboration of a service connection results in a set of basic connections between the constituents of the objects whose interfaces contain those services. Each of these basic connections connects constituents denoted by the same name in the services. Thus, a basic connection between dual services, say S and R, results in a set of basic connections, one for each pair of constituents with the same name in the two services. This set of basic connections is constructed as follows : *(i)* in a basic connection between objects, the **requires** object name (say, S.f) is the trigger expression and the **provides** object name (say, R.f) is the body expression; *(ii)* in a basic connection between actions, the **out** action name (say, S.A) is the action name of the trigger and the **in** action name (say, R.A) is the action name of the body; *(iii)* a basic connection between nested (dual) services is further decomposed into a set of connections between their consituents with the same names. Any guards in a service connection are added to the triggers of the corresponding basic connections. The resulting set of basic connections is then elaborated.

3.5.3 Semantics of Event Pattern Connections

Next, we describe the semantics of connections whose triggers and bodies are patterns.

Whenever a component calls one of its **requires** functions or **out** actions, or the architecture's **provides** functions or **in** actions are called, the resulting events become observable by the connections of the architecture. The events generated by calls to a function f, namely f'call events, are described in section 2.3. The events generated by action calls are described in section 2.4.

Event observation. Observation of events, matching of connection rule triggers, and evaluation of guards have the same semantics as observation of events, matching of transition rule triggers and evaluation of guards in interface behaviors (see Section 2.6).

Connection rule execution. Whenever an event has been selected and attempts to match all rule triggers with the current set of selected events have been made, those rules whose triggers are matched are then executed independently (possibly concurrently).

To execute a rule, the binding of placeholders in the match of its trigger is applied to its body and the resulting instance of the body is executed. (Note: all placeholders in the body must be bound by the match, see static semantic restrictions above). The execution of an instance of a body results in either evaluating an expression or generating a poset of events as described for each kind of rule below. Events are generated as result of action calls. **in** action calls result in events that are observable by components of the architecture; **out** action calls result in events that are generated by the architecture at its own interface, and are observable by the architecture and its parent.

Even though all placeholders in a pattern body of a connection rule must be bound by a match of the trigger, an instance of a pattern body may contain basic patterns that omit parameters (i.e., "don't care" parameters, see RAPIDE 1.0 Event Pattern Language Reference Manual). Such missing parameters are chosen arbitrarily to define the action calls to be executed, and the poset that is generated.

Commentary

> An action call does not await a result. Execution of the caller may proceed immediately after calling an action, independently of connection rules that may be triggered by the event generated by the action call. Actions thus model *asynchronous* communication.

□

3.5.4 Basic pattern connections

When a basic connection between actions is triggered, the instance of its body is a basic pattern. The result of executing this body is a call to an action that generates an event matching the basic pattern. Any parameters of the action call that are ommitted from the basic pattern are chosen arbitrarily. The generated event is causally and temporally equivalent to the event that caused the match. The generated event is observed according to the semantics of action call (Section 2.4).

In the case that more than one basic connection is triggered by an event, then *all* of the connections are executed and the resulting action calls generate events that are equivalent to the triggering event and to each other with respect to the dependency and temporal relationships between events.

Commentary

> Consequently, if a basic connection connects an **out** action A1 of one component with an **in** action A2 of another, then every call to A1 generates an event E1, which results in a call to A2 which generates an event E2. These two events will have the same timestamps w.r.t. any given clock, and will have identical causal relationships to all other events.

□

3.5.5 Pipe connections

An instance of a pipe connection's body resulting from a match of its trigger is a pattern that matches posets of events. Execution of the instance results in a set of action calls that generates a poset matching the instance (the *generated poset*). Events of the generated poset also causally depend upon all events contributing to the match that triggered the execution of the rule, and upon all events generated by previous executions of the connection. Any parameters of the action calls that are omitted by the basic patterns in instance of the body are chosen arbitrarily.

Commentary

The generated poset matches the dependencies and time stamps of the instance of the connection rule body.

A pipe connection can be used to model a single reactive thread of control if the generated poset is a linear sequence of dependent events.

□

3.5.6 Agent connections

An agent connection is executed similarly to a pipe connection by executing action calls to generate a poset that matches the instance of the rule body (the *generated poset*). An event generated by executing the body also causally follows all events involved in the match triggering the execution of the rule, but it is causally unrelated to the events generated by previous executions of the connection unless otherwise defined to be dependent upon them (e.g., an event generated on a previous execution of the rule triggers the current execution).

Commentary

An agent connection may be used to model a reactive system whose execution creates new threads of control to execute each reaction.

It is possible for the posets generated by pipe and agent rules to violate timing consistency, e.g., by generating events with an earlier time stamp than that of a triggering event. The compiler may not detect these errors.

□

3.5.7 Connection generators

Connection generators allow connections to be defined between sets of components in an architecture provided there is a method of enumerating the sets. Typically the components are declared as components of objects that have an iterator (see RAPIDE 1.0 Predefined Types Reference Manual, e.g., arrays of components).

For generation scheme. The *type_expression* and any *expressions* of the **for** generation scheme must be defined when the connection generator is elaborated. The *identifier*, called the *loop parameter*, may occur in the connection generator body, and denotes an object of the type_*expression*.

The elaboration of a **for** scheme proceeds as follows for each iteration of the loop parameter over its set of denotations. First, all occurrences of the loop parameter in the body are replaced by the current denotation of the loop parameter. This may result in the following cases : *(i)* a list of connection rules, each of which is then elaborated; *(ii)* another *generator*, which is then elaborated; *(iii)* a list of expressions which are bodies of connection rules with a common trigger and operator, and each of these rules is then elaborated; *(iv)* another generation scheme which is the body of a connection rule, which is then elaborated.

If generation scheme. The boolean expression of the **if** generation scheme is evaluated at elaboration time. If it is true, then elaboration of the generator body proceeds according to the following cases. *(i)* the body is a list of connection rules, and each connection rule is elaborated; *(ii)* the generator body is another *generator*, which is then elaborated; *(iii)* the body is a list of expressions which are bodies of connection rules with a common trigger and operator, and each of these rules is then elaborated; *(iv)* the body is another generation scheme which is the body of a connection rule, which is then elaborated.

Commentary

A designer will often wish to define a number of identical connections, or connections which differ only slightly. Also, it may be that a set of connections should only be defined in an architecture under certain conditions, determinable when the architecture is elaborated, or the set of expressions to be evaluated when a connection is triggered is dependent upon the state of the parent architecture at the time of triggering. *Connection generators* allow this.

□

Commentary

Events are selected for matching triggers in a one-at-a-time order consistent with event dependency in the same way as transition rule triggering. This implies that a unique value will be used as the value of a guard, although which value may depend upon the order in which events are selected.

Similarly to transition rules, guards in the pattern triggers of connection rules are evaluated when an event is generated that can be observed by the architecture. Their values are associated with the event for future reference should the event be the last one used to match a guarded pattern. The number of guards that need be evaluated for any observable event can clearly be reduced in general. Any given event will be a potential participant in matching only a subset of the guarded patterns in the set of connection triggers. This is a compiler optimization.

However, the execution of connection bodies can take place independently since there is no shared state between connections — unlike transition rules in behaviors.

□

Commentary

Pipe and agent connections provide a powerful and concise pattern notation for defining connections between components.

A pipe expresses the semantics of a connection such as a wire or pipeline — provided, of course, that the pattern in the body does not contain the $||$ operator. Events generated by a pipe depend on all events previously generated by that pipe. Pipes are fundamental in hardware architectures, for example.

An agent expresses the semantics of a connection of a dynamic nature such as radio transmission. There is no dependency between events generated on different triggerings of an agent, unless implied by state operations. Agents are fundamental in dynamic architectures in which communication depends upon runtime conditions.

A complex connection can be replaced by a new component (called a *connector*) and a set of basic connections between the connector and the components that were connected by the complex connection. Thus, basic connections are the only connection feature essential to define architectures.

□

3.6 Constraints

The constraint part of an architecture contains constraints expressed in the constraint language (RAPIDE 1.0 Constraint Language Reference Manual), and map generator applications (Chaper 5).

