DRAFT
Rapide 1.0 Pattern Language Reference Manual

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

July 18, 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 the RAPIDE overview given in the Guide to Rapide, particularly Chapter 3 on events and posets. It describes the event patterns of RAPIDE-1.0. Event patterns are a fundamental part of the reactive programming constructs of RAPIDE (i.e., *processes* in the Executable LRM, and interface *behaviors* and architecture *connections* in the Architectures LRM), and also *constraints* in the Constraint LRM.

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

Contents

List of Figures

Chapter 1

Computations and the Theory of Patterns

1.1 Introduction

This chapter introduces the basic concepts and operations of RAPIDE computations. Then the RAPIDE pattern language is defined in terms of these operations.

1.2 Computations

Computations consist of *events*, which are uniquely identifiable tuples of values. The Event() type is defined in the Predefined Types LRM.

The *identity* relation $==$ is a congruence relation; that is, it satisfies the *equivalence axioms*:

$$\begin{array}{ll}
(\forall \text{ event } e) e == e & \text{(reflexivity)} \\
(\forall \text{ event } e_1, e_2) e_1 == e_2 \quad \rightarrow \quad e_2 == e_1 & \text{(symmetry)} \\
(\forall \text{ event } e_1, e_2, e_3)(e_1 == e_2 \wedge e_2 == e_3) \quad \rightarrow \quad e_1 == e_3 & \text{(transitivity)}
\end{array}$$

as well as the *functional substitutivity axiom schema*, for every n-ary function f and every i from 1 to n:

$$\begin{array}{l}
(\forall \text{ event } e_1, e_2)(\forall z_1, z_2, ... z_n) e_1 == e_2 \quad \rightarrow \\
\quad f(z_1, ..., z_i, e_1, z_i + 1, ..., z_n) == f(z_1, ..., z_i, e_2, z_i + 1, ..., z_n) \quad \text{(functional substitutivity)}
\end{array}$$

A *computation* is a set of events, where $==$ is the equality operator on set elements.

The events in a computation have the preorder relation $\leq_c$ (causal preordering) and the preorder relations $\leq_t$ (temporal preordering, for each Clock t); and the equivalence relation, $=_c$ (causal equivalence) and the equivalence relations $=_t$ (temporal equivalence).

$=_c$ and $=_t$ are equivalence relations; that is, they satisfy the equivalence axioms of reflexivity, symmetry and transitivity (shown above). The relations $\leq_c$ and $=_c$ together satisfy the preorder axioms:

$$\begin{array}{ll}
(\forall \text{ event } e) e \leq_c e & \text{(reflexivity)} \\
(\forall \text{ event } e_1, e_2)(e_1 \leq_c e_2 \wedge e_2 \leq_c e_1) \quad \rightarrow \quad e_1 =_c e_2 & \text{(antisymmetry w.r.t. } =_c\text{)} \\
(\forall \text{ event } e_1, e_2, e_3)(e_1 \leq_c e_2 \wedge e_2 \leq_c e_3) \quad \rightarrow \quad e_1 \leq_c e_3 & \text{(transitivity)} \\
(\forall \text{ event } e_1, e_2, e_3)(e_1 =_c e_2 \quad \rightarrow \quad e_1 \leq_c e_3) \quad \leftrightarrow \quad e_2 \leq e_3 & \text{(left substitutivity)} \\
(\forall \text{ event } e_1, e_2, e_3)(e_1 =_c e_2 \quad \rightarrow \quad e_3 \leq_c e_1) \quad \leftrightarrow \quad e_3 \leq e_2 & \text{(right substitutivity)}
\end{array}$$

$\leq_t$ and $=_t$ satisfy the same axioms. [1] **Sigurd thinks there's a cleaner and simpler set of axioms than these.**

From these relations we derive the associated relations $<_c$ (causal ordering) and $<_t$ (temporal ordering). $<_c$ is defined in terms of $\leq_c$ and $=_c$:

$$\begin{array}{lcll} (\forall \text{ event } e_1, e_2) e_1 <_c e_2 & \leftrightarrow & (e_1 \leq_c e_2 \wedge \neg e_1 =_c e_2) & \text{(irreflexive restriction)} \\ (\forall \text{ event } e_1, e_2) e_1 \leq_c e_2 & \leftrightarrow & (e_1 <_c e_2 \vee \neg e_1 =_c e_2) & \text{(reflexive closure)} \end{array}$$

$<_t$ is defined in terms of $\leq_t$ and $=_t$ with the same axioms.

The equivalence relations of time and causality are consistent with the identity relation (this relationship is inferrable from the congruence of $==$):

$$\begin{array}{lcll} (\forall \text{ event } e_1, e_2) e_1 == e_2 & \rightarrow & e_1 =_c e_2 & \\ (\forall event e_1, e_2) e_1 == e_2 & \rightarrow & e_1 =_t e_2 & \text{(identity-equality consistency)} \end{array}$$

Causal ordering and temporal ordering have the following consistency relationship:

$$\begin{array}{lcll} (\forall \text{ event } e_1, e_2) e_1 <_t e_2 & \rightarrow & \neg(e_2 <_c e_1) & \\ (\forall event e_1, e_2) e_1 <_c e_2 & \rightarrow & \neg(t.finish(e_2) <_t t.start(e_1)) & \text{(causal-temporal consistency)} \end{array}$$

1.3 The Pattern Language

A Computation $\mathcal{C}$ is a set of events.

We use the notation $\mathcal{C} \models P$ to mean pattern P matched in the computation $\mathcal{C}$. The result of matching a pattern in a computation is a set of sets of events. Every such set will be a subset of $\mathcal{C}$.

The pattern language is defined formally in Figure 1.1.

