DRAFT
Rapide 1.0 Constraint Language Reference Manual

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

July 17, 1997

Preface

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

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

This manual assumes familiarity with the RAPIDE overview given in the Guide to Rapide, and refers to other LRMs, particularly the Patterns LRM. It describes how constraints can be defined to constrain the behavior of RAPIDE programs.

There are two kinds of RAPIDE constraints: *concurrent* constraints, also called pattern constraints, which specify required and disallowed patterns of events produced by a program; and *sequential* constraints, which specify predicates on the state of a RAPIDE object.

Pattern constraints use the RAPIDE 1.0 Event Pattern Language; sequential constraints use a syntax based on first-order logic.

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

Contents

Part I

Constraints for Concurrent RAPIDE-1.0

Chapter 1

Pattern Constraints

The behavior of RAPIDE-1.0 programs may be constrained by *pattern constraints*, which describe a computation (recall that a computation is the set of events produced by a single execution of a RAPIDE-1.0 program). Pattern constraints denote the patterns of events (see the RAPIDE 1.0 Event Pattern Language Reference Manual) that are acceptable and unacceptable.

General concepts of pattern constraints are introduced in Section 1.1. *Filtering* is defined in Section 1.3; *constraint bodies* are defined in Section 1.4. A form of constraint with implicit filtering is discussed in Section 1.5. Examples are given in Section 1.6.

Section 1.7 defines conformance criteria of modules and maps to constraint.

1.1 Introduction

This section introduces the syntax and semantics for constraints. Later sections give more detail on the semantics of different parts of constraints.

1.1.1 Syntax

```
pattern_constraint ::=
        [ label ] observe filter_part constraint_body end [ observe ] ';'
|       simple_constraint_body_item ';'


filter_part ::=
        filter { '|' filter }


filter ::=
        pattern
|       [ identifier is ] ( map map_generator_expression | map_literal )
|       from basic_pattern_list


constraint_body ::=
        constraint_body_item { constraint_body_item }
```

```
constraint_body_item ::=
        simple_constraint_body_item
|       pattern_constraint
```

```
simple_constraint_body_item ::=
        [ label ] [ not ] match pattern ';'
|       [ label ] never pattern ';'
```

1.1.2 Informal Semantics

A RAPIDE constraint may have several components. It may have a *label*, which is used to identify the constraint for inheritance and error-reporting. Every constraint has a (possibly implicit) *filter*, which is used to filter the computation consisting of available events into zero or more subcomputations of interest. The *body* of a constraint has one or more *body items* that denote which computations produced by the filter are acceptable. And finally, constraints may be *nested*, so that the inner constraint operates on the computations produced by the filter of the outer constraint.

Each of these components is described in the following sections.

An example of the basic constraint semantics is illustrated in Figure 1.1. A constraint may contain *filters*, here called F and G. Filters observe and produce *computation/substitution pairs*; that is, they observe a computation to which is attached a set of *placeholder substitution values* that have been derived from the filtering process.

A filter observes exactly one such pair, and produces zero or more pairs. (The number of computations produced by a computation depends on the kind of filter.)

The first filter in a constraint observes a single computation, namely the set of *available events* of the module or map to which the constraint applies; and the empty substitution set.

So filter F observes the single pair defined above, and produces pairs, each of which is observed by an instance of filter G, which in turn produces some number of pairs.

The result of a sequence of filters is a set of computations; in the figure, the resulting computations are $G11, G12, \ldots, G21, G22, \ldots$. Each of these constraints must satisfy the body of the constraint, the pattern P.

If the constraint is positive, i.e. "**observe** F | G **match** P;," then each of the resulting computations must be an exact match (that is, must match with no events remaining) for P.

If the constraint is negative, i.e. "**observe** F | G **not match** P;," then none of the resulting computations may be an exact match for P.

Though there are several forms of pattern constraints, each may be rewritten to one of these two basic forms.

Semantics of immediate sequence in constraints. The semantics of the immediate sequence pattern operator "$\triangleright$" depends upon its location in a constraint. If used in a pattern filter (see Section 1.3.1), it applies to the computation observed by a filter instance. If used in a constraint body, for each computation being matched, it applies to that computation.