3.6.1 Syntax

```
architecture_constraint ::=
        constraint_language_constraint
|       map_generator_application
```

3.6.2 Semantics

The poset behaviors generated by the architecture must satisfy the constraints declared in the constraint section (RAPIDE 1.0 Constraint Language Reference Manual).

Commentary

> Constraints may, for example, require connections to trigger in certain orders, thus defining a communication protocol. As discussed in the commentary of Section 2.8, constraints may be supported in different ways by various tools.

□

3.7 Type conformance of architectures

Purpose

An interface type specifies constituents of modules of the type. These constituents are visible to observers of a module. A module's interface is a contract between the module and its observers. An architecture is a module, and consequently must conform to its interface type.

Semantics

The type of an architecture is determined when the architecture is created. The type of an architecture created by an architecture generator is the type specified after the keyword **return** in the architecture generator, after parameter association. The type of an architecture created by an architecture 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 architecture 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 RAPIDE 1.0 Executable Language Reference Manual, section on qualified-expressions), then the type of the literal is the type given in the qualified expression.

These are the only allowed uses of an architecture literal.

Conformance of an architecture to its type requires that:

1. A **provides** or **private** object name in its interface must have a denotation that is an object of a subtype of the type of the object name in the interface.

   Commentary

   To achieve this, the architecture should have basic connections connecting **provides** object names in the interface to **provides** objects of its components, or alternatively it can declare an object with the same name. □

2. Type names declared in the interface must have corresponding types with the same names declared in the architecture.

3. Any poset of **in** and **out** interface events resulting from executing the architecture should satisfy constraints in the interface.

   Commentary

   That is, calls to **in** actions of the interface will generate events that are observed by the architecture and may trigger its connections. Such triggerings may result eventually, possibly after internal events communicated between components, in calls to **out** actions of the interface that generate **out** events of the interface. These **out** events will be related by dependencies and time to the **in** events, thus defining posets of interface events. The posets of interface events must satisfy the interface constraints. □

4. An interface poset generated by an architecture must be a super-poset of the poset generated by its interface behavior (i.e., a superset of the events with an identical dependency order on the common subset).

   Commentary

   In this sense, an interface behavior (Section 2.6) acts as a constraint on an architecture of that interface type. □

3.8 Examples

Example: *Basic event pattern connection.*

```
type Button is interface
action in Press ( );
       out Move ( );
end Button;

type Sensor is interface
action in Activate ( );
       out Light (X : enum On, Off end enum);
end Sensor;

architecture Lift_Panel is
   A_Button : Button is ...;
   A_Sensor : Sensor is ...;
connect
   A_Button . Move to A_Sensor . Activate;
   ...
end Lift_Panel;
```

Commentary:

The type of the lift panel is Root. The lift panel architecture declares two components, a button and a sensor. There is a basic connection between the **out** action Move of the button and the **in** action Activate of the sensor. Each move event generated by the button causes an activate event to be received by the sensor. Call this a *pair* of move/activate events. The basic connection defines an equivalence between pairs of move and activate events. The two events in a pair are identical with respect to dependency and time – i.e., they have the same dependencies with all other events and the same timing. It is not possible to distinguish the two events either by a clock or by looking at their causal history.

□

Example: *A dynamic basic connection.*

```
connect
    (?P in Person, ?L in Lift, ?B in Button)
        ?P.Push(?L, ?B) to ?L.?B.Press;
```

Commentary:

We assume an architecture whose components are persons and lifts. Lifts have **provides** button modules in their interface. The basic connection shown connects all persons and lift buttons. If the number of person components of the architecture can vary, then the number of objects connected by this connection at any time will vary. For the connection to be well formed the Push action must be an **out** action of a person. Also, note that all placeholders in the right pattern are bound by a match of the trigger.

The connection defines causal and temporal equivalence between pairs of Push and Press events. So the connection links persons and lift buttons in a very strong way; the actions of pushing buttons and pressing buttons appear identical according to time and causal history.

□

Example: *An object connection.*

```
connect
    (?i in Integer)
        M.F(?i) to N.G(?i+1);
```

Commentary:

Assume M and N are components of an architecture that contains this connection rule, and F is a **requires** function of M and G is a **provides** function of N. The trigger and body of this rule are both expression generators. Any instance of the trigger, obtained by replacing ?i by an integer expression will define a corresponding instance of the body with the same integer expression for ?i. The return type of any instance of M . F must be a super type of the return type of the corresponding instance of N . G. This connection rule means that any expression that is an instance of M . F(?i) has as its value (or denotation) the value of the coresponding instance of N . G(?i+1). For example, the value of M . F(3) is N . G(4). Essentially, this rule aliases calls to M . F to calls to N . G. Synchronization is implied; i.e., a call to M . F blocks awaiting the return of results from corresponding call to N . G.

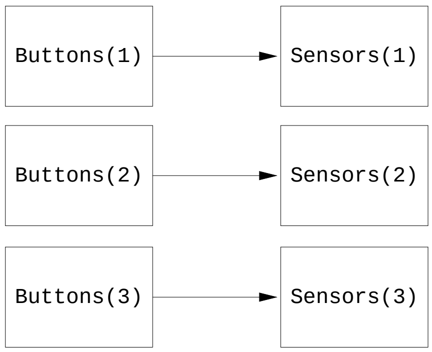

Figure 3.1: Direct connection architecture.

□

Example: *Generating a set of basic connections*

```
type Count is  range 1 .. 50;

architecture Lift_Panel is
      Buttons : array [Count] of Button is ...;
      Sensors : array [Count] of Sensor is ...;
      ...
connect
      for I : Count in 1..50 generate
            Buttons[I].Move to Sensors[I].Activate;
      end generate;
      ...
end Lift_Panel;
```

Commentary:

Lift_Panel contains an array of fifty Button modules and an array of fifty Sensor modules. The generate statement expands into fifty basic connections, each connecting the Move action of the i^{th} button to the Activate action of the i^{th} sensor. For each call matching Buttons[I]D.Move, an equivalent (under causality and time) event matching Sensor[I].Activate is observed by the Sensor[I] module. The architecture created by this connection is illustrated in Figure 3.1.

□

Example: *Fan-in: a many-to-one connection*

```
connect
      (?X in Button) ?X.Move to Sensors[1].Activate;
```

Commentary:

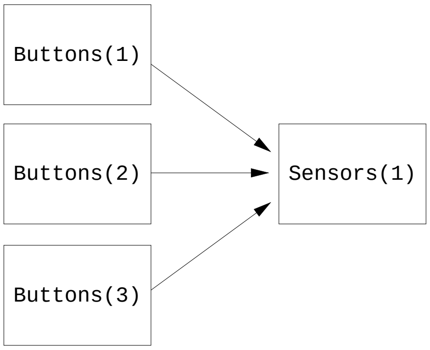

Figure 3.2: Fan-in connection architecture.

This is an example of a many-to-one connection, where **out** actions of many Button modules are connected to an **in** action of a single Sensor module. Since ?X can match any Button, all Move actions of Button modules are connected to the Activate action of Sensors[1]. This architecture is depicted in Figure 3.2. Note that fan-in connections between **requires** functions and a **provides** function can be defined similarly.

If one wanted to restrict this example to connecting only the buttons of the array, then one would use a connection generator that iterates over the buttons in the array and connects them to Sensors[1].

Note also that the pattern language (RAPIDE 1.0 Event Pattern Language Reference Manual) defines a shorthand for a placeholder declaration over a type, such as ?X : Button;. This shorthand allows the placeholder to be omitted, and the type name used with a "::" symbol, e.g., Button::Move.

□

Example: *Fan-out: a one-to-many connection*

```
connect
    Buttons(1) . Move  to
      for J : Count generate
          Sensors[J] . Activate
      end for;
```

Commentary:

In this example, a single Move action is connected to fifty activate actions. Each Move event triggers fifty basic connections (i.e., those generated by the generator) causing fifty Activate events. All fifty-one events are equivalent under causality. This architecture is shown in Figure 3.3.