$\mathcal{C} \models a$	$\equiv \{\{e\} \mid (e \in \mathcal{C}) \wedge (a \in \mathcal{B}) \wedge matches(e,a)\}$	Basic Patterns[a], 2.3
$\mathcal{C} \models p_1 \longrightarrow p_2$	$\equiv \{s_1 \cup s_2 \mid (s_1 \in (\mathcal{C} \models p_1)) \wedge (s_2 \in (\mathcal{C} \models p_2)) \wedge$ $(\forall e_1, e_2\ (e_1 \in s_1 \wedge e_2 \in s_2) \rightarrow e_1 <_c e_2)\}$	Sequence, 2.4.1
$\mathcal{C} \models p_1 \rhd p_2$	$\equiv \{s_1 \cup s_2 \mid (s_1 \in (\mathcal{C} \models p_1)) \wedge (s_2 \in (\mathcal{C} \models p_2)) \wedge$ $s_1 \cup s_2 \in (\mathcal{C} \models p_1 \longrightarrow p_2) \wedge$ $(\forall e_1, e_2\ (e_1 \in s_1 \wedge e_2 \in s_2) \rightarrow$ $\neg \exists w\ (w \in \mathcal{C}) \wedge (w \notin s_1 \cup s_2) \wedge$ $e_1 <_c w <_c e_2\}$	Immediate Sequence, 2.4.2
$\mathcal{C} \models p_1 \sim p_2$	$\equiv \{s_1 \cup s_2 \mid (s_1 \in (\mathcal{C} \models p_1)) \wedge (s_2 \in (\mathcal{C} \models p_2)) \wedge$ $(s_1 \cap s_2) = \emptyset\}$	Join, 2.4.3
$\mathcal{C} \models p_1 \Vert p_2$	$\equiv \{ s_1 \cup s_2 \mid (s_1 \in (\mathcal{C} \models p_1)) \wedge (s_2 \in (\mathcal{C} \models p_2)) \wedge$ $(\forall e_1, e_2\ (e_1 \in s_1 \wedge e_2 \in s_2) \rightarrow$ $\neg(e_1 <_c e_2 \vee e_2 <_c e_1))\}$	Independence, 2.4.4
$\mathcal{C} \models p_1$ **or** p_2	$\equiv (\mathcal{C} \models p_1) \cup (\mathcal{C} \models p_2)$	Disjunction, 2.4.5
$\mathcal{C} \models p_1$ **and** p_2	$\equiv (\mathcal{C} \models p_1) \cap (\mathcal{C} \models p_2)$	Conjunction, 2.4.6
$\mathcal{C} \models p_1 \cup p_2$	$\equiv \{s_1 \cup s_2 \mid (s_1 \in (\mathcal{C} \models p_1)) \wedge (s_2 \in (\mathcal{C} \models p_2))\}$	Union, 2.4.7
$\mathcal{C} \models p_1 \leftrightarrow p_2$	$\equiv \{s_1 \cup s_2 \mid (s_1 \in (\mathcal{C} \models p_1)) \wedge (s_2 \in (\mathcal{C} \models p_2)) \wedge$ $(\forall e_1, e_2\ (e_1 \in s_1 \wedge e_2 \in s_2) \rightarrow e_1 =_c e_2)$	Equivalence, **??**
$\mathcal{C} \models op(a_1, \ldots, a_n)$	$\equiv \mathcal{C} \models b_{op}(p_1, \ldots, f_n) \vert^{p_1,\ldots,f_n}_{a_1,\ldots,a_n}$	Pattern Macros [b], 2.5
$\mathcal{C} \models (id{:}t$ in it rel $op)f$	$\equiv if\ l_{it} = [v_1, v_2, ..., v_n]\ then$ $\mathcal{C} \models p \vert^{id}_{v_1}\ op\ p \vert^{id}_{v_2}\ op\ ...\ op\ p \vert^{id}_{v_n}$	Iteration [c], 2.4.8
$\mathcal{C} \models (id{:}t)p$	$\equiv (s \mid \exists v \in d_t(s \in \mathcal{C} \models p \vert^{id}_{v}))$	Placeholder Patterns [d], 2.4.9
$\mathcal{C} \models (p)$	$\equiv \mathcal{C} \models p$	Parenthesized Patterns, 2.4.10
$\mathcal{C} \models p$ where b	$\equiv if\ b\ then\ (\mathcal{C} \models p)\ else\ \{\}$	Guarded Patterns, 2.4.11
$\mathcal{C} \models p$ during (c, t_1, t_2)	$\equiv \mathcal{C} \models p \wedge (\exists e_1, e_2 \in \mathcal{C} \models p)(\forall e_3 \in \mathcal{C} \models p)$ $c.start(e_1) = t_1 \wedge$ $(c.start(e_1) <= c.start(e_3)) \wedge$ $c.finish(e_2) = t_2 \wedge$ $(c.finish(e_2) >= c.finish(e_3))$	Timing Operators [e], 2.6

[a] We assume the existence of a predicate $matches(a,e)$ which is true if a basic pattern $a \in \mathcal{B}$ matches an event $e \in \mathcal{C}$.

[b] $b_{op}(p_1, \ldots, f_n)$ is the pattern associated with the macro op, parameterized by its formal parameters.

[c] l_{it} is the set of objects denoted by it. Currently a bounded definition; need an unbounded one.

[d] d_t is the domain of type t; that is, the set of all values of t.

[e] $c.start(e)$ is the value of the Start function of clock c on the event e. $c.finish(e)$ is the value of the Finish function.

Figure 1.1: The Pattern Language

Chapter 2

Event Patterns

Event patterns are expressions that describe partial orders of events. A pattern acts as a "template" that may fit a partial order of events, if the events and the orderings between them satisfy the semantics of the pattern. The pattern language is used to describe *computations*. A computation is a set of events and the orderings between them.

A computation that fits a pattern is said to *match* the pattern.

A pattern is usually an incomplete description of a computation; there are several ways to leave "holes" in the description, which may be filled by any values. For example, *placeholders* are description holes such that all occurrences of a particular placeholder must be filled by the same value in a match.

Patterns are used three ways in RAPIDE: in *triggering* constructs, in *constraints*, and as *poset generator* patterns.

When occurring in a triggering construct—that is, a pattern matching statement of a process (see Executable LRM, Section 7.3), a behavior transition rule (see Architectures LRM, Section 2.6), a map rule (see Architectures LRM, Section 5.2.3), or as the left-hand side of a connection (see Architectures LRM, Section 3.4)—a pattern describes a subcomputation that is to be observed by the construct; once the construct observes that subcomputation, execution of the construct continues.

A pattern may also occur in a pattern constraint; see the Constraint LRM for the use of patterns in constraints.

Finally, a pattern may be used as a poset generator in a behavior (see Architectures LRM, Section 2.6), or as the right-hand side of a connection (see Architectures LRM, Section 3.4).

In this chapter, where the orderings of events in a computation are relevant and important, we will use figures to show the computation, such as in Figure 2.1. When such orderings are irrelevant, we will use simple set notation to give the set of events comprising the computation of interest. For example, { A, B, C } refers to a computation containing events A, B, and C, where the ordering of these events is not important.

2.1 Syntax

The syntax for patterns is

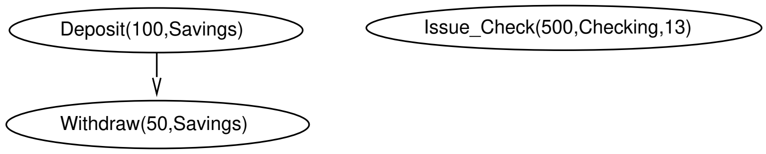

Figure 2.1: A computation

pattern ::= *basic_pattern* |
pattern *binary_pattern_operator* *pattern* |
iterator_decl *pattern* |
pattern **where** *guard* |
pholder_decl_list *pattern* |
'(' *pattern* ')' |
pattern_macro_instance

binary_pattern_operator ::= '$\longrightarrow$' | '$\triangleright$' | '$\sim$' | '$\|$' | **or** | **and** | '$\cup$'

iterator_decl ::= '[' [*ident* **in**] *iterator_exp* ']'

guard ::= *algebraic_constraint*

pholder_decl_list ::= '(' *pholder_decl* { ';' *pholder_decl* } ')'

pholder_decl ::= *identifier_list* ':' *type_expression*

The characters '->' may be used as a replacement for '$\longrightarrow$'; '|>' may be used as a replacement for '$\triangleright$'; and '**union**' may be used as a replacement for '$\cup$'.

The character '^' followed by the binary_pattern_operator may be used as a replacement for the superscripted binary_pattern_operator in pattern iteration.

Guards have higher precedence than the other pattern operators, all of which have the same precedence. All non-pattern operators have higher precedence than guards.