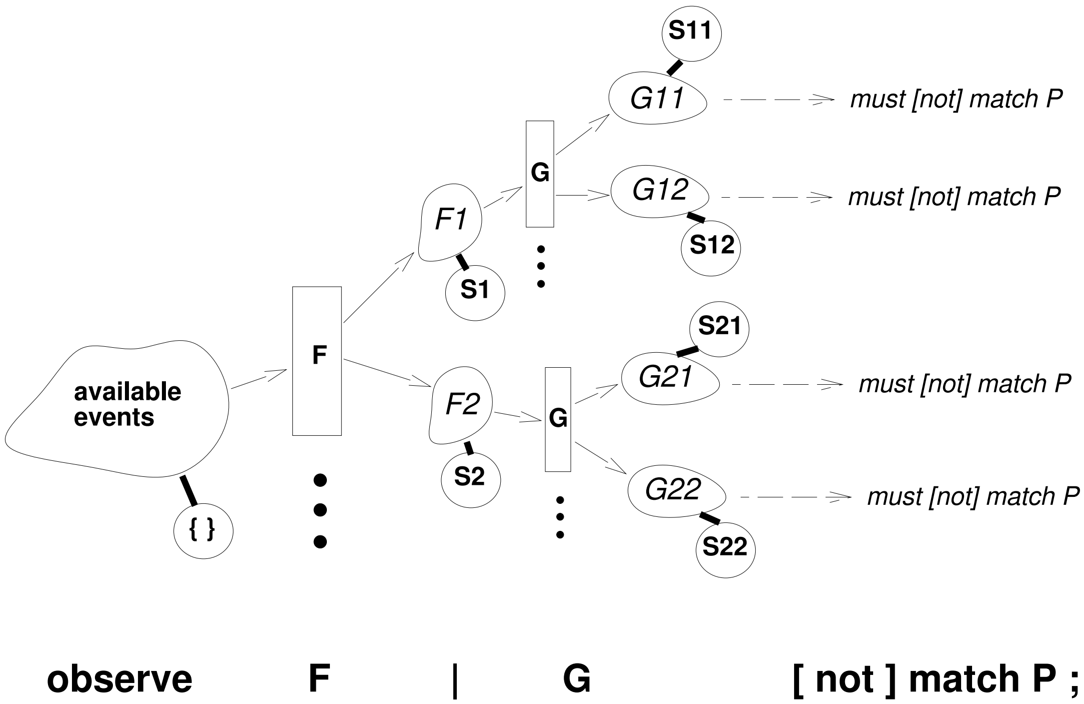

Figure 1.1: The Semantics of Constraints

1.2 Labels

The occurrence of a labeled constraint declares the associated label. The scope of the label extends from the point of declaration of the label to the end of the enclosing declarative region.

Labels may not be overloaded.

1.3 Filters

The filter part of a constraint allows the set of available events, potentially a very large and unwieldy set, to be pared down into many smaller subsets. These smaller computations are then much easier to constrain.

There are three basic variants of filters: pattern filters, alphabet filters, and map filters. Every filter observes one computation/substitution pair, and produces zero or more such pairs.

The filter part of a constraint,

$$F_1 \mid F_2 \mid \ldots \mid F_k$$

denotes a number of computations and substitution values determined as follows. One instance of the leftmost filter, F_1, is created; it observes the computation consisting of the events available to the module or map in which the constraint appears, and produces computation/substitution pairs. For $i > 1$, n instances of filter F_i are created, one for each or the n computations produced by the filter instances created for F_{i-1}.

Thus, the computation/substitution pairs produced by the filter part are all those pairs produced by instances of filter F_n.

Strictly speaking, a filter observes and produces computation/substitution pairs. However, when the substitution values of placeholders are not important, we will for simplicity say that a filter observes and produces computations.

Computation Type. For every computation available to a constraint there is an associated *type*, identifying which events and entities are nameable in the constraint. The type of the computation observed by a constraint in an interface type definition is the interface type being declared. The computation observed by a constraint located in a module has the implementation type of that module as its type.

Which events are available to a constraint is determined by the type of the computation that it observes. Only events associated with entities of the type are visible – i.e. events generated by the object itself through invocations of interface constituents, events generated by the environment of the object through invocations of interface constituents, and events generated by constituents of the interface through invocations of other visible entities.

The notion of a computation's *type* carries over to filters, as well.

We associate two types with a given filter: The *observed* type – the type of the computation it observes, and the *generated* type – the type of the computations it generates.

In general the generated and the observed types are the same, and identical to the type of the computation observed (and constrained) by the constraint. The only exception is the use of a map filter (see Section 1.3.3) with explicitly given domain and range types.

1.3.1 Pattern Filters

The type of a pattern filter's generated computation is equivalent to the type of its observed computation.

A *pattern filter* instance observes a computation/substitution pairs, and produces any number of such pairs. The filtering is based on the pattern language — every computation that is produced by an instance of a pattern filter is a match for the pattern associated with the filter.

Pattern filtering always produces subsets of the input set; that is, if the filters in Figure 1.1 are both pattern filters, the following relations hold: $G11 \subseteq F1$, $G12 \subseteq F1$, $G21 \subseteq F2$, $G22 \subseteq F2$, and $F1$ and $F2$ are subsets of the available events.

Syntax

filter ::= *pattern*

Semantics

An instance of a pattern filter P observes a pair (C, S), and produces all *unique, maximal* pairs (C', S'), where C' that matches P, $S' = S \cup S''$, and S'' is the set of substitution values that were determined in matching C'.

Each pair is unique and maximal, meaning there do not exist two pairs produced by a filter instance (C_1, S) and (C_2, S) such that $C_1 \subseteq C_2$.

Note that, unlike the reactive constructs of the RAPIDE 1.0 Executable Language Reference Manual, where events may be matched once and are then considered unavailable, a single event may occur in several different computations produced by a pattern filter.

Every computation produced by an instance of a pattern filter is always a match for the pattern given in the filter.

If an existential placeholder is defined at the outermost level of a filter pattern, the scope of that placeholder includes the subsequent filter patterns, as well as the pattern following **match**.

The set of placeholder substitution values determined for an outermost placeholder scope in the production of a subcomputation is passed along with the subcomputation, where it may be used by subsequent filters and body components.

Example:

observe Withdraw $\longrightarrow$ Balance | ...

Commentary:

The filter in this filter part may produce several pairs. Each computation in the pairs consists of a Withdraw and a Balance event that follows it. Because the filter has no outermost placeholder scope, the substitution value set of each pair is empty.