□

Example: *A dynamic agent connection*

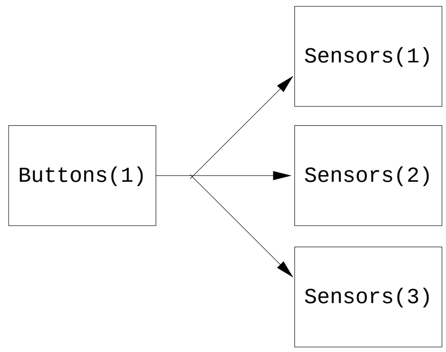

Figure 3.3: Fan-out connection architecture.

```
architecture Air_Control_Sector is
      SFO : Control_Center;
      ...
connect
      (?A in Airplane, ?M in Msg)
            ?A.Radio(?M) where ?A.InRange(SFO) ||> SFO.Receive(?M);
      ...
end Air_Control_Sector;
```

Commentary:

Assume the interfaces of Airplane and Control_Center are already defined. The connection defines event communication between any airplane object and a particular control center. Whenever any airplane (a match for ?A) generates a Radio event containing a message and the InRange predicate of that airplane is true when the Radio event is generated, then SFO will observe a Receive event with the same message. The connection triggers only when an airplane is in range. It is an agent connection, so there is no dependency between events generated by different triggerings.

This connection rule is a conditional broadcast between all airplanes and the control tower. It defines communication in a system that may have varying numbers of airplane components. It defines a causal relation between pairs of Radio and Receive events; different pairs of these events can be independent.

□

Example: *Using RS-232 to connect computers and modems*

```
architecture Office is
      PC : Computer;
      Mod: Modem;
      ...
connect
      PC.S1 to Mod.S;   -- bi-directional flow of events.
      ...
end Office;
```

Commentary:

Following the RS−232 example in Section 2.10, the office architecture connects the interfaces of two of its components, a computer and a modem. A single connection rule between the pair of dual RS−232 services in PC and Mod is all that is necessary. It denotes a set of basic connections between pairs of RS−232 constituents with the same name; in each connection the **out** action is the trigger.

□

Example: *Connecting services in drivers and cars*

```
architecture Vehicle_Test is
        Dummy : Driver;
```

```
        TestCar : Automobile;
connect
        Dummy.ControlPanel to TestCar.Controls;
         ...  -- other connections.
end Vehicle_Test;
```

Commentary:

In Section 2.10 automobiles were defined to have a service of type AutoControls, and drivers were defined to have the corresponding **dual** service. The connection in Vehicle_Test denotes the set of basic connections between each pair of constituents with the same name, one in the service and the other in the **dual** service. So, for example, whenever the dummy calls its **requires** speedometer function, the call is connected to the **provides** speedometer function in the TestCar. The return value is returned to the dummy. Similarly, whenever the dummy generates an **out** SteeringWheel(angle) event, TestCar will receive an **in** event with an identical tuple. When the TestCar generates Warning_Light events, the dummy will receive them as input warning light events in its control panel.

□

Example: *A generate connection*

```
-- see service set example in Chapter 2
type Resource_Manager is
interface
    ...
service
    R_Intface : Resource_Interface;
    ...
end
architecture System (NumRMs : Integer) return DTP_System is
    TM : Transaction_Manager (NumRMs);
    RMs : array [1 .. NumRMs] of Resource_Manager;
    ...
connect
    for I : Integer in 1 .. NumRMs generate
        RMs[I].R_Intface to TM.Resources(I);
    end generate;
     ...
end;
```

Commentary:

The service set example in Chapter 2 defines the interfaces, Resource Interface and Transaction Manager, the latter containing a set of services of the dual of Resource Interface. The System architecture generator here declares a Transaction Manager component, TM, and an array of Resource Manager components, RMs. The value of NumRMs will be defined when a System architecture is generated. At elaboration of the System architecture, the generate connection will result in NumRMs service connections, one between each member of RMs and the corresponding dual service in the TM.

□

Chapter 4

Communication Model

The *communication model* explains how RAPIDE modules [1] communicate. Modules communicate by calling functions or by generating events and observing events. A communication implies the possibility that data is passed between modules. At any point in an execution two modules may or may not be able to comunicate. The ability to communicate is a *dynamic* concept depending upon runtime conditions.

The communication model depends upon the features that allow a module to *initiate* or *receive* communication, and upon the concept of a module's runtime *context*. Note that initiating and receiving communication are not complementary — e.g., module A may be able to receive events from B but not able to initiate communication with B.

4.1 Initiating Communication

There are two ways a module P can initiate communication with other modules, *direct* communication and *broadcast* communication.

1. a module P initiates direct communication with another module Q by calling the **in** actions or **provides** functions in the interface of Q, or by selecting a visible object in the interface of Q.

   To do this, P must have a denotation for Q — i.e., it must be able to evaluate an expression that denotes Q. Such an expression may be evaluated in the behavior part of P if P is an interface object (Section 2), or in the body of P in the case that P is a module. Common kinds of expression that can denote Q include a name denoting Q, a module generator call that returns Q, or a placeholder that is bound to Q.

2. a module P *broadcasts* to other modules by calling its **out** actions or **requires** functions in its own interface (Chapter 2).

   The events and function calls broadcast by a module P are received by another module Q if P is in the *context* of Q (see Section 4.2.1).

Communication by generating events by calling actions is asynchronous; the caller does not block to await a response. Communication by calling functions is synchronous; the caller blocks awaiting a return value from the call (see Architectures LRM).

[1] An object of an interface type is a module, which includes architectures (i.e., the result of an architecture generator call) as a special case. For emphasis we sometimes refer to "modules and architectures".

4.2 Receiving communication

A module P can receive direct or broadcast communication.

A module P receives direct communication whenever its **in** actions or **provides** functions in its own interface are called.

Whenever Q belongs to the *context* of P, the **out** events and **requires** function calls broadcast by Q are received by P.

4.2.1 Context

The *context* of a module P at any point during its execution is the set of modules Q such that:

1. if the declaration of Q is a member of the declarative part of P and the elaboration of P results in the elaboration of Q, then Q is added to the context of P.
2. if P creates Q, then Q is added to the context of P.
3. if P calls **link**(Q) (Predefined Types LRM), then Q is added to the context of P.
4. if a third module, R calls **link**(Q, P) then Q is added to the context of P.
5. if P calls **unlink**(Q), then Q is subtracted from the context of P.
6. if a third module, R calls **unlink**(Q, P) then Q is subtracted from the context of P.
7. if Q terminates then Q is subtracted from the context of P.

P *creates* Q if P is a module and is the least global module enclosing the expression whose evaluation resulted in Q.

In the cases $1, 2$ where P elaborates Q or creates it, P is said to be the *parent* of Q (conversely Q is a *child* of P).

Commentary

> A module has only one parent, and the child of a child of P is not a child of P. Note that when a module is elaborated it begins life in its parent's context. Later, as a result of being linked to other modules, it can enter other contexts, and possibly be in different contexts concurrently.
> A module will remain in the context of its parent, and possibly the contexts of its parent's children, unless it is passed as a parameter of its parent's **requires** function calls or **out** events. In this case, it may become possible to link the module into contexts outside of its parent.

□

4.3 Observing Events and Function Calls

A module P can *observe* the events and function calls that it receives by either direct or by broadcast communication.

The mechanisms that allow a module to observe events and function calls are *matchable patterns* in the behavior part of P (if it is an interface module), or in P's module or architecture body (i.e., if P is the result of a module or architecture generator call). Matchable patterns are:

- the triggers of:
  - processes (await and when statements) in the module (Executable LRM),
  - transition rules in the behavior of an interface module (Chapter 2).
  - connections in an architecture (Chapter 3).
- constraints.

Once a received event or function call has been successfully *observed* in matching the trigger of a process, transition rule, connection rule, or constraint, it cannot be observed again for matching that particular trigger or constraint; observation of events and function calls, and matching of patterns, are described in the relevent LRMs.