In patterns, as in all expressions, operators of higher precedence are associated with their operands before operators of lower precedence. For a sequence of operators with the same precedence level, the operators are left associative. Parentheses can be used to impose other associations.

2.2 Placeholders

Patterns may contain occurrences of *placeholders*. These are identifiers that act similar to free variables. Placeholders represent "holes" in the patterns, where any value that is of the placeholder type may fit. Placeholders are fully defined in Section 2.4.9.

2.2.1 Syntax

pattern ::= *pholder_decl_list* *pattern*

pholder_decl_list ::= '(' *pholder_decl* { ';' *pholder_decl* } ')'
pholder_decl ::= *identifier_list* : *type_expression*

A placeholder declaration is followed by a pattern, which defines its scope.

2.2.2 Semantics

An occurrence of the placeholder in a pattern is an object expression of the placeholder's declared type.

For a particular match of a pattern, a placeholder scope defines a textual area of the pattern for which all occurrences of a placeholder must be matched by the same value, which must be of the placeholder's type.

2.2.3 Examples

Example: *A basic pattern*

$$(?D : \text{Dollar}) \ \text{Deposit}(?D) \longrightarrow \text{Withdraw}(?D)$$

Commentary:

> This pattern defines the placeholder ?D to be of the type Dollar[1]. A match of this pattern must be a Deposit event followed (as required by the ⟶ pattern operator) by a Withdraw event, where both events must have the same parameter value.

□

2.3 Basic Patterns

The simplest patterns, *basic patterns*, are templates for a single event. Any match for such a pattern will be a computation consisting of one event.

2.3.1 Syntax

The syntax for a basic pattern is

basic_pattern ::= [*event_part*] '@' ['(' *parameter_association* ')']

basic_pattern ::= [*event_part* '@'] *action_part* ['(' *parameter_association* ')']

event_part ::= *expression*

action_part ::= [*expression* '.'] *ident*

[1] Stylistically, placeholders are represented in these LRMs as identifiers beginning with '?'. This style is recommended if the placeholders need to be distinguished from other identifiers in a pattern.

2.3.2 Semantics

The *event_part*, if it exists, is an expression of type **event**().

The *action_part* consists of an identifier, possibly preceded by an object expression, using the selection syntax.

Matching rules. A basic pattern

    exp@(<parameters>)

is matched by a computation consisting of a single event, { E }, if the following hold:

- The parameters of the pattern are *consistent* with the parameters of the event type: meaning, for every positional parameter in the pattern, the event has a parameter in that position consistent with the pattern parameter; and, for every named parameter in the pattern, the event has a parameter of the same name that is consistent with the pattern parameter.

  A pattern's parameter is consistent with an event's parameter if they are of the same kind (that is, if the event's parameter is an object parameter and the pattern's parameter is an object expression, and similarly with type parameters), and where the type associated with the event parameter is of the type associated with the pattern parameter expression.

  Note: The positions of the predefined parameters of all events (see the RAPIDE 1.0 Predefined Types and Objects Reference Manual) are considered not available. That is, named association must be used to denote the value of a predefined event parameter.

- Each object parameter of the pattern is equal (i.e. has the "=" relationship) to its associated constituent in the event. If the type of the parameter does not define "=" the identity relationship "==" (see Chapter 22 of the RAPIDE 1.0 Predefined Types and Objects Reference Manual) is used.

  Therefore, if a parameter association for a particular parameter is omitted in a basic pattern, it may have any value.

- E is an event object that is identical (has the identity relationship) to the object denoted by exp, if an event part is given.

The basic pattern "@" matches any single event in a computation.

For a discussion of when object expressions in patterns are evaluated, see Section 3.4.

Alternative Syntax. An alternative syntax is also available for basic patterns; it has further static restrictions.

The basic pattern

    exp@obj#id(<parameters>)

is matched by the computations that match

    exp@(<parameters>, Name **is** "id", Owner **is** obj)

The basic pattern

    exp@id(<parameters>)

is matched by the computations that match

    exp@(<parameters>, Name **is** "id")

That is, the identifier supplied in the action part is interpreted as the string literal value of the Name parameter; the object expression of the action part, if any, is interpreted as the value of the Owner parameter.

The further restrictions are:

- If the action part expresses both the name and the owner, the name must denote an action in the owner's type; if the action part expresses only the name, that name must denote an action that is directly visible.
- The rules for parameter association and overload resolution are the same as those for function applications, except that an association for any particular parameter may always be omitted.
- The event type inferred by the parameter association (or by the empty parameter association, if none is given) must be a subtype of the type of the event part, if one exists.

2.3.3 Examples

The following examples use a banking system, where each action name indicates the set of transactions it models — Balance events report the balance of an account, Issue_Check represents the issuing of a check drawn on a particular account, Withdraw represents the withdrawal of money from a particular account, and Deposit represents cash deposits. The sums, accounts and check numbers involved are identified by the parameters of the events.

The type of Balance, Withdraw and Deposit events is
    event(D: Dollar; A: Accnt)
The type of Issue_Check events is
    event(D: Dollar; A: Accnt; N: Check_No)
Assume that Dollar, Accnt, and Check_No are Integer types.

Example:

@(235, Savings, Name **is** "Deposit")
Deposit(235,Savings)

Commentary:

The basic patterns above are equivalent; the second simply uses the more intuitive syntax.

Consider the computation of Figure 2.2, four events ordered in a sequence.

The second of the two Deposit events of the figure would constitute a match for the basic patterns, since only it has the action name given in the pattern, and its parameters are equal to those in the pattern.

Recall that the case of the Name string parameter is irrelevant: an equivalent basic pattern is @(235, Savings, Name **is** "DEPOSIT").

□

Example:

Deposit

Commentary:

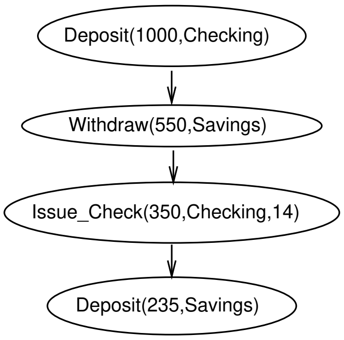

Figure 2.2: A simple computation (Arrows indicate causal dependency)

Each of the Deposit events of Figure 2.2 constitutes a match for this basic pattern. Because no parameters are given in the pattern, any parameter values are allowed.

The Issue_Check and Withdraw events would not match because their action names are different from the action name of the pattern.

□

Example: *A pattern using placeholders*

(?D : Dollar) Deposit(D **is** ?D, A **is** Savings)

Commentary:

This pattern declares and uses the placeholder ?D of type Dollar. It will be matched by a Deposit event with an A parameter equal to Savings, and any value for its D parameter. That value is the substitution value for ?D.

In the computation of Figure 2.2 only the second Deposit event would match the pattern.

□

Example: *An event part*

(?E : **event**(D: Dollar; A: Accnt)) ?E@Deposit

Commentary:

Each of the Deposit events of the figure would constitute a match for the pattern.

The substitution for ?E is a value that uniquely identifies the event matched. We can use the selection notation of record constituents to represent the parameter values of an event referred to by a placeholder; for example ?E.D would give us the D parameter value of the event represented by ?E. If the basic pattern were part of a larger pattern that contained other occurrences of ?E, all such occurrences would refer to the same event.