Consider the computation shown in Figure 1.2; assume it shows the events available to the constraint. This is the computation observed by the filter. Computations produced by the filter are shown in the dotted lines — three Withdraw–Balance pairs.

□

Example:

observe (Withdraw $\longrightarrow$ Balance) | Withdraw | ...

Commentary:

This filter part contains two filters; the filter of the previous example, followed by a pattern filter which is simply the basic pattern Withdraw.

The filter observes the two-event computations produced by the first filter, and produces one-event subcomputations, each of which is a Withdraw event — namely, the Withdraw events from the pairs of events it observed.

Consider the computation shown in Figure 1.2. There are two computations resulting from the entire filter; both are single Withdraw events, shaded in the figure.

□

Example:

observe [$*$ **rel** $\sim$] ((Task $\triangleright$ Subtask)) ...

Commentary:

This pattern filter produces all the subsets of the observed computation that consist of a Task event immediately followed by zero or more Subtask events.

□

[1] **An example using placeholders values passed along.**

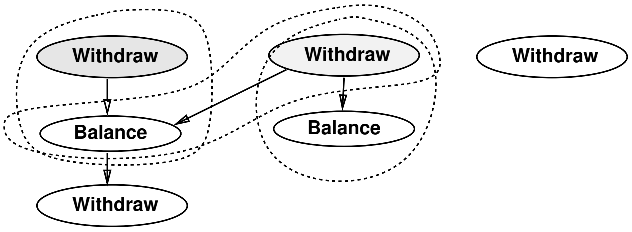

Figure 1.2: A pattern filter

1.3.2 Alphabet Filters

While pattern filters provide a general, expressive way to define subcomputations of interest, a user often simply wants to filter out events by action name, so that only events from a small set of names are produced by the filter. RAPIDE provides a syntactic shorthand notation for this called *alphabet filters*.

Syntax

filter ::=
    from basic_pattern_list

Semantics

The type of an alphabet filter's generated computation is equivalent to the type of its observed computation.

Semantically, the filter
    from P_1, P_2, ..., P_n
is equivalent to the more unwieldy filter
    [$*$ **rel** $\sim$] ((P_1 **or** P_2 **or** ... **or** P_n))

The list following the keyword **from** is a list of basic patterns. For each of its observed computations, an alphabet filter always produces a single computation that consists of all events of the input computation that match one of the basic patterns in the list; associated with it is a placeholder substitution set that is equivalent to the substitution set it observed. This is illustrated in Figure 1.3.

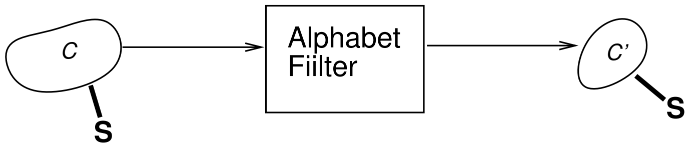

Figure 1.3: An alphabet filter

By its definition, an alphabet filter instance always produces exactly one computation/substitution pair. The computation consists of all observed events matching one of the basic patterns in the list; if there are no such events, the empty computation is produced. Because placeholders are not defined in an alphabet filter, the single substitution set produced is the same as the observed substitution set.

Example:

observe from Send, Receive | ...

Commentary:

This alphabet filter produces a single computation, consisting of all the available Send and Receive events.

□

Example:

observe from Producer.Send, Consumer.Receive | ...

Commentary:

This alphabet filter produces a single computation, consisting of all the available Send events that were owned by Producer and Receive events that were owned by Consumer.

□

1.3.3 Map Filters

Map filters are maps that observe an incoming computation, transform it, and generate an outgoing computation. Maps are defined in Section 5 of the RAPIDE 1.0 Architectures Reference Manual.

Syntax

filter ::= [**map** identifier **is**] { map_generator_expression | map_literal }

Semantics

The type of a map filter's generated computation is that of the map's range type, and only those events that are nameable in the domain type are observable to the map filter. Note that this provides an implicit filtering of the observed events when the domain type of a map is a true supertype of the type of the observed computation: an event placeholder in a map filter will never be bound to events that are not constituents of its domain type.

The scope of the optional identifier introduced by a map filter is the remainder of the constraint following the map, and it denotes the map. References to it are meaningful only in those succeeding filters that observe (wholly or in part) the computation generated by the map (i.e. only in those filters preceding the next map filter with a different range type, if any).

The triggers of the map's rules have direct visibility to the constituents of the observed computation.

Filter type compatibility

A map filter must have exactly one domain, which must be a supertype of the type of its observed computation. If the domain type of the map is a strict supertype of the type of the observed computation then only those events that are nameable in this supertype are observed (cfr. above).

Execution of map filters

A map filter represents an implicit evaluation of the map expression (be it a call to a map generator or just a map literal). The domain of the map is an object representing the observed computation. (*I.e.,* for the first map filter in the filter part, the module in which the constraint appears; and for subsequent filters, the generated computation from the previous filter.

A map executing as a filter is initialized (*i.e.* its **initial** part is called) when the observed computation is provided to the map. When the last event of a particular computation has been provided (*i.e.* the observed computation is *terminated*) and handled by the map's transition rules, the **final** part is called. When the **final** part terminates then the generated computation terminates as well.