Commentary

> The use of placeholders in observing events and function calls adds a new dynamicism to the communication model that is not found in most programming and simulation languages. For example, when a pattern is matched during observation of an event, its placeholders are bound to objects in the event (i.e., event parameters such as the module that generated the event, the time of the event, etc.). A placeholder denotes the object it is bound to for the lifetime of that binding — usually a "short lifetime" such as the execution of a single transition or connection rule. Since the event could be generated in another scope that is not visible to the receiving module (the usual case with broadcast), placeholders can denote objects that it may not be possible to denote in any other way within the receiving module. As an analogy with Natural Language, a pattern might be, "tall dark stranger". When matched, the placeholder, "stranger", can denote an object that may otherwise be unknown. The denotation of "stranger" can subsequently be used in initiating communication with that stranger. The pattern language provides the ability to refer to causal and temporal relations, e.g., "the tall dark stranger that opened the door and came in with the blonde".

□

4.4 Examples

Example: *A dynamic architecture*

```
type Airplane is ...
type  Control_Center is ...
type  Msg is ...
type EnRouteCenter is interface
     action in Accept(X : Airplane);
            out HandOff(X : Airplane);
end EnRouteCenter;

architecture FremontSector return EnRouteCenter is
     ?A : Airplane; ?M : Msg;
     SFO : Control_Center;
     ...
connections
     Accept(?A) => link(?A);
     ?A.Radio(?M) where ?A.InRange(SFO)
```

```
            ||> SFO . Receive(?M);;
      HandOff(?A)  =>  unlink(?A);
  end  FremontSector;
```

Commentary:

Assume the interface types of Airplane, Control_Center, and Msg are already defined. The interface for Airplanes, for example, may define that they can generate Radio events, provide a Position attribute (i.e., a function), etc.

The interface type for EnRouteCenter declares **in** and **out** actions that allow an EnRouteCenter to receive and send aircraft as parameters of **in** and **out** events. So new airplanes (modules) are continually being received and handed-off.

FremontSector is an architecture generator. When it is called it returns an architecture of type EnRouteCenter.

The FremontSector architecture defines the communication between airplanes and its Control_Center module by connection rules. Note first of all, that SFO is a *child* so that its broadcast events are always received by the AirControlSector. Furthermore, the sector can communicate directly with SFO. This is not true of airplanes which are received as parameters of Accept events from the containing architecture. The Sector must first bind the airplane to a placeholder by matching. The first connection **link**s an airplane whenever an Accept event with that airplane is received. Events generated by the airplane will then be received by the AirControlSector. The second connection defines the following event-based communication:

> The rule triggers whenever any airplane (a match for ?A) generates a Radio event containing a message ?M and the InRange predicate of that airplane is true in the state when the radio event is generated. ?A is bound to the airplane and ?M to the message. Then the rule calls SFO's Receive action, resulting in generating a Receive event with the same message. The connection triggers only when an airplane is in range.

Finally, whenever the Sector hands off an airplane it **unlink**s from it, so communication from the airplane is no longer received.

This is a dynamic interface connection architecture. It defines communication between varying numbers of airplanes and a ControlCenter. The second connection rule defines a conditional broadcast between all airplanes and the control tower. It is essentially a fan-in connection. The architecture does not define any communication between airplanes. So, if the modules conform to it, we can conclude that airplanes do not communicate directly with one another.

□

Example: *Observability of Events*

```
architecture User_Arch() return A_Type is
    Comp1, Comp2 : Comp_Type;
initial
    ... new(Dyn_Comp_Type) ...
connect
    -- What events are observable for matching here?
end;
```

Commentary:

The events observable for matching at the marked point are:

- The in- and out-events of A_Type generated by User_Arch.
- The out-events of Comp_Type generated by Comp1 and Comp2.
- The out-events of Dyn_Comp_Type generated by the dynamically generated module.

This shows the events that can be observed by the connection rules of the architecture. Note that if we change the example to a module generator by changing "**architecture**" to "**module**" and "**connect**" to "**parallel**", the same set of events would be available for matching the triggers of processes in the module.

□

Chapter 5

Maps

The purpose of *maps* is to define how the executions of one architecture may be interpreted (or viewed) as executions of another architecture. More generally, a map can interpret the executions of several architectures taken together (called its *domains*) as executions of another architecture or interface (called its *range*). Maps are an experimental feature of RAPIDE.

The principle feature of maps is the pattern triggered rule that generates a poset. This is similar to a transition rule in an interface behavior (section 2.6). It provides the necessary expressive power needed to define correspondences between architectures. Hence the map construct described here is often called an *event pattern mapping*.

A map generates a new poset of events from a given poset. Essentially, a map triggers on a *domain poset* generated by its domain architecture(s) and generates a new *range poset*. The range poset may be a possible behavior of the range architecture, but is generated by the map rather than by the range architecture. Events of the range poset are causally incomparable to events of the domain poset. Dependencies between events of the range poset are induced from the dependencies between events of the domain poset.

The range posets generated by a map must satisfy the constraints of its range architecture or interface. Thus a map is a *constraint* on its domain parameters, which can be objects (usually architectures) and other maps. Maps can be composed by using them as domain parameters in the definition of other maps.

RAPIDE allows two uses of maps: *(i)* as constraints, and *(ii)* as domains of maps.

Particular applications of event pattern mappings that have been established experimentally include *(i)* reduction of complexity of large simulations by mapping posets of events in a detailed low level simulation into single events in a higher level simulation, and *(ii)* runtime comparison of an architecture with a standard (or reference) architecture. Comparison of architectures is accomplished by mapping the behaviors of the domain architecture(s) into behaviors of the range architecture and checking them for consistency with the constraints of the range architecture.

When a hierarchical design methodology is used to develop a low level detailed architecture from a highly abstract specification, maps relating architectures at successive levels of abstraction can be composed transitively to define maps across several levels.

5.1 Syntax

```
map_generator ::=
```

```
        map identifier '(' [ formal_parameter_list ] ')'
                      from domain_indicator_list to range_indicator is
               {architecture_ declaration_item }
        rule
               { state_transition_rule }
        [ constraint
               { constraint_language_constraint }]
        end [ map ] [ identifier ] ';'

map_literal ::=
        map
        from domain_indicator_list is
               {architecture_ declaration_item }
        rule
               { state_transition_rule }
        end [ map ] [ identifier ] ';'

domain_indicator_list ::=
        domain_indicator ',' domain_indicator

domain_indicator ::=
        [ identifier ':'  ]  type_expression
|       [ identifier ':'  ]  module_generator_name


range_indicator ::=
|       type_expression
|       module_generator_name

map_generator_application ::=
        function_application
```

Terminology

A map generator has a list of domain parameters (its *domains*) indicated by a domain indicator list, and a range parameter (its *range*) indicated by its range indicator.
Transition rules of a map are called *mapping rules*.
The objects declared in a map are called the *state* of the map.

Static Semantics

Formal parameter names and domain indicator names must be unique; e.g., a domain indicator cannot have the same name (i.e., identifier) as a formal parameter.

The type of a domain or range indicator must be an interface type.

A map generator application uses the same notation as function application with the domains being the last parameters in the actual parameter list.

Commentary

Module generators include architecture generators. The intended application of maps is to architectures although they apply to modules in general.

□

5.2 Semantics

A *map* is the result of evaluating a map generator application or evaluating a map literal. It can be used either as a constraint on its domains or as a domain of another map (see Section 5.4).

A map generator is applied to a list of objects (actual parameters) corresponding to its formal parameters, and a list of actual domains, which are modules and maps, corresponding to its domain indicators. If an actual domain corresponds to a domain indicator that is an object name declaration, then its range type must be a subtype of the domain indicator's type. If an actual domain corresponds to a domain indicator that is a module generator name, then the actual domain must be module generated from that module generator or a map whose range indicator is the same module generator.

Evaluation of a map generator application begins with parameter and domain evaluation and association. The rules for parameter and domain evaluation and association, as well as rules for overload resolution, are the same as those for evaluating function calls (see RAPIDE 1.0 Executable Language Reference Manual 2.5), except that the compatibility of a domain with the corresponding domain indicator relies on the rules described in the preceeding paragraph rather than on type conformance.