Section 2.4 on composite patterns show uses of identifying events with placeholders.

□

Example: *Using subtyping in matching*

> **type** Privileged_Accnt **is interface**
>   **include** Accnt;
>   Priv_Code : Integer;
> **end interface**;
>
> (?PA : Privileged_Accnt) Deposit(A **is** ?PA)

Commentary:

Since ?PA is of type Privileged_Accnt, which is a subtype of Accnt, only a Deposit event that has a value of this subtype as its A parameter will match this pattern.

By subtyping of placeholders as well as by explicitly stating required parameter values, one can in most cases make sure a basic pattern is only matched by an event of interest, ignoring all others (another way to do this is by using *guards* — see Section 2.4.11). Subtype indications and parameter value requirements essentially function as simple *filters* in observing the computation.

□

Example: *A placeholder action part*

> (?E : **event**(D: Dollar; A: Accnt)) ?E@
> (?E : **event**(D: Dollar; A: Accnt) ?E@(D **is** 550)

Commentary:

The first pattern would match an event of that type (or a subtype of it) regardless of which action name the event is associated with. Thus each event of Figure 2.2 would be a match. Note that the type of the Issue_Check event is a *subtype* of **event**(D: Dollar; N: Check_No), since it has the required parameters (and others).

The second pattern would be matched only by those events that are of the type **event**(D: Dollar; A: Accnt) with a D parameter value of 550; ?E denotes the event.

In the computation of Figure 2.2 only the Withdraw event would match the pattern.

□

Example: *Performer of an action*

> Deposit(Performer **is** Cust1)
> (?C : Customer_Type) Deposit(Performer **is** ?C)

Commentary:

The first pattern will be matched by every Deposit event that has been performed by the Cust1 object; the second will be matched by every Deposit event that has been performed by an object of type Customer_Type.

□

2.4 Composite Patterns

One is often interested in recognizing complex patterns of behavior. Composite patterns are pattern expressions built from smaller patterns; while basic patterns are always matched by exactly one event, composite patterns may potentially be matched by any number of events.

The binary operation '$\longrightarrow$' on patterns is called the *sequence* combinator, '$\triangleright$' is called the *immediate sequence* combinator, '$\sim$' is called the *join* combinator, '$||$' is called the *independence* combinator, '**or**' is called the *disjunction* combinator, '**and**' is called the *conjunction* combinator, and '$\cup$' is called the *union* combinator.

An informal semantics for each combinator is given in the following sections.

2.4.1 Sequence

The sequence operator allows matching of two sets of events that are ordered in the dependency computation.

Semantics

A subcomputation C matches the sequence pattern

$$P_1 \longrightarrow P_2$$

if C can be partitioned into two subcomputations C_1 and C_2 such that (1) all events of C_1 precede all the events of C_2 in the dependency computation, and (2) C_1 matches P_1, and C_2 matches P_2.

Example: *A sequence pattern*

Deposit(A **is** Checking) $\longrightarrow$ Balance(A **is** Checking)

Commentary:

> Assume Checking is declared as a value of type Accnt.
>
> In the banking example, there may be many possible dependencies between deposits, withdrawals, and balance reporting. The balance of an account is dependent upon all the deposits and withdrawals that result in the particular balance value. Similarly, a withdrawal may be dependent upon a balance statement, in the sense that a person's decision to withdraw some amount may be influenced by whatever the balance statement says about the state of his account.
>
> A subcomputation will match the above pattern if it consists of two events (one for each of the basic descriptors of the pattern), one matching the first basic pattern, the other the second, *and* they are ordered in the computation, with the Deposit event preceding the Balance event in the computation. Additionally, both must have the A parameter value of Checking.
>
> In Figure 2.3 the two shaded events constitute a match for the pattern. However, the shaded Balance and the Deposit of \$950 do not, since they are unordered. Neither does the Deposit for \$100 together with the Balance constitute a match, since their ordering is the opposite of the one the pattern requires.

□

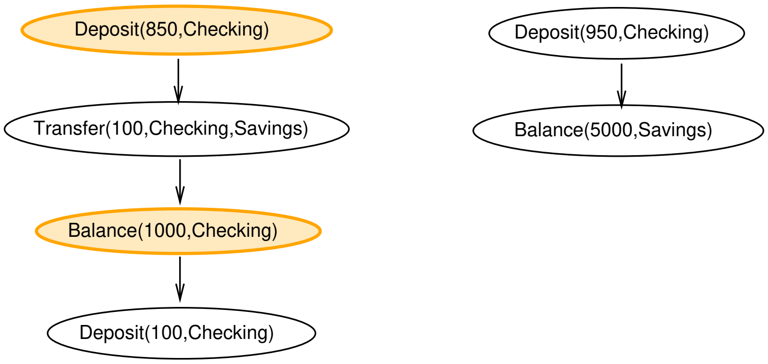

Figure 2.3: A match for a sequence pattern

2.4.2 Immediate Sequence

In addition to the sequence ordering of event patterns, it is often useful to require the immediate sequence ordering: two sets of events, one of which follows the other with no intervening events.

Semantics

A subcomputation C matches the sequence pattern

$$P_1 \;\triangleright\; P_2$$

if it can be partitioned into two subcomputations C_1 and C_2 such that (1) all events of C_1 precede all the events of C_2 in the dependency computation, (2) in the set of events S in which a match is sought (i.e. $C \subseteq S$), there exists no event that both follows any maximum of C_1 and precedes any minimum of C_2 in the dependency computation, and (3) C_1 matches P_1, and C_2 matches P_2.

This combinator refers to the entire set of events in which the match is sought; it is the only pattern combinator that does so. For patterns occurring in reactive statements of processes, see Section 7.3 of the Executable LRM; for patterns occurring in pattern constraints, see Section **??** of the RAPIDE 1.0 Constraint Language Reference Manual; and for poset generators, see Section **??** of the Executable LRM.

Example: *An immediate sequence pattern*

Deposit $\triangleright$ Balance

Commentary:

> This pattern will be matched by two events, a Deposit followed in the ordering by a Balance, where no other visible events are between these two events in the ordering. In Figure 2.3, only the rightmost Deposit/Balance pair matches the pattern.

□

2.4.3 Join

The join operator combines two disjoint subcomputations that match the sub-patterns. In these LRMs we equivalently say that two disjoint computations are *joined* to form a composite computation, and that a computation is *partitioned* into two smaller computations.

Semantics

A subcomputation C matches the join pattern

$$P_1 \sim P_2$$

if C can be partitioned into two subcomputations C_1 and C_2 such that (1) C_1 matches P_1, and (2) C_2 matches P_2. In other words, disjoint C_1 and C_2 may be joined to form a match for the pattern.

Example:

Receive(?D1, His_Employer) ~ Receive(?D2, Her_Employer)

Commentary:

> All matches for this pattern will consist of two Receive events, one with the employer parameter being equal to the value of His_Employer, the other with the employer parameter being equal to the value of Her_Employer. Whether the two events are ordered does not matter.
>
> If His_Employer=Her_Employer then a subcomputation consisting of only *one* Receive does *not* constitute a match for the pattern, since there is no partitioning of the singleton set which would result in two distinct matches for the two operands.