The map instantiated as a filter creates a computation of the map's range type. Using transition rules, maps recognize conditions under which to add events to the generated computation. The causal and time relationships between the events of the generated computation are induced by the observed computation and the form of the rules (see the RAPIDE 1.0 Architectures Reference Manual).

Utility of map filters

Map filters serve diverse purposes. They are indispensable if a filter needs to accumulate information before deciding whether events are to be passed through or not, or when such accumulated information is required to make a subsequent determination of whether a computation is saitsfactory or not (as we shall see in the example below).

Map filters also serve as useful abstraction tools (as they do in the definition of architectures), to consolidate sets of events, etc.

Examples

Example: *Subtasks*

```
type Task_Master is interface
    in action Task( Size : Integer );
    out action Subtask( Size : Integer );
constraint
    type Task_Sum_Type is
        interface
            action out Task( Size : Integer );
            action out Task_Done( Size : Integer );
        end interface;

    map Sum_Tasks() from Comp : Task_Master to Task_Sum_Type is
        Sum : var Integer := −1;
```

```
        rule
            (?I in Integer)Comp.Task(?I) ⇒
                -- Output previous sum if not first time
                if $Sum /= −1 then
                    Task_Done($Sum);
                    Sum := 0;
                end if;
                Task(?I);;

            (?I in Integer)Comp.Subtask(?I) ⇒ Sum := Sum + ?I;;
        final
            Task_Done($Sum); -- Output final sum
        end;

        observe Sum_Tasks()
        match (?I in Integer) [ ∗ rel −→] ((Task(?I) −→ Task_Done(?I)));
    interface;
```

Commentary:

The constraint to be checked for conformance is that given an interface that observes Task events and produces Subtask events, to require that each Task event be followed immediately by zero or more independent Subtask events, with the further restriction that the sum of the parameters of the Subtask events must be the same as the parameter of the preceding Task event.

In order to specify this, we have to have (1) some means of accumulating information, and (2) to use this information in checking.

The former is achieved by using a map filter with a local state which captures precisely the required information about the computation, and the latter is achieved by letting this map produce events at the appropriate points, reporting on the state.

First a new type, Task_Sum_Type, is defined as the type produced by the map filter; it defines Task events (type-equivalent to the incoming Task events) and Task_Done events. The latter indicate the completion of a task, and the resources used to complete it. The map filter, Sum_Tasks, has a domain that defines the type of the surrounding interface type (so the map observes only the interface events available to the module containing the the constraint), and a range of the new interface type (so the map produces Task and Task_Done events).

To check the constraint, a map is created. A variable called Sum keeps track of the sums of the incoming Subtask event parameters. When a Task event is observed by the map, it generates an equivalent event (essentially propagating the event). When a Subtask event is observed, its parameter value is added to Sum. At the end of a Task–Subtask sequence, the map generates the event Task_Done, reporting the amount of resources consumed by the task.

The ordering induced by maps should mean that the Task_Done event follows the Task event in the dependency ordering.

As a result, pairs of Task and Task_Done events are produced. The final pattern of the constraint defines what these pairs must look like: a Task event followed by a Task_Done event, where both have the same parameter value.

□

1.4 Constraint Bodies

The filter part of a pattern constraint creates a context, namely zero or more computation/substitution pairs. In the body of a constraint, a computation in this context may be constrained by either requiring it to take or prohibiting it from taking a certain form.

The body of a constraint may contain any number of components. In each case, the component observes the pairs produced by the filter part of the constraint. If any of these pairs causes a *violation*, then the entire constraint is said to be violated.

A constraint body component may be: a **match** component, which is the fundamental body component; a **never** component, which is a useful syntactic shorthand for a more complex **match** component; or a nested constraint.

1.4.1 Body Component Labels

A body component may always begin with a label; the scope of the label extends to the end of the component. The label of a component is not visible in the declarative region in which the constraint occurs, and therefore may not be used, e.g., in inheritance. As such, a component label has no semantic effect on the constraint. A label may, however, be used by the toolset in the event of a violation to indicate that this is the body component that caused the constraint to violate.

1.4.2 Match components

Syntax

```
constraint_body_component ::=
        [ label ] [ not ] match pattern_expression ';'
```

Semantics

The pattern following **match** must be consistent with the range of the last filter in the filter part; that is, it must be a legal pattern in the context of the type of the computation denoted by the last filter.

First, the substitution values of the computation/substitution pair are applied to the pattern.

If the keyword **not** does not appear, the constraint component is said to be *positive*. A positive component violates if one of the computations it observes violates; a computation violates a positive component if the computation does not form an *exact* match for the pattern expression. In an exact match, every event in the computation is used in the matching of the computation to the pattern.

If the keyword **not** does appear, the constraint component is said to be *negative*. A negative component violates if one of the computations it observes violates; a computation violates a negative component if the computation forms an exact match for the pattern expression.

Example:

observe from Issue_Check **match** [∗ **rel** ⟶] (Issue_Check) **end**;

Commentary:

Assume this constraint applies to some module that observes and generates banking events. Events of many different actions will be available to constraints in that module. Because of the alphabet filter, only Issue_Check events are considered to be of interest for this constraint; there may also be Withdraw, Deposit, and other events, but they are not constrained by this example.

The subcomputation of available events that consists solely of the visible Issue_Check events must match the pattern: that computation must be a total order of Issue_Check events. If any two visible Issue_Check events were independent, this constraint would be violated.