First the actual parameters are evaluated and associated with the formal parameters, and then the actual domains are evaluated. Then evaluation of the application proceeds sequentially as follows :

1. A new object is created to be used as the denotation of any use of **self** that refers to the map.
2. The predefined Start event of the map is generated. The Start event precedes all the events generated by the map.
3. The object and type declarations of the map are elaborated.
4. The rules of the map are elaborated.
5. The constraints in the implementation type (Section 5.2.1) of range indicator are elaborated. If their elaboration depends upon the elaboration of objects declared in the range indicator, the declarations of those objects must be repeated in the map, and those objects are used in place of the objects in the range indicator. The elaboration of these constraints makes them ready to check conformance of events generated by the mapping rules.
6. The constraints of the map are elaborated. This has the effect of making them ready to check conformance of events that are observed by the mapping rules, and also events generated by the mapping rules.

Map literals are evaluated in this same order.

Commentary

Note that the range indicator is not elaborated.
The object created to denote the **self** of the map is used, for example, as the performer part of some of the events generated by the map (Section 5.2.3).

□

After elaboration, a map observes events from its domains, generates events of its range indicator when its rules are triggered, and checks those events for conformance with its range indicator's constraints, as defined in the following sections.

5.2.1 Visibility within map generators

A map generator has visibility to the *implementation type* as defined below of each of its domains and its range.

Implementation types

1. the implementation type of a type expression is the type denoted by the type expression.
2. the implementation type of a module generator is an interface type constructed by extending the interface type of the module generator as follows : *(i)* for each object declaration of the declarative part of the module generator and each formal object parameter of the module generator, an object name declaration with the same name, type and type constraint (if any) is added to the **provides** part of the interface type, *(ii)* for each action declared in the declarative part of the generator, an **in** action with the same name and signature is added to the interface type, *(iii)* all other kinds of declarations in the declarative part of the generator are added to the **provides** part of the interface type, and *(iv)* constraints in the generator are added to the constraints of the interface type.
3. the implementation type of a module literal is defined similarly.
4. the implementation type of a map generator is the implementation type of its range indicator.
5. the implementation type of a module or map (i.e., the result of a module or map generator application) is the implementation type of the module or map generator with the formal parameters of the generator replaced by the corresponding actual parameters of the generator application.

Commentary

> A map generator has visibility to the declarations in its domains and range (module and map generators) as well as their interfaces. It can therefore name events and patterns of events that can be generated by components of its domains and range as well as at their interfaces.
>
> Note, however, that if a domain generator *creates* components of a type declared outside of itself, the map generator will not be able to name events of those components if the type is not visible to the map generator. Maps are intended for use on domains that are, in a sense, self-contained and have small external continuations.

□

5.2.2 Events Observed by a Map

The events available to a map during execution are defined as follows. If an actual domain of a map corresponds to an object name indicator then the map observes events of the domain's interface. If an actual domain of a map corresponds to an indicator that is a module generator, then if the actual domain is a module, the map observes any event generated in the domain if that event is generated by a call to an action or function of the domain's interface or its components' interfaces; if the actual domain is a map, then the map observes all events generated by the domain.

5.2.3 Semantics of mapping rules

The *rules* in a map consist of a list of agent state transition rules (see Section 2.6).

Basic patterns in rule triggers are restricted to matching events of **in** and **out** actions of the domain implementation types and 'call and 'return events of functions (Section 2.3) of the domain implementation types.

The rule bodies may call operations on objects of the map state, its domains, and components of its domains. A rule body may not call constituents of the map's range implementation type.

Basic event patterns in the poset generators in rule bodies must generate events of actions of the range interface type or of objects named in the range implementation type. They may also generate 'call and 'return events of functions of the range implementation type and of functions of objects named in the range implementation type.

A parameter of a basic event of the range type, including the *performer* of the event, may be an object name in the range implementation type or an object of the map state or an object name resulting from a call to Map_New (see below) or the object name denoting **self** of the map.

A poset generator may omit to define a performer of a range event, in which case a default performer is defined as follows : the performer of an **out** event of an object named in the range implementation type is the object name; the performer of an event of an **in** or **out** action of the interface type or an **in** action of an object named in the range implementation type is the **self** of the map.

The triggering and execution of map rules follows the same semantics as for transition rules in interface behaviors (Section 2.6) with the exception that they do not define dependencies between domain events that trigger them and range events that they generate.

Commentary

The events that may be observed by mapping rules are the events received and generated by the domains of the map and their components. The map rules trigger on these events. The bodies of map rules then eexecute by operating on the state of the map and generating range posets consisting of events of actions and functions of the range implementation type and of objects named in the range implementation type. There is no dependency relation between domain and range posets.

A user should take care that a mapping rule that calls constituents of a domain does not have a side effect on its domain, e.g., a call to a function provided by a domain should not have side effects. This style guide is not enforced. In general runtime checks are needed to ensure there are no side effects on a domain.

A map can use its visibility of implementation types to trigger on function call and return events in the domains, and to generate function call and return events associated with functions in the range architecture or interface without invoking the functions.

Note that the map state can implement functions declared in the range implementation type. Functions in the map state can be called to compute parameters of range events that are generated by the map.

□

New object names can be created by calling the polymorphic function

function Map_New(**type** T) **return** T;

which returns a fresh name of an object of the argument type. Map_New may only be called within maps.

Commentary

> The purpose of Map_New is to enable maps to generate range posets corresponding to dynamic range architectures.

□

Dependency Order of Generated Events. The range events generated by each triggering of a map rule have a dependency order defined by the poset generator of the rule. In addition, events generated by separate triggerings of rules can have a dependency order defined in terms of the dependencies between the domain events that triggered the rules. This is called an *induced* dependency.

Several definitions of induced dependency are supported by RAPIDE tools. Some of them are defined below.

Weak induced dependency. A range event E1 causally precedes another range event E2 if either (1) both are generated by the same triggering of a rule and such a relationship is asserted by the generating pattern, or (2) all the domain events that contributed to triggering the rule that generated E1 causally precede *all* the domain events that contributed to triggering the rule that generated E2.

5.2.4 Conformance to Range Constraints

The posets generated by a map must conform to the constraints of the implementation type of its range indicator.

Commentary

> Maps are one-directional from domain to range. They interpret patterns of events in domain architectures as patterns of events in the range architecture.
>
> Maps may be composed transitively by using them as domains of maps.

□

Commentary

> Maps can be used to make error detection in large architectures more efficient. Usually the number of range events generated by a map are far fewer than the number of domain events. Of course, this application assumes the range constraints are logically complete with respect to the mapped behaviors in the sense that any violation of an architecture constraint will lead to a violation of a range constraint.

□

Commentary

> The rationale for making maps and map generators untyped is to prevent maps from being treated as objects. If they were objects, one would expect to be able to wire them up as components of architectures using the range type as their interface type. A call to a range function should return a result. But a map's range functions don't have implementations; the range function declarations are there simply to define which map events are generated.

□

5.3 Map Declarations

To simplify notation for composition, maps are declared using a syntax similar to object declarations.

5.3.1 Syntax

```
map_declaration ::=
              identifier_list is map map_generator_application ';'
```

5.3.2 Semantics

A map declaration may appear in a declarative region. When a map declaration is elaborated the map generator application is evaluated and the identifiers denote the resulting map.

5.4 Uses of Maps

The only allowed uses of maps are *(i)* in map declarations, *(ii)* as a domain in a map generator application, and *(iii)* as a constraint in the constraint part of a module (or architecture) generator (Section 3.6).

5.5 Examples

Example: *Constituents of implementation types*

```
type T is interface I : Integer; end;

module M() return T is
   I, J : Integer is 12;
end;

map P() from X:T, Y:M to T1 is
      A : Integer is X.J;   -- Illegal
      B : Integer is Y.J;    -- Legal
rule ...
end;
```

Commentary:

The type of domain X is the interface type T. Even though map P could be applied to an actual domain (for X) that is generated by M say, the internal objects of X are not visible.

Domain Y is generated by M. Therefore the implementation type of Y, which is the implementation type of M, is visible in P. This means that the names I, J of type integer are visible in the implementation type of Y.