□

2.4.4 Independence

One often wants to determine not only the ordering of events, but also *whether* there is an ordering. Independence is a combinator requiring its constituent matches be unordered in the computation.

Semantics

A subcomputation C matches an independence pattern

$$P_1 \; || \; P_2$$

if C can be partitioned into two subcomputations C_1 and C_2 such that (1) no event in C_1 is dependent upon any event in C_2 and vice versa, and (2) C_1 matches P_1 and C_2 matches P_2.

Note that since the definition requires a disjoint partitioning of the subcomputation, independence is not reflexive; that is, an event is not independent with itself.

Example: *Two independent events*

(?A : Accnt)
Deposit(A **is** ?A) || Deposit(A **is** ?A)

Commentary:

This pattern would be matched by two Deposit events that are unordered and that have the same A parameter value. The two shaded events of Figure 2.4 have this relationship.

□

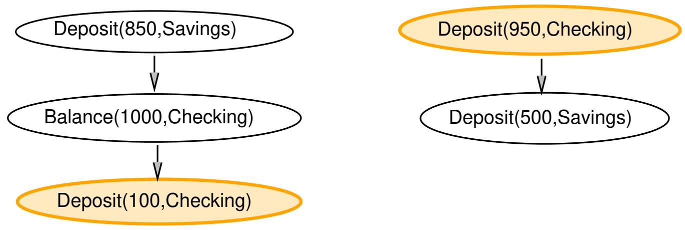

Figure 2.4: A match of two independent events

2.4.5 Disjunction

Instead of looking for a set of (sub-)patterns which all have to be matched for a match of the whole pattern to occur, we may equally well wish to look for a match of any *one* among a set of patterns.

Semantics

A subcomputation C matches a disjunction pattern

$$P_1 \ \textbf{or} \ P_2$$

if C matches P_1 or C matches P_2.

Example:

Withdraw **or** Balance

Commentary:

This pattern would be matched by a single event if that event is *either* a Withdraw event *or* a Balance event.

□

Example: *A disjunction of composite patterns*

$$\begin{array}{ll} & (\text{Balance}(\text{A} \ \textbf{is} \ \text{Checking}) \ \| \ \text{Deposit}(\text{A} \ \textbf{is} \ \text{Checking})) \\ \textbf{or} & (\text{Balance}(\text{A} \ \textbf{is} \ \text{Checking}) \longrightarrow \text{Deposit}(\text{A} \ \textbf{is} \ \text{Checking})) \end{array}$$

Commentary:

This pattern would match pairs of Balance and Deposit events with A parameters equal to Checking where either: 1) the Balance event precedes the Deposit event in the computation, or 2) the two events are independent.

Figure 2.5 shows a computation with two matches for this pattern. In the first match, the Balance event that precedes the Deposit and the A constituent of which is equal to Checking, together with the Deposit form one match. The Deposit event and the Balance event that is independent with it form the other match.

If this pattern occurred in, for example, a process, either match could trigger continuation of the process. See Chapter 7 of the Executable LRM for a description of how a process chooses among more than one possible match.

□

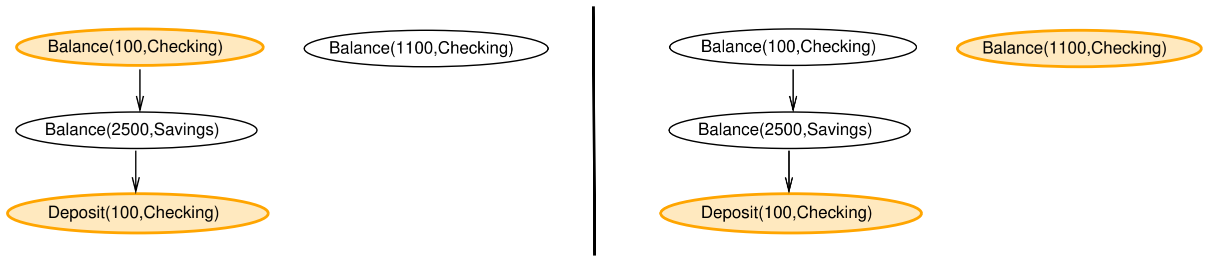

Figure 2.5: Matches of a disjunction pattern

2.4.6 Conjunction

We might want to describe a subcomputation that is a match for two different patterns. To do this, we use conjunction.

Semantics

A subcomputation C matches the conjunction pattern

$$P_1 \ \ \textbf{and} \ \ P_2$$

if C matches P_1 *and* C matches P_2.

Example:

((Deposit $\longrightarrow$ Withdraw) $\sim$ Issue_Check) **and**
((Withdraw || Issue_Check) $\sim$ Deposit)

Commentary:

Here we state two requirements for a match of this pattern: a Deposit event must precede a Withdraw event, and a Withdraw event must be independent with an Issue_Check event. The pattern does not require or forbid an ordering between the Deposit and the Issue_Check. Figure 2.6 shows a computation that matches the pattern; again, if there were an ordering between the Deposit and the Issue_Check, the three events would still form a match.

□

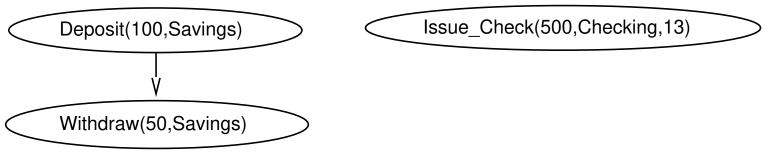

Figure 2.6: A match for event placeholders and conjunction

2.4.7 Union

We are sometimes interested in the union of two possibly overlapping sets of events, but not in the dependencies (if any) among them.

Semantics

A subcomputation C matches the union pattern

$$P_1 \cup P_2$$

if there are two (possibly overlapping) subcomputations C_1 and C_2 such that C_1 matches P_1 and C_2 matches P_2, and $C = C_1 \cup C_2$.

Example:

Receive(?D1, His_Employer) $\cup$ Receive(?D2, Her_Employer)

Commentary:

> Assume His_Employer and Her_Employer are values identifying two persons' respective employers.
>
> This pattern will be matched by a subcomputation consisting of either one or two Receive events.
>
> Two Receive events, one with the employer parameter being equal to the value of His_Employer, the other with the employer parameter being equal to the value of Her_Employer, will be a match for the pattern. Whether or not the two events are ordered is irrelevant.
>
> Additionally, if His_Employer=Her_Employer then a subcomputation consisting of only *one* Receive would constitute a match for the pattern as well, since there is no requirement that the matches for the two subcomputations be disjoint; ?D1 and ?D2 would in that case have the same substitution value.
>
> When His_Employer=Her_Employer any two-event match would contain two other, one-event matches. For example, the subcomputation { Receive(100,Boss), Receive(200,Boss) } where His_Employer=Her_Employer=Boss contains a total of three matches of the above pattern: two of size one, and one of size two.

□

Example: *Union with event placeholders*

(?E@Deposit $\longrightarrow$ Withdraw) $\cup$ (?E@Deposit $||$ Issue_Check)

Commentary:

This pattern will be matched by a Deposit event that is both followed by a Withdraw event and independent with an Issue_Check event. The relationship between the Withdraw event and the Issue_Check event is irrelevant: the two may or may not have an ordering. Figure 2.6 shows a computation that matches this pattern. The computation would also form a match if the Issue_Check preceded the Withdraw.