Consider Figure 1.4, showing the Issue_Check events of a visible computation. This violates the constraint, as the events do not form a linear dependency order.

□

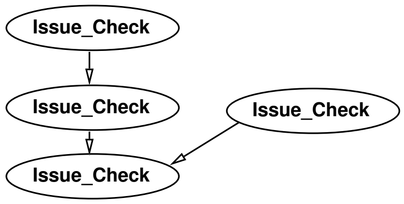

Figure 1.4: Issue_Check events of a visible computation

1.4.3 Never components

Syntax

```
constraint_body_component ::=
        [ label ] never pattern_expression ';'
```

Semantics

A **never** component violates if one of the computations it observes contains a match for the pattern expression.

A **never** component is a syntactic shorthand for a negative constraint component:

never P;

is semantically equivalent to

not match P $\sim$ ([$*$ **rel** $\sim$] (@));

The pattern expression above will be matched by any computation containing a match for P plus any number of other events.

1.4.4 Nested Constraints

The context created by a filter part may be sufficient for some of the components of a constraint. However, it is also useful to filter the computations created by this context even further. In such cases a nested constraint may be used as a body component.

Syntax

```
constraint_body_item ::=
        pattern_constraint
```

Semantics

A nested constraint component has the same semantics as a constraint that is not nested, except for the computation/substitution pairs it observes. Whereas a non-nested constraint observes the single computation of events available to the module or map in which it occurs and the empty substitution set, a nested constraint observes all pairs produced by the filter part of the constraint in which it is nested. If any of these computations violates, the constraint is violated.

A constraint with only a nested component is equivalent to a constraint with no nested component:

  observe *F_1* | *F_2* | ... | *F_n*
    observe *G_1* | *G_2* | ... | *G_m*
      <<body>>
    end;
  end;

is semantically equivalent to:

  observe *F_1* | *F_2* | ... | *F_n* | *G_1* | *G_2* | ... | *G_m*
    <<body>>
  end;

This rewriting, however, does not take into account labeling of the constraint or its components.

1.5 Implicit Filtering

A simple constraint body component may appear by itself as a constraint, in which case the filter part is implicit.

1.5.1 Syntax

```
pattern_constraint ::=
        simple_constraint_body_item ';'
```

1.5.2 Semantics

The implicit filter of a constraint with no filter part is the pattern filter

  observe [$*$ **rel** ~] (@)

That is, the constraint

  <<simple_body_item>>

is equivalent to

  observe Any
    <<simple_body_item>>
  end observe;

1.6 Examples

Example:

```
observe from Withdraw, Balance
    match [ * rel ~] (( (?Ac in Accnt) Withdraw(A is ?Ac) ⟶
                                       Balance(A is ?Ac) ));
end;
```

Commentary:

In this constraint, the computation produced by the filter consists of all visible Withdraw and Balance events. The constraint requires that this computation be partitionable into zero or more parts, where: each part consists of one Withdraw event and one Balance event, both events must have the same A component, and the Withdraw must precede the Balance in the dependency ordering.

Figure 1.5 shows the visible computation for this constraint with respect to some module. The shaded area shows the single filtered computation that applies to this constraint. The constraint is satisfied because the computation can be partitioned (as shown by the dotted areas) into subcomputations that match the given pattern.

An obvious interpretation of this constraint is that a Balance event is generated in response to a Withdraw event, and that is the only time a Balance is generated (there can be no "extra" Balance events). This may be more restrictive than is desired; a more realistic requirement might be that a Balance is always generated in response to a Withdraw, but Balance events may also occur in other situations as well. Here is a modification of the above constraint to meet this new requirement:

```
observe from Withdraw, Balance
    match [ * rel ~] (( (?Ac in Accnt) Withdraw(A is ?Ac) ⟶
                                       Balance(A is ?Ac) ))
          ~ [ * rel ~] (Balance);
end;
```

The right-hand argument of the binary "~" operator will match all extraneous Balance events; it can be thought of as "soaking up" the extra events.

□

Example:

```
observe (?D in Dollars) Withdraw(Dollar is ?D)
    match Withdraw where ?D < Max_Withdrawal;
end;
```

Commentary:

This constraint requires that all Withdraw events in the visible computation have Dollar components less than the value Max_Withdrawal. The filter produces a set of computations, the singleton computations of Withdraw events. Each of these computations must match the guarded pattern; that is, each must consist of a single Withdraw event with a component satisfying the guard. The substitution value of ?D derived from matching a Withdraw event is used in the body of the constraint.

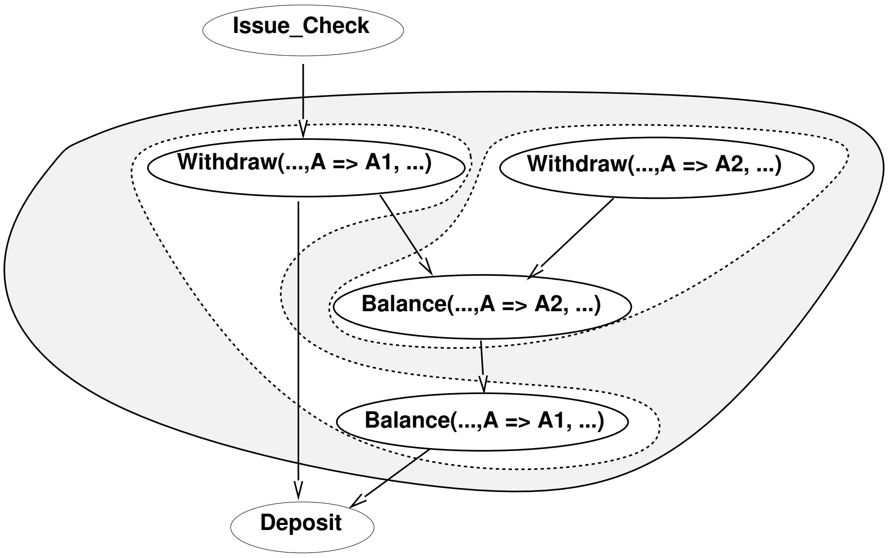

Figure 1.5: A banking computation

□

Example:

> **observe from** Withdraw
> **match** [$*$ **rel** $\sim$] (((?D **in** Dollars)
> Withdraw(Dollar **is** ?D) **where** ?D $<$ Max_Withdrawal));
> **end**;

Commentary:

Here we see the difference between a pattern filter and an alphabet filter. This constraint is equivalent to the previous example. The the previous constraint examined and constrained Withdraw events singly; the filter produces as many computations as there are Withdraw events. This filter produces a single computation consisting of all Withdraw events; then, in the body of this constraint, the pattern requires that all such events must must not exceed Max_Withdrawal.

□

Example:

> **observe** (?Ac **in** Accnt) [$*$ **rel** $\sim$] (Withdraw(A **is** ?Ac))
> **match** [$*$ **rel** $\longrightarrow$] (Withdraw);
> **end**;

Commentary:

In this constraint, all Withdraw events of a single account must be ordered in the dependency ordering. Here the filter produces a set of computations, each of which consists of all the Withdraw events with the same A component. Each of these computations must form a linear ordering.

□

Example:

$$\textbf{never}\ (?\text{Ac}\ \textbf{in}\ \text{Accnt})\ \text{Withdraw}(\text{A}\ \textbf{is}\ ?\text{Ac})\ \|\ \text{Withdraw}(\text{A}\ \textbf{is}\ ?\text{Ac})\ ;$$

Commentary:

This constraint has the same meaning as the previous one, using the **never** syntax and implicit filtering. So this constraint states that there should never be independent pairs of events with the same A parameter value.

□

Example:

$$\begin{array}{l} \textbf{observe}\ (\text{Call} \longrightarrow \text{Call}) \sim [\ *\ \textbf{rel} \sim]\ (\text{Ack}) \\ \textbf{match}\ (?\text{I},\ ?\text{J}\ \textbf{in}\ \text{Integer}) \\ \qquad\qquad (\ \text{Call}(?\text{I}) \longrightarrow \text{Call}(?\text{J})\ ) \sim \\ \qquad\qquad (\ \text{Ack}(?\text{I}) \longrightarrow \text{Ack}(?\text{J})\ ) \sim [\ *\ \textbf{rel} \sim]\ (\text{Ack}); \end{array}$$

Commentary:

This constraint examines computations consisting of a pair of ordered Call events, and all Ack events. It requires that, among the Ack events, there must be two with the same parameter, and in the same dependency order, as the Calls. The final [$*$ **rel** $\sim$] (Ack) matches all Ack events not matched by the rest of the body pattern.

□

Example:

$$\textbf{never}\ \text{Issue\_Check}\ \|\ \text{Issue\_Check};$$

Commentary:

The constraint requires that no two visible Issue_Check events be unordered. Figure 1.4 contains two violations of this constraint: there are two pairs of Issue_Check events that are not ordered.

□

Example:

$$\begin{array}{l} \textbf{observe}\ \text{Task} \rhd [\ *\ \textbf{rel} \sim]\ (\text{Subtask}) \\ \textbf{match}\ (?\text{I}\ \textbf{in}\ \text{Integer})\ \text{Task}(?\text{I}) \rhd [\ 1..?\text{I}\ \textbf{rel} \sim]\ (\text{Subtask}); \end{array}$$

Commentary:

This constraint is similar to the example of the previous section. In this case, a Task event has an integer parameter, and is immediately followed by a number of Subtask events equal to that parameter value.

The pattern filter produces all the subsets of the observed computation that consist of a Task event immediately followed by zero or more Subtask events. Each of these subsets must be a match for the subsequent pattern, meaning the number of Subtask events must be equal to the parameter of the Task.

□

1.7 Conformance to Pattern Constraints

A pattern constraint constrains only those events that are available to it. Every module has associated with it a set of available events generated by a RAPIDE-1.0 execution. The RAPIDE 1.0 Executable Language Reference Manual describes the events available to a module. [2]

Constraints in a map?

- For a pattern constraint occurring in an interface definition (see the RAPIDE 1.0 Types Language Reference Manual), each module of the interface type must satisfy that constraint.
- For a pattern constraint occurring in a module constructor (see the RAPIDE 1.0 Executable Language Reference Manual), each module created by a call to the constructor must satisfy that constraint; for a pattern constraint in a module literal, the module must satisfy the constraint.
- For a pattern constraint occurring in a map constructor or map literal (see the RAPIDE 1.0 Architectures Reference Manual), each map created by a call to the constructor or literal must satisfy the constraint. The visible computation with respect to each map consists of the visible computation of the domains of the map; the visible computation of locally declared constituents of the map; and the visible computation of the range of the map.