□

Example: *Naming constituents of implementation types*

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

module M() return U is
     X : T is M1();
     ...
end;

type ImpType_M is interface
provides X : T;
     ...
action in C;
end IMpType_M;

map P() from D:M to T1 is
rule
    X.A ...      -- Illegal
    D.X.A ...    -- Legal
     ...
end;
```

Commentary:

ImpType_M shows the implementation type of module M. It provides the object name, X, as well as constituents of M's interface. This is the interface type of D that is visible in P. Thus constituents of D's interface can be named in P; they must be qualified by "D".

□

Example: *Map rules are agent rules*

```
map P() from X:T, Y:M to T1 is
     A : Integer is X.I;
     B : Integer is Y.J;
rule
    R => Q; -- Illegal.
    R ||> Q; -- Legal.
end;
```

Commentary:

All mapping rules must be agent rules because a map does not imply any dependency between its domain events and its range events. Pipeline rules would imply dependency between domain and range.

□

Example: *A Paradigm Application of Maps as Constraints.*

```
architecture US_Congress (... params ...) return Public_View is
    ... -- lots of objects and connections.
end US_Congress;

type Laws is interface
    ... -- lots of actions and legal constraints.
end Laws;

map Monitor(... params ...) from Cong : US_Congress
                             to Laws is
    ...
rule
    ... -- rules interpreting posets of local events happening
        -- in the US Congress as posets of events defined by
        -- laws and constrained by laws.
end Monitor;

architecture Washington_95 return Some_View is
   Cong_104 : Public_view is US_Congress(... actuals ...);
   Press is map Monitor(... actuals ...; Cong_104);-- reports violation of Laws.
constraint
    Monitor(... actuals ...; Cong_104); -- also reports violation of Laws.
end Washington_95;
```

Commentary:

The US_Congress architecture has a Public_View interface but much hidden activity goes on in the Congress itself. The Laws interface defines abstract legal actions and constraints on how those actions may take place (i.e., constraints on posets of legal actions, which we consider to be laws). The Monitor defines a mapping of the internal events of any US congress into legally defined events in Laws. Washington_95 is an architecture defining communication between the 104^{th} Congress (an instance of the generator for US_Congress), and other objects in Washington. During the elaboration of this architecture, the map declaration, Press will be elaborated and become active. During execution of Washington_95, Press will map events that happen both internally in the 104^{th} Congress and at its Public_View interface, into legally defined events. Constraint violations will be reported. The Monitor application in the constraint part of the architecture will do the same thing. Press, however, can also be used as a domain in other map applications.

□

Example: *Mapping a Database to a Read-From Graph*

```
type DBMS is interface
   function Transaction (...);
end DBMS;

type Object_Mgr is interface
```

```
    function Read(T : Tid; O : Object) return Data;
    function Write(T : Tid; O : Object; D : Data);
    function Version(D : Data) return Natural;
         ...
end Object_Mgr;

type Transaction_Mgr is interface
    function Begin_Trans return Tid;
    function Commit(T : Tid);
    function Abort(T : Tid);
         ...
end Transaction_Mgr;

-- architecture generator for DBMS.
architecture DBMS_Gen ( ) return DBMS is
      OM : Object_Mgr is ...;
      TM : Transaction_Mgr is ...;
      ... -- implementation of interface function, Transaction.
end DBMS_Gen;

type ReadFromGraph is interface
    action  Make_Node(T : Tid);
    -- makes a node for a new transaction id.

    action Make_Arc(T1, T2 : Tid);
    -- draws a directed arc from node for T1 to node for T2.
         ...
constraint
      -- the graph contains no cycles.
end ReadFromGraph;


map RFG_View from DBMS_Gen to ReadFromGraph is
      ...
rule
 -- make a node representing a new transaction.
       TM.Begin_Trans'call → TM.Begin_Trans'return(?T)
              => Make_Node(?T);

 -- draw a directed arc between two nodes when one transaction be-
 -- comes dependent on another, either by reading the same data or by
 -- writing the next version of the data.
       OM.Write'return(?Ti, ?O, ?D1) and
            ( (OM.Read'return(?Tj, ?O, ?D2) where ?D1 = ?D2)
            or
            ( (OM.Write'return(?Tj, ?O, ?D2) where Version(?D1) + 1 = Version(?D2)))
              where ?Ti /= ?Tj
              =>    Make_Arc(?Ti, ?Tj);
end RFG_View;

architecture System return Root is
      My_DB : DBMS is DBMS_Gen ( );
      ...   -- operations on My_DB.
```

```
constraint
      map RFG_View(My_DB);
      ...
end System;
```

Commentary:

This is an example of mapping an architecture to an interface. A DBMS architecture executes its interface function calls (transactions) by internally breaking them down into sets of small transactions (Read, Write, Begin_Trans, etc.) on its components.

The map generator RFG_View is intended to represent the transaction activity in a DBMS architecture in terms of a directed acyclic graph. It takes as domain a module that must be generated from DBMS_Gen. Consequently RFG_View has visibility to the implementation type of DBMS_Gen and hence to the objects OM and TM. Its rules trigger on patterns of events denoting operations on these objects and generate graph events.

To define the mapping rules, RFG_View has visibility to the implementation type of DBMS_Gen. For example, a mapping rule generates an event of the action of making a new graph node whenever a new transaction is allowed by the TM component of the architecture. Another rule generates an event of the action of drawing a directed arc between nodes representing two transactions whenever patterns of events occur during those transactions that signify certain calls to the read and write functions of the OM component of the architecture.

In the System architecture a database object, My_DB is generated from DBMS_Gen. A map constraint in System is generated by applying RFG_View to My_DB as its domain. This map generates events corresponding to graph actions when patterns of events occur in the architecture of My_DB. Consequently, behaviors consisting of both interface and internal events of My_DB are mapped into events of the interface type, ReadFromGraph.

To see how the map behaves as a constraint on My_DB, imagine that various transactions are performed on My_DB during the execution of System. The map maps the transactions to ReadfromGraph events. It produces an event behavior (events corresponding to actions to create nodes and draw arcs) resulting from events in My_DB which trigger the map rules. The mapped events must satisfy the interface constraints in ReadFromGraph, for example that the graph that results from these events must not contain cycles. The "no cycles" constraint will check for sets of events that denote construction of a cycle in the graph.

□

Example: *Legal and illegal map applications*

```
architecture A(X : T) return T1 is   ...
architecture B(X : T) return T2 is  ...
type C(X : T) is interface  ...
map M(X:T) from A to B is ...
map M'(X:T) from B to C is ...
     ...

  H, K : T ...            -- objects of type T.
  P : T1 is A(H);         -- P is an architecture of type T1.
```

Q **is map** M(H, P); -- *Q is a map from P to implementation type of B(H).*
R **is map** M′(H, Q); -- *R is a map from P to interface type C(H).*
S **is map** M′(K, R); -- *illegal map composition.*

Commentary:

P is an architecture generated by evaluating A(H). Q is a map generated by applying M to H as its actual parameter and P as its domain. P is legal as a domain because it is generated from A, the domain indicator of M. Q is a map from P to the implementation type of its range indicator instance, B(H).

R is a composition of maps M′, M. It results from applying M′ to H as parameter and Q as domain. Q is a legal domain here because its range indicator is B, the same module generator as the domain indicator of M′. R is a map from P to the interface type, C(H).

The execution of Q proceeds by sending inputs to P; the rules of Q react to the execution of P by generating events of the implementation type of B(H). Similarly, the execution of R is the result of its rules reacting to the execution generated by Q.

An illegal map application is S. This is because R is not a legal domain in M′(K, R). the range indicator of R is an interface type, C(H), whereas the domain indicator of M′ is architecture generator B.

□

Chapter 6

Bindings

The purpose of bindings is to view an object through different interfaces at runtime. Bindings allow an object to be associated with interfaces that need not be related in the subtype hierarchy. Bindings are an experimental feature of RAPIDE.

Bindings extend the feature already provided by the architecture construct (Chapter 3) for connecting events and objects in an architecture to events and objects in an interface, and conversely.

An architecture generator with a non trivial return interface type may connect **provides** constituents and **in** actions of the interface to similar constituents of components, and conversely it may connect **requires** constituents and **out** actions of components to similar constituents of the interface. As a result, a flow of events is defined whereby events received at the interface cause events in the architecture, and events generated by the architecture cause events at the interface. This provides an encapsulation of the architecture generator as an interface – i.e., architectures generated from it are modules of the interface type, and can be connected to other modules through that interface.

Bindings extend this feature by allowing an architecture (or any module) to be bound dynamically to an interface. Essentially, distinct interface "views" of the hidden structure of an object can be provided by bindings, simultaneously if desired. This functionality cannot be duplicated, for example, by a module generator with an object as a parameter – this would allow connections only between the visible constituents of the object and the interface. [1] As in the case of maps (Chapter 5), in order to define detailed correspondences between patterns of events in an architecture and in an interface, a binding has visibility to *all* of the components of the architecture – not just the interface of the architecture.

6.1 Syntax

```
binding_generator ::=
        binding identifier '(' [ formal_parameter_list ] ')'
                    from binding_domain return type_expression is
        connect
              { connection_list }
        end [ binding ] [ identifier ] ';'
```

[1] There are ways to write a module generator to allow other interface "views" in addition to its return type, but one must be able to predict those views when the generator is written.