□

2.4.8 Iteration

One would often like to apply the binary pattern operations not only to two operands, but to any number of them. For example, we might want to match subcomputations with twelve Deposit events, or with an arbitrary number of ordered events.

Syntax

The binary pattern combinators may be iterated over via an iterator expression (See the Predefined Types LRM, Chapter 12), in which case from a pattern P, a pattern combinator O and an iterator that denotes n objects, we get a new pattern equivalent to n occurrences of P, related by the combinator O.

Instead of an iterator, one can use "$*$" and "$+$", which play roles similar to their namesakes in regular expression languages.

The syntax for iteration patterns is

pattern	::=	*iterator_decl pattern*
iterator_decl	::=	'[' [*ident* :] *iterator_exp* **rel** *binary_pattern_operator* ']'
iterator_exp	::=	*iterator* | '$*$' | '$+$'
iterator	::=	*object_expression* | *range*

Semantics

The *binary_pattern_operator* is either one of the predefined pattern combinators, or a pattern macro definition (see Section 2.5) with exactly two parameters, both of which are pattern parameters. The *iterator* is either an object expression that, for some type T, is a subtype of Iterator(T); or a range (see Predefined Types LRM, Section 12).

For a discussion of when object expressions in patterns are evaluated, see Section 3.4.

A subcomputation C matches the iteration pattern
[id : iter **rel** op] (P)
for a pattern operator *op*, an iterator expression *iter* that is a subtype of Iterator(T) for some type T, and a pattern P, in the following case.

The iterator expression implicitly defines a list of objects of type T of length n: $[L_1, L_2, \ldots, L_n]$

A list of patterns $P_1, P_2, \ldots, P_n$ is also implicitly defined where:

- $P_i \equiv P \mid^{id}_{L_i}$; that is, in P_i occurrences of id in the original pattern are replaced with an expression denoting L_i.

C matches the iteration pattern if and only if C matches

$$P_1 \ op \ P_2 \ op \ \ldots \ op \ P_n$$

If $n = 0$ (the iterator defines a list of length zero), the pattern is matched only by the empty computation. If $n = 1$, the pattern is matched by exactly those computations that match P.

A subcomputation C matches the pattern
[id : ∗ **rel** op] (P)
if C matches
(?n : Integer) [id : 1..?n **rel** op] ((P **where** ?n >= 0))

A subcomputation C matches the pattern
[id : + **rel** op] (P)
if C matches
(?n : Integer) [id : 1..?n **rel** op] ((P **where** ?n > 0))

Iteration such as "[∗ **rel** ⟶] (P)" and "[+ **rel** ⟶] (P)" is reminiscent of the "∗" and "+" operators on regular expressions.

The pattern
[iter **rel** op] (P)
(i.e. an iterator pattern with no identifier) is matched by the computations that match
[unique : iter **rel** op] (P)
where unique is an unique identifier that does not occur in P. In this case, $P_1 \equiv P_2 \equiv \ldots \equiv P_n \equiv P$.

Example:

[1..12 **rel** ∼] (Deposit(A **is** Checking))

Commentary:

This pattern would be matched by 12 Deposits to the Checking account.

□

Example:

[∗ **rel** ⟶] (Balance)

Commentary:

This pattern describes zero or more ordered Balance events. A computation consisting of three ordered Balance events would actually contain eight matches for this pattern: one of size zero (the empty computation), three of size one, three of size two, and one of size three.

□

Example: *Placeholders in iterator patterns*

([1..?I **rel** ||] (Deposit)) ⟶ Num_Deposits(?I)

Commentary:

Note that the iterator expression may contain placeholders.

This pattern would be matched by a Num_Deposits event, preceded by a number of independent Deposit events equal to the parameter of the Num_Deposits. There may be many distinct matches for this pattern in a given computation. For example, in a computation including the event Num_Deposits(3) which is preceded by five independent Deposit events, each of the $\binom{5}{3} = 10$ subcomputations of three Deposit events, along with the Num_Deposits event, will be a match for this pattern.

□

Example: *Named iterator*

$$[\ \mathrm{I} : 1..10\ \mathbf{rel} \longrightarrow]\ (\mathrm{Issue\_Check}(\mathrm{N}\ \ \mathbf{is}\ \ \mathrm{I}))$$

Commentary:

The range 1..10 is an integer iterator, so the identifier I is of type Integer. Thus the pattern is matched by successive ordered matches of the basic pattern that follows; for each such match, the values of I are the successive values in the range.

That is, the above pattern is equivalent to the pattern

$$\mathrm{Issue\_Check}(\mathrm{N}\ \ \mathbf{is}\ \ 1) \longrightarrow \mathrm{Issue\_Check}(\mathrm{N}\ \ \mathbf{is}\ \ 2) \longrightarrow \ldots$$
$$\longrightarrow \mathrm{Issue\_Check}(\mathrm{N}\ \ \mathbf{is}\ \ 10)$$

□

Example: *Named iterator over a complex type*

$$[\ \mathrm{A} : \mathrm{Airplane\_Iter}\ \mathbf{rel}\ ||]\ ((\mathrm{A.Transmit\_Position} \longrightarrow \mathrm{Tower.Ack\_Position(A)}))$$

Commentary:

Here the expression Airplane_Iter is assumed to be an iterator over type Airplane. So identifier A is of the type Airplane, a type defining complex objects that that generate their own events. The pattern will be matched by a computation that consists of parallel matches of the pattern that follows the placeholder definition, where A has the value of an Airplane object for each match, and there is exactly one match for each value in the domain of Airplane.

Because the || operator is symmetric, the order defined by the iterator is irrelevant.

□

2.4.9 Placeholder Patterns

Placeholders may be declared in a pattern by giving a list of placeholder declarations, followed by a pattern. The scope of the placeholders is the pattern that follows the declarations.

Syntax

The syntax of a placeholder pattern is

pattern ::= *pholder_decl_list pattern*

pholder_decl_list ::= *'(' pholder_decl* { *';' pholder_decl* } *')'*

pholder_decl ::= *identifier_list* : *type_expression*

Semantics

The placeholder declaration defines a list of placeholders and their type. An occurrence of a placeholder in the pattern following its declaration is an object expression of the placeholder's type.

A subcomputation C matches a pattern

$$(I : T)P(I)$$

if there exists an object O of type T such that $\mathsf{P(O)}$ is matched by C.

For a particular match of a pattern, a placeholder scope defines a textual area of the pattern for which all occurrences of a placeholder must be matched by the same value. This value is called the *substitution value* of the match.

Often there will exist only one such substitution; for example, for a placeholder being matched to the argument of an event, the value of the argument will be the only possible substitution in that scope. If two occurrences of a placeholder are being matched to two different event arguments with different values, no substitution is possible, and so the subcomputation does not form a match. It is also possible for there to be more than one substitution (see the last example in Section 2.4.11).

There is no requirement that two distinct placeholders have distinct substitution values in a match.

Example: *Pattern-wide placeholder*

(?A : Accnt) [1..12 **rel** ~] ((Deposit(A **is** ?A) $\longrightarrow$ Deposit(A **is** ?A)))

Commentary:

The pattern will be matched by a computation C if it consists of twelve disjoint ordered pairs of deposit events, and all twenty-four deposit events have identical A parameter values.