Part II

Constraints for Sequential RAPIDE-1.0

Chapter 2

Sequential Constraints

2.1 Introduction

The sequential behavior of Rapide-1.0 programs may be constrained by *sequential constraints*, which require certain boolean expressions to be true at various states of objects in a program.

The states of an object are defined in Section 2.2. A simple extension of Rapide-1.0 expressions for sequential constraints is given in Section 2.3. Constraints are then defined for various Rapide-1.0 constructs: formal parameters (Section 2.4), type expressions (Section 2.5), objects (Section 2.6), and statements (Section 2.7).

2.2 Object State

▬[3] **TBD, especially the meaning of "object state" in a module with several concurrent threads of control.**

2.3 Constraint Expressions

Sequential constraints use the expression syntax and semantics defined in the rapide 1.0 Executable Language Reference Manual, and extend it with a single modifier: **in**.

2.3.1 Syntax

The syntax of Constraint Expressions is

constraint_expression	::=	... *executable_language_expression* ... |
		in *constraint_expression*
constraint_expression_list	::=	*constraint_expression* { ',' *constraint_expression* }

2.3.2 Informal Semantics

Executable language expressions have the semantics described in the rapide 1.0 Executable Language Reference Manual.

Some sequential constraints are defined for constructs that execute over multiple object states. If an expression is not modified by **in**, it refers to the value of the expression at the *checking state* of the object. If an expression is modified by **in**, it refers to the value of the expression at the *initial state* of the object.

Initial and checking object states are defined for each constraint below.

2.4 Parameter Constraints

The object parameters of a parameter list may be constrained using parameter constraints.

2.4.1 Syntax

The syntax of parameter constraints is:

formal_parameter_list ::=
    formal_parameter { ';' *formal_parameter* } [**constraint** *constraint_expression_list*]

patt_macro_parameter_list ::=
    patt_macro_parameter { ';' *patt_macro_parameter* } [**constraint** *constraint_expression_list*]

2.4.2 Informal Semantics

The expressions in the *constraint_expression_list* must be object expressions of Boolean type.

If the parameter list occurs in the definition of a construct that is called at run-time, such as a function type, function object, or module constructor, then for each call the checking state is the state of the object at the time of the call.

If the parameter list occurs in the definition of a construct that is elaborated at run-time, such as a type constructor, the checking state is the state of the object containing the construct when it is elaborated.

All expressions must be True. Use of the **in** modifier has no effect in a parameter constraint; that is, the initial state is considered equivalent to the checking state.

2.4.3 Examples

Example:

```
function /( Num, Denom : Integer
            constraint Denom /= 0 ) return Integer;
```

Commentary:

This constraint on the parameters of the function "/" requires that the value of the Denom parameter be non-zero.

□

Example:

```
type Range( type T <: Discrete(T);
            Lower_Bound, Upper_Bound : T;
            constraint Lower_Bound <= Upper_Bound );
```

Commentary:

Here we are defining a polymorphic type constructor Range that takes two values of the parameter type as parameters, and defines a required relationship between them: the Lower_Bound may not exceed the Upper_Bound.

This constraint will be checked for all applications of Range at the time the construct containing the type expression is elaborated.

□

2.5 Type Expression Constraints

A type expression constraint allows restrictions to be defined for a particular type which must then be satisfied by every object of that type.

Interface types are constrained with an axiom list, and function types are constrained with a function constraint.

2.5.1 Interface Axioms

Interface types may include have a **constraint** part in which algebraic expressions may be defined. These expressions constrain the visible states of object of the interface type.

Syntax

The syntax of interface axioms is:

interface_constraint_part ::= { *constraint* }

constraint ::= *algebraic_constraint*

algebraic_constraint ::= [*label*] **axiom** *algebraic_expression*

Informal Semantics

An algebraic constraint is an algebraic expression that must hold over all *visible states* of all objects of the interface type. The visible states of an object are the object states where the object generates an event of an interface action.

At each visible state, the algebraic expressions are evaluated. The checking state is the current state of the object, and the initial state is the initial state of the object. Thus, an expression modified by **in** in an algebraic constraint refers to the value of the expression in the initial state of the object.

In particular, if the object is a module with an **initial** part, its initial state is the state of the object when the statements of the **initial** part have been executed.

Examples

Example:

```
type Stack_Type( type Elem ) is interface
    type Stack;
    function Empty() return Stack;
    function Push( S : Stack; E : Elem ) return Stack;
    function Pop( S : Stack
        constraint Length(S) /= 0 ) return Stack;
    function Top( S : Stack ) return Elem;
    function Length( S : Stack ) return Natural;
    function =( S1, S2 : Stack) ) return Boolean;
constraint
    axiom for all S : Stack; E : Elem =>
        Top( Push(S, E) ) = E,
        Pop( Push(S, E) ) = S,
        Length( Empty() ) = 0,
        Length( Push(S, E) ) = Length(S) + 1;
end;
```

Commentary:

This type constructor provides a resource for creating stacks. The axioms show some of the relationships between functions defined in the interface. Note that the axioms do not define what happens when Pop is called on an empty stack, but the parameter constraint of Pop disallows this call.

□

Example:

```
type Even is interface
    include Integer;
constraint
    axiom self mod 2 = 0;
end;
```

Commentary:

The type Even is an Integer type that has the further restriction that all values of Even objects must be even.