```
binding_literal ::=
        binding
              from binding_domain is
        connect
              { connection_list }
        end [ binding ] [ identifier ] ';'


binding_domain ::=
        type_expression
|       module_generator_call
|       module_literal
```

For the syntax of connection see Section 3.1.

A binding generator has a list of formal parameters, a *domain* which must be a module or architecture generator call, a literal or an interface type, and a return interface type called its *range*.

Formal parameter names and the domain name must be unique; e.g., the domain cannot have the same name as a formal parameter. The parameters of a generator call denoting the domain must all occur in the formal parameter list.

6.2 Semantics

A binding consists of a list of *connections* (Section 3.4). The semantics of a binding are identical to those of an architecture (see Chapter 3) with the following modifications: *(i)* the connection rules have visibility to the implementation type (see Section 5.2.1) of the domain object rather than of its declared type, and *(ii)* the domain object is the only component of the binding.

A binding must satisfy the type conformance requirements of architectures with respect to the range type (see Section 3.7).

A *binding generator* is a module generator. It is called using the usual positional notation of function application with the actual parameters in the formal parameter positions and the domain module as the last parameter. The result of a binding generator call is a module of the range type (called a *binding*). A DOMAIN_ERROR exception is raised if the domain module is not of the given domain implementation type.

Commentary

The binding literal

binding B() **from** mk_obj **return** T **is**
    ...
end binding;

is semantically equivalent to an architecture generator

architecture B′(O: mk_obj′Implementation) **return** T **is**
    open O;
    ...
end binding;

where an actual parameter of B' must denote an object of the implementation type of mk_obj. The use of **open** indicates that the components of O are directly visible to the connections of the architecture.

□

Commentary

The semantics of connections in a binding is the same as for connections in an architecture (Section 3.4). Consequently, binding connections define dependencies and equivalences between triggering events and generated events, and aliases between functions of the interface and functions of domain architecture components.

□

Commentary

A connection in a binding can connect **provides** objects of the range interface to **provides** objects of the implementation type of the domain, and conversely for **requires** objects, following the rules for connections in architectures (Section 3.4). Similarly, it can connect **in** events of the range interface to **in** events of the implementation type of the domain and conversely for **out** events. This gives binding connections an ability to define new interface views of a module, which is the intended use of bindings.

A binding connection can also connect **requires** objects and **out** events of the implementation type of the domain to its **provides** objects and **in** events. This gives binding connections a limited ability to redefine the architecture of a domain by adding connections. This is not foreseen as a good use of bindings, but in the spirit of experimentation, it is allowed.

□

Commentary

Bindings are intended primarily to apply to architectures although they may be applied to modules in general. A more general definition would allow bindings to contain object declarations (state) and both state transition rules and connections. State transition rules would be more general than architecture connections because they could operate on the state of the binding in addition to generating posets. This added capability would facilitate applying bindings to modules in general — not only architectures.

□

6.3 Examples

Example: *Different views of an automobile architecture.*

```
-- Automobile interface from Chapter 3.
interface Automobile is
      service Controls : AutoControls;
      ...
end interface;
```

```
-- An architecture generator for the automobile type
architecture Ford return Automobile is

    Rad : Radiator;
    Eng : Engine;
    Trn : Transmission;
    Elc : Electrical_System;
    Inj : Fuel_Injection_system;
    ...
end architecture;


interface MechanicsView is
    function CoolantLevel( ) return Percentage;
    function OilLevel( ) return range High .. Low;
    function RPM( ) return Integer;
    action AdjustIdle(D : Integer);
    ...
end interface;

binding OpenHood from Ford return MechanicsView is
begin
    ...
connect
    -- function connections.
     RPM( )    to Eng.RPM( );
     Coolant.Level( )    to Rad.Resevoir.Level( );
     OilLevel( )    to Eng.Dipstick( );
     ...
     -- action connections.
     AdjustIdle(?D) => Inj.Setting(?D) || ...;
     ...
end OpenHood;

    FordCar : Automobile is Ford( );
    -- FordCar can be accessed only through its Automobile interface.

    Mechanic : MechanicsView is OpenHood(FordCar);
    -- Mechanic can directly access the architecture of FordCar.
```

Commentary:

The Automobile interface in Section 2.10 gives a driver's view in terms of controls and gauges. The Ford architecture will connect its radiator, engine, and other components to these interface constituents. When an instance of a Ford, FordCar, is generated this is the only interface that can be used to interact with it – e.g., to drive it.

During the lifetime of FordCar, a mechanic may need to view its architecture differently through a MechanicsView interface. To do this, the OpenHood binding is applied to FordCar to generate the Mechanic module. Mechanic can interact directly with the internal architecture of FordCar — even if the driver is driving it.

If we had tried to define MechanicsView as a module generator that takes Automobile modules as arguments, the mechanic would only be able to interact with FordCar through its Automobile interface.

□

Example: *Optimistic and pessimistic views of a glass.*

```
type Glass is interface
provides
   function Level return range 0.0 .. 1.0;
action in Raise;
         in Lower;
         in Pour_In;
         in Pour_Out;
         out Stop;
end Glass;

architecture Glass_Gen ( ) return Glass is ... end;

type Optimist is interface
provides
   function How_Full return range 0.0 .. 1.0;
action in Drink;
         out Enough;
end Optimist;

type Pessimist is interface
provides
    function How_Empty return range 0.0 .. 1.0;
action in Fill;
         out Enough;
end Pessimist;

binding O_View from Glass_Gen return Optimist is
connect
    How_Full to Level;
    Drink => Raise < Pour_Out;
    -- A Drink event causes a Raise event followed in time by a Pour_Out
    -- event.
    Stop => Enough;
end O_View;

binding P_View from Glass_Gen return Pessimist is
connect
    How_Empty to Level;
    Fill => Lower < Pour_In;
    -- A Fill event causes a Lower event followed in time by a Pour_In
    -- event.
    Stop => Enough;
end P_View;