□

Example: *Local placeholder definition*

[1..12 **rel** ~] (((?A : Accnt) Deposit(A **is** ?A) $\longrightarrow$ Deposit(A **is** ?A)))

Commentary:

The pattern will be matched by a computation C if it consists of twelve disjoint ordered pairs of deposit events, and the A parameters are equal *within each pair*. There are no restrictions on the values of A in different pairs.

□

Example: *Nested placeholder definitions with hiding*

$$\begin{array}{l}(\ ?A \ : \ \mathrm{Accnt} \ ) \\ \quad \mathrm{Balance}(A \ \mathbf{is} \ ?A) \longrightarrow \\ \quad ((\ ?A \ : \ \mathrm{Accnt} \ ) \ \mathrm{Deposit}(A \ \mathbf{is} \ ?A) \longrightarrow [\ 1..12 \ \mathbf{rel} \sim] \ (\mathrm{Deposit}(A \ \mathbf{is} \ ?A))) \longrightarrow \\ \quad \mathrm{Balance}(A \ \mathbf{is} \ ?A)\end{array}$$

Commentary:

The pattern will be matched by a computation C if it consists of (1) twelve disjoint ordered pairs of deposit events, and all twenty-four deposit events have the same A parameter values, (2) two Balance events respectively preceding and succeeding all the Deposit events, and (3) the A parameters of the two Balance events are equal.

There is no requirement that the A parameters of of the Balance events be equal to the A parameters of the Deposit events.

The ?A of the inner pattern could be replaced with a differently named placeholder of the same type without the meaning of the pattern changing. Indeed, to avoid confusion, nested definitions using the same placeholder identifier are not recommended.

□

2.4.10 Parenthesized Patterns

Syntax

The syntax of a parenthesized pattern is

'(' *pattern* ')'

Semantics

A subcomputation C matches a pattern

(P)

if it matches P.

Parenthesized patterns are used to associate sub-patterns.

2.4.11 Guarded Patterns

Context may be determined by various components of a pattern. For example, multiple occurrences of a placeholder must all have the same substitution value (see Section 2.4.9). Context may also be determined by the use of *guarded patterns*.

Syntax

pattern ::= *pattern* **where** *guard*
guard ::= *algebraic_constraint*

A guarded pattern contains a sub-pattern and a boolean expression, called the *guard*.

Semantics

A subcomputation C matches a *guarded pattern*

P **where** B

if C matches P and B has the value True.

For a discussion of how object expressions that occur in guarded patterns are evaluated, see Section 3.2.

Example: *A simple audit attention pattern*

(?D : Dollar)
Deposit(D **is** ?D) **where** (?D > 10_000)

Commentary:

An auditor process could be interested in deposits of checks whose amount exceeds \$10_000. This pattern would be matched only by such events.

It is often useful (though not strictly correct) to consider ?D as "getting" its value from the Deposit event, and that value then being used in evaluating the guard.

□

Example: *Guarded composite pattern*

(?N1, ?N2 : Check_No)
(Issue_Check(N **is** ?N1) ⟶ Issue_Check(N **is** ?N2)) **where** ?N1 > ?N2

Commentary:

This pattern would be matched by two Issue_Check events where the checks are issued in reverse order. This could be of interest to a process auditing the banking transactions.

□

Example: *Using "=" in a guard*

(?D, ?Available : Dollar)
Balance(?D, Checking) **where** (?Available = ?D−Minimal_Balance) ⟶ ...

Commentary:

Because ?Available appears as an argument to the "=" operation, the value of the other argument is the only possible substitution value for ?Available. In the elided part of the pattern, the guard insures that other occurrences of ?Available are matched only to this value.

□

Example: *Bouncing checks—guards and state*

(?D : Dollar; ?A : Accnt)
Withdraw(D **is** ?D, A **is** ?A) **where** \$(Balances(?A)) < ?D

Commentary:

Suppose there is an array Balances, where the array component Balances(n) reflects the balance of account n. The auditor could be looking for attempts to withdraw amounts exceeding the account balance — checks that would bounce — with the above pattern.

□

Example: *Multiple placeholder substitutions*

(?D : Dollar)
(Deposit(D **is** ?I) **or** Withdraw) **where** ?I > 0

Commentary:

This pattern is matched either by a Deposit event with Dollar argument greater than zero, or by any Withdraw event. In the case where a Withdraw event is the match, the placeholder ?D has multiple substitution values: any integer greater than zero would be a valid substitution. Thus the guard serves no useful purpose with respect to Withdraw, and may be moved to guard only the Deposit basic pattern.

□

2.5 Pattern Macros

One can use *pattern macros* to abstract (and parameterize) a pattern, and to define new operations on patterns.

2.5.1 Syntax

The syntax of these abstractions is

pattern_macro ::= **pattern** *identifier* '(' [*macro_parameter_list*] ')' **is** *pattern* ';'

macro_parameter_list ::= *macro_parameter* { ';' *macro_parameter* }

macro_parameter	::=	*value_parameter* | *type_parameter* | **pattern** *identifier_list*
pattern_macro_instance	::=	*expression*

2.5.2 Semantics

A *pattern macro instance* is an application of parameters to a pattern macro. A subcomputation C matches a pattern macro instance if it matches the pattern obtained by taking the body of the macro, and replacing occurrences of the formal parameters by the actuals given in the instance.

Parameter conformance in a macro instance for value and type parameters follows the rules defined in Chapter 6 of the RAPIDE 1.0 Executable Language Reference Manual. The actual parameter associated with a parameter of the form **pattern** P must be a pattern expression. For some pattern *patt* passed as a parameter in a pattern macro instance, "(*patt*)" (i.e. the pattern parenthesized) is considered to be the substituted expression of the above definition; this avoids associativity problems.

Pattern macro declarations follow the rules for defining infix and prefix syntax, given in Chapter 4 of the RAPIDE 1.0 Executable Language Reference Manual. A pattern macro name may be used as a binary infix pattern operator if it has exactly two parameters. A unary prefix pattern operator may be defined if it is one of the predefined unary operators, and the macro has exactly one parameter.

Pattern macros serve several purposes. They may act as reusable patterns. They may also simplify patterns by breaking them up into smaller units. Recursive macros allow the definition of patterns that are otherwise inexpressible. Finally, new pattern operators can be defined using macros.

Recursive macros and matchability. Note that recursive pattern macro definitions are possible. Some such macros will be inherently unmatchable, e.g.:

pattern P() **is** P(); -- *Macro calls itself*

Others may be matchable, depending on the strength of a particular toolset:

pattern P() **is**
P() **or** A **where** Fermats_Last_Thm_Is_True();

A toolset is not required to support all patterns that may be matchable. Further, matchability may not necessarily be detected by a compiler or run-time system.

Pattern macro profile. More than one pattern macro with the same name may be defined in a declarative region as long as the macros do not have the same *profile*. The profile of a pattern macro consists of: the number of parameters the macro has; whether the parameter is a type-, object-, or pattern-parameter; and typing information if it is a type- or object-parameter.