□

2.5.2 Function Constraints

A function constraint may be defined for a function type expression, and constrains all calls to objects of the function type. In particular, the value returned by the function may be constrained, and exception conditions may be defined.

Syntax

The syntax of function return constraints is:

function_type_expression ::=
  function '(' [*formal_parameter_list*] ')' [**return** *type_expression*]
    [**constraint** { *function_constraint* }]

function_constraint ::= *function_return_constraint*
  function_exception_precondition
  function_exception_postcondition

function_return_constraint ::= [**return** *identifier* '=>'] *constraint_expression_list* ';'

function_exception_precondition ::= *constraint_expression* '=>' **raise** *pattern* ';'

function_exception_postcondition ::= **raise** *pattern* ['=>' *constraint_expression*] ';'

Informal Semantics

In a function return constraint, the optional identifier, if it exists, is called the *return object*. An occurrence of it in the *constraint_expression_list* is an object expression, whose type is the return type of the function; its scope is the constraint_expression_list. The return object may not have the same name as any of the object parameters of the function type.

The constraint_expression_list of a function return constraint, if it exists, must consist of object expressions of type Boolean. A function return constraint is satisfied if all constraint expressions are True for each call to all objects of the function type that return normally. The initial state for the constraint is the object state at the time of the call. The checking state is the object state at the time of the return.

Instances of the return object in the constraint expressions represent the value returned by the function.

The constraint_expression in a function_exception_precondition must be an object expression of type Boolean; the pattern must be a basic pattern denoting an exception, which may possibly be guarded (see Patterns LRM, Section 2.4.11). The initial state and the checking state of the constraint expression are both the object state at the time of the call. If the expression has the value True, a function exception precondition is satisfied if an exception is generated that is matched by the given pattern.

The pattern in a function_exception_postcondition must be a basic pattern denoting an exception, which may possibly be guarded; the constraint_expression, if it exists, must be an object expression of type Boolean. The initial state for the constraint is the object state at the time of the call. The checking state is the object state at the time the exception is raised. If execution of the function causes an exception to be raised that matches the pattern, a function exception postcondition is satisfied if the constraint_expression is True.

Example:

```
function Square( N : Integer ) return Integer
    constraint Rtn => Rtn = N * N;
```

Commentary:

This constraint requires all calls to the function to return the square of their parameter N.

□

Example:

```
function Swap( type T;
               Left, Right : Ref(T) )
    constraint
        $Left = in $Right,
        $Right = in $Left;
```

Commentary:

Here we are constraining the values of the parameters at the time the function returns.

□

▬[4] **Exception examples**

2.6 Object Constraints

Whereas type constraints hold for all objects of a type, object constraints allow individual objects to be constrained.

2.6.1 Syntax

The syntax of object constraints is:

object_declaration ::=
  executable_language_object_declaration [**constraint** *constraint_expression_list*] ';'
object_name_declaration ::=
  basic_name_list ':' *type_expression* [**constraint** *constraint_expression_list*] ';'

2.6.2 Informal Semantics

The *constraint_expression_list* must be a list of Boolean expressions.

If the object is of interface type, each expression must be True in all visible states of the object. The initial state is the initial state of the object, and the checking state is the current state of the object (see Section 2.5.1).

If the object is of a function type, each expression must be True for each call to the function. The initial state for the constraint is the object state at the time of the call; the checking state is the object state at the time of the return (see Section 2.5.2).

2.6.3 Examples

Example:

```
Even_Int : var Integer := 0
    constraint $Even_Int mod 2 = 0;
```

Commentary:

In all of its states, this constraint requires that Even_Int have an even value. In particular, an assignment of an odd number to Even_Int will violate the constraint in the visible state denoted by the ":='Return" event generated by Even_Int.

□

Example:

```
My_Airplane : Airplane is Airplane_Generator()
    constraint My_Airplane.Altitude() > Sea_Level;
```

Commentary:

This constraint states that a call to the Altitude function of the object My_Airplane would return a value greater than Sea_Level in any visible state of that object.

□

2.7 Statement Constraints

Any statement may be constrained, allowing the state of the object in which it occurs to be constrained immediately before and after execution of the statement.

2.7.1 Syntax

The syntax of statement constraints is

statement ::= *executable_language_statement* [**constraint** *constraint_expression_list*] ';'

2.7.2 Informal Semantics

The *constraint_expression_list* must be a list of Boolean expressions. Each expression must be True each time the statement is executed.

The initial state for the constraint is the object state immediately before it executes the statement, before any expressions in the statement have been evaluated. The checking state is the object state immediately after the statement has completed execution normally.

There are cases when the execution of a statement does not allow a checking state to occur: **return** statements, **raise** statements, other statements that cause an exception to be raised, and **assert** statments that fail. Constraints on these statements have no effect.

2.7.3 Examples

Example:

```
X := $X + 1
    constraint X = in $X + 1;
```

□

Example:

```
for I : Integer in 1 .. N do
    Total := $Total + I;
end for
    constraint $Total = (in $Total) + (N*(N+1))/2;
```

Commentary:

This simple for loop adds the sum of 1 to N to the Ref object Total. The constraint means that the value of Total after execution of the loop is (N*(N+1))/2 greater than it was before execution of the loop;

□

Glossary

constraint, pattern a description of the behavior of Rapide-1.0 programs that denotes acceptable and unacceptable patterns of events. See page 7.

Index