   G : Glass is Glass_Gen ( );
```

```
O : Optimist is O_View(G);
P : Pessimist is P_View(G);
...
```

Commentary:

Bindings are module generators. Their domains are modules generated from given module generators. Here, both bindings, O_View and P_View, take as argument a module generated from Glass_Gen and return a module of type Optimist and Pessimist respectively. In this example they only use the Glass interface of the module and do not refer to its internals. Each of them binds a public function in their view interface to the public Level function in their argument module. Each of them has a connection rule that triggers on an **in** event from their interface and generates two **in** events in their domain architecture (a glass). In each case one of the generated events is timed to come after the other (the time delay is not specified). Each of them triggers on **out** Stop events from the glass and generates **out** Enough events in their interface.

In the main program, G is a glass module generated from Glass_Gen. O and P are modules with respectively optimistic and pessimistic views (i.e., interfaces) of G. They both share G so that their operations on it will affect each other's view of it.

□

Glossary

action: a name declaration indicating ability of a module to observe or generate events of an event type denoted by the declaration. See page 8.

agent connection: A connection whose operator is ——; an agent connection defines a causal ordering between triggering events and generated events on each triggering, but does not define an ordering between events generated by distinct evaluations of the connection. See page 29.

agent: a transition rule modelling a new thread of control for every execution. See page 13.

architecture conformance: the requirements of an architecture resulting from it being of a paricular type. See page 34.

architecture generator: A function conforming to a the syntax for architecture declaration; when applied to actual arguments it returns an architecture. See page 25.

basic connection: a connection rule with the operator **to**; basic connections define two functions, objects or actions to be causally and temporally identical. See page 28.

binding: a module generator taking as argument an architecture or module and returning a module of a particular interface type. See page 61.

body of a connection: the pattern or expression to the right of the operator of the connection See page 27.

body, of a transition rule: the sequence of executable language statements followed by a poset generator that occurs to the right of the operator of a behavior transition rule. See page 13.

communication model: the runtime conditions for two modules to be able to communicate. See page 43.

component, of an architecture: an object of an interface type that belongs to the context of the architecture. See page 2.

connection rule: a connection of an architecture consisting of an expression or pattern (the trigger), one of the operators **to**, =¿, ——¿, and an expression or pattern (the body). See page 28.

connector: an architecture component whose role in the architecture is to transfer data between other components. See page 33.

constituent, of an interface: A declaration in an interface type. See page 5.

creation rule: a connection rule whose body contains a call to a module generator or the **new** operation. See page 28.

domain events: events generated by the domain(s) of a map. See page 54.

domain poset: a poset generated by a domain of a map and causing the map to generate a range poset. See page 49.

dynamic architecture: an architecture having a varying number of components or a communication topology that varies during execution. See page 24.

function call connection: a basic connection between function calls. See page 28.

generate: a call to an action generates (or creates) an event of the event type denoted by the action name declaration, and which contains the actual parameters of the call. See page 8.

hidden constituent: a constituent of a module whose name is not declared in the module's interface. See page 5.

induced dependency: a dependency between range events of a map defined in terms of dependencies between domain events that generated them. See page 54.

instance, of an architecture: a module that results when an interface component in the architecture is assigned a module. See page 24.

instance, of an architecture: a module that results by assigning an a module to an interface object that is a component of the architecture. See page 2.

instance, of an expression generator: a legal expression obtained by replacing each placeholder by the same type compatible expression at all of the placeholder's occurrences in the generator. See page 27.

interface : the interface type expression denoting an interface type. See page 5.

interface component: an interface object that is a component of an architecture. See page 24.

interface constituent: a declaration in an interface. See page 5.

interface object: an object resulting from the elaboration of a name declaration where the type is an interface type. See page 2.

interface type An interface defines the externally visible constituents of modules of its type. Also called "interface". See page 5.

internal action: an action declared in a module. See page 8.

internal constituent: see hidden constituent. See page 5.

internal event: an event of an internal action. See page 8.

map: an event pattern mapping takes as argument an architecture or module and transforms its behaviors into behaviors of another architecture, module or interface. See page 49.

mapping rule: a state transition rule implementing a map. See page 53.

module constituent: types, actions, and objects declared in the module, or objects created in the module. See page 5.

module: A construct encapsulating an implementation of an interface type. See page 2.

object connection: a basic connection between objects. See page 28.

object: a module, interface object, or function. See page 2.

observer of a module: a the module itself, its parent, or any module which is linked to it – see Chapter 4. See page 2.

outermost placeholder declaration: a placeholder declaration on the left of a transition rule or connection rule; its scope is the following transition rule or connection rule. See page 27.

outermost placeholder declaration: a placeholder declaration on the left of a transition rule or connection; its scope is the following transition rule or simple connections. See page 13.

parent architecture, of a component: the architecture that declares the component, or declares its type and creates the component. See page 2.

pipe connection: A connection whose operator is $\Rightarrow$; pipe connections define a causal ordering between triggering events and generated events on each triggering, and also defines a causal ordering between events generated by distinct triggerings of the connection. See page 29.

pipe: a transition rule modelling a single thread of control. See page 13.

poset generator: a pattern appearing on the right side of a transition rule or connection See page 13.

range events: events generated by a map. See page 54.

range poset: a poset generated by a map. See page 49.

required constituent: a constituent of a **requires** part of a module's interface. See page 5.

root: the interface type with the empty interface. See page 25.

service connection: a basic connection between services. See page 28.

start event: a system event issued by an architecture before execution of its components can start. See page 25.

state, of a map: the objects declared in the body of a map generator. Cfr. *map components*. See page 50.

static architecture: an architecture having a fixed number of components and a fixed communication topology. See page 24.

trigger of a connection: the pattern or expression to the left of the operator of the connection See page 27.

trigger, of a transition rule: a pattern or boolean expression on the left side of the operation of a behavior transition rule. See page 13.

visible constituent: a constituent of a module whose name is declared in the module's interface. See page 5.

in action: an **in** action declared in an interface. See page 8.

in event: an event of an **in** action. See page 8.

out action: an **out** action declared in an interface. See page 8.

out event: an event of an **out** action. See page 8.

Index

in action, 8
in event, 8
out action, 8
out event, 8

action, 7
action_call, 8
action_declaration, 8
action_name_declaration, 8
action_part, 8
agent, 13
agent connection, 28, 29
architecture, 1, 2, 23–25
  conformance of, 34
  creation, 25
  type of, 34
architecture generator, 25
architecture_constraint, 34
architecture_declaration_item, 24
architecture_generator_declaration, 23
architecture_language_interface_declarative_region, 6
architecture_literal, 23

basic connection, 28
basic pattern connection, 31
behavior_declarative_item, 6
behavior_part, 13
binding, 61
binding_domain, 62
binding_generator, 61
binding_literal, 62
body
  of a connection, 27
  of a transition rule, 13

child
  ofa module, 44
communication model, 43
component, 23, 24
  of an architecture, 2
conformance
  of an architecture to a type, 34
connection, 1, 23, 26
  agent, 32
  pipe, 32
  service, 29
connection generator, 33
connection rule, 27
connection rule list, 28
connection_generator, 27
connection_rule_list, 26
connection_set, 27
connection_set_item, 27
connector, 33
constituent, 5
constraint, 17
context
  of a module, 44
creation rule, 28

domain events, 54
domain poset, 49
domain_indicator, 50
domain_indicator_list, 50
dynamic architecture, 24

event
  visibility, 9
events
  observed by a map, 52
exception
  DOMAIN_ERROR, 62
expression generator, 27
expression_generator, 26

formal_parameter_list, 24
function call connection, 28
function_name_declaration, 6

generate, 8
generation_scheme, 27
guarded_expression_generator, 26
guarded_expression_generator_list, 26

hidden constituent, 5

implementation type, 52
instance

of an architecture, 2, 24
interface, 1, 5
interface component, 24
interface constituent, 5
interface object, 2
interface type, 5
interface type expression, 5
interface_behavior_part, 6
interface_constraint_part, 6
interface_declarative_item, 3
interface_services_part, 6
internal action, 8
internal constituent, 5
internal event, 8

Link, 44

map, 49
map_declaration, 55
map_generator, 49
map_generator_application, 50
map_literal, 50
mapping rule, 50, 53
module, 1, 2, 23

object, 2
object connection, 28
semantics, 30
object_name_declaration, 6
observer, 2, 7, 25
observing events or function calls, 44
op, 13
outermost placeholder declaration, 13, 27

parameter, 26
parameter_list, 26
parent
of a module, 44
parent architecture, 2
pattern, 27
pattern_list, 26
pipe, 13
pipe connection, 28, 29
placeholder
scope, 14, 27, 28
poset generator, 13
poset_generator, 13
private interface constituents, 7

range events, 54
range poset, 49
range_indicator, 50
required constituent, 5
root, 25

service, 9
service connection, 28, 29
semantics, 30
service_declaration, 10
service_declarative_item, 10
service_list, 26
service_set_declaration, 10
service_set_index, 10
Start action, 14
Start event, 25, 51
start event, 25
state
of a map, 50
state_assignment, 13
state_transition_rule, 13
static architecture, 24

termination
of an architecture, 25
of an interface behavior, 16
transition rule, 12
transition_rule, 13
trigger, 13, 27

UnLink, 44

visibility
within map generators, 52
visible constituent, 5