Two pattern macros have the same profile if they have the same number of parameters, and if, positionally: where one macro has a type parameter, the other also has a type parameter (further, if either has a type constraint, the other has a type constraint of an equivalent type); where one has a pattern parameter, the other also has a pattern parameter; and where one has an object parameter, the other has an object parameter of an equivalent type.

2.5.3 Examples

Example: *Defining a pattern operator*

> **pattern** Optional(**pattern** P) **is** P **or** Empty();

Commentary:

The macro Optional defines a new unary pattern operator. The semantics of Optional are that, given some pattern P, Optional(P) is equivalent to the pattern (P) **or** Empty().

□

Example: *A recursive pattern macro definition*

> **pattern** Ticks() **is** Tick $\longrightarrow$ Ticks();

Commentary:

Assume Tick is an action name. The pattern "Ticks()" is an instance of the above macro. It describes an infinite chain of ordered Tick events. Pattern macros permit the description of infinite computations, which are otherwise inexpressible (iteration can only describe finite, though arbitrarily large, computations). An infinite pattern will, of course, have limited use in RAPIDE processes, but can be useful in pattern specifications.

□

Example: *A bounded recursive macro*

> **pattern** Bounded_Ticks(I : Integer) **is**
> (Tick(I) **where** (I <= 20) $\longrightarrow$ Bounded_Ticks(I+1))
> **or**
> Empty() **where** (I > 20);

Commentary:

Assume Empty is a pattern macro, instances of which are matched by zero events.

An instance of this macro is a pattern matched by a subcomputation consisting of a sequence of Tick events with incrementally increasing parameter values, which terminates whenever a Tick with parameter 20 is encountered. For example, Figure 2.7 shows a computation that matches the pattern Bounded_Ticks(18). Instances of Bounded_Ticks with a parameter greater than 20 will be matched only by the empty computation.

□

2.6 Timing Operator

The Timing operator is a predefined macro that allow patterns to refer to the time at which events matched by the patterns are generated.

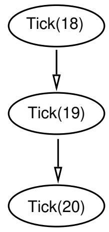

Figure 2.7: A match for a macro instance

Several other related pattern time operators are defined in terms of Timing; they are defined in the Predefined Types LRM, Chapter 18.

pattern Timing(**pattern** P; Time, Duration : Integer; L : Clock) **is**
implementation_defined

The pattern Timing(P,T,D,L), where T and D are of type Integer (a subtype of all Ticks types) and L is of type Clock, is matched by a computation C such that (1) C matches P, and (2) all events in C are related to L, and (3) T is the minimum (earliest) C.Start value of any event in C, and D is the duration of the pattern, calculated by taking the maximum (latest) C.Finish value of any event in C, minus T.

2.6.1 Examples

Example:

(?First, ?Dur : Integer)
Timing((Start_Proc1 || Start_Proc2) ⟶ End_Proc, ?First, ?Dur, C)

Commentary:

This pattern will be matched by computations consisting of three events, a Start_Proc1, Start_Proc2, and End_Proc event. Where S1 is the event matched to Start_Proc1 and S2 is the event matched to Start_Proc2, ?First will have the substitution value of C.Start(S1) or C.Start(S2), whichever is smaller. Where E is the event matched to End_Proc, ?Dur will have the substitution value of C.Finish(E) − ?First.

□

Chapter 3

Program State and Patterns

The concept of "state" in a concurrent program is more complex than it is for sequential programming languages. Because processes execute independently, the concept of a single sequence of program states does not apply. Similarly, several different processes may be independently accessing program data (in particular, dereferences of reference objects).

The result of a RAPIDE simulation is a partially ordered set of events; hence the state of a RAPIDE program is defined in terms of the events produced. State is defined with respect to a single event, or a set of events.

This definition of state enables us to define the meaning of reference object dereferences that occur in patterns.

3.1 State

A *consistent cut* of a computation C is a subset C_1 such that no event in $C - C_1$ precedes any event in C_1. (*i.e.*, one may think of the cut as a line across the computation, with all the causal arrows crossing the line doing so in the same direction. C_1 is then the set of events before the line. A different phrasing of the same definition: A consistent cut is a prefix-closed subset of C.)

A consistent cut *at* an event e is any consistent cut where e is a maximum of the cut. For a given event e, there may be more than one such consistent cut.

Observe that a consistent cut of the computation C identifies a set of modules—namely those modules that were generated before the cut (*i.e.*, those modules whose start event is in the cut). Furthermore, a consistent cut identifies the value associated with a reference object, namely the parameter of the last assignment to the reference object.

A *state* is then a set of objects and the dereference values (of the reference objects) determined by a consistent cut.

A *state at an event* e is a state determined by a consistent cut at e. Note that there may be more than one such state associated with an event.

A *state at a set of events* P is a state determined by a consistent cut at a maximum of P such that P is a subset of the cut. Note that there may be more than one such state associated with a set of events.

3.2 State in Guards

Guarded patterns are defined in Section 2.4.11.

Recall that a subcomputation C matches

P **where** B

if C matches P and B has the value True. B is evaluated in some state at the set of events that comprises the match.

The evaluation of B does not by itself have any effect on the computation. In the evaluation, attempts to perform assignments to reference objects not created by the evaluation itself raises the exception Illdefined_Guard.

3.2.1 Commentary

There are two interesting properties of guards using state.

Consistent Semantics in Pattern Matching and Constraint Checking: If a region contains a trigger
P **where** B
and a constraint
never P **where** B;
then if the trigger ever fires, a constraint violation must also occur.

Repeatable Reads: Dereferencing the same reference object multiple times in a guard will always return the same value. This is called repeatable reading; dereferencing (reading) a variable repeatedly in a single guard returns the same value.

Note that this does not hold in a statement, for example: two dereferences in a single statement may return different values, if other processes are independently assigning to that object.

3.3 Examples

Example: *State expression in a process*

```
when (?D in Dollar, ?A in Accnt)
        Withdraw(?D, ?A) where ?D > Balance(?A) do
    -- Take overdraft action
end when;
```

Commentary:

> Assume Balance is a function returning the balance of various accounts. Here the value of Balance(?A) is not determined until the pattern is being matched. If a Withdraw event is generated during the execution of this process body, Balance(?A) for the new event is not evaluated until the body execution is complete, and the pattern is again matched.

□

Example: *State expression in a process*

> **never** (?D **in** Dollar, ?A **in** Accnt)
> Withdraw(?D, ?A) **where** ?D > Balance(?A);

Commentary:

In this constraint, Balance(?A) is evaluated as soon as a Withdraw event is generated and matches the guarded pattern.

□

3.4 References to State Outside of Guards

A pattern containing dereferences of Ref objects (where those dereferences do not appear in a guard expression) is equivalent to a pattern in which all Ref object derefereces appear in a guard expression.

3.4.1 Examples

Example: *State reference in a basic pattern parameter*

> E($expr)
> (?P **in** T) E(?P) **where** ?P = $expr

Commentary:

Where "expr" is some Ref object expression of type Ref(T), the above patterns are equivalent.

□

Example: *State reference in an iterator expression*

> [1. .$expr **rel** ⟶] (A)
> (?P **in** T) [1. .?P **rel** ⟶] (A) **where** ?P = $expr

Commentary:

The above patterns are equivalent.

□

Glossary

basic pattern A template for matching a single event. See page 7.

computation A set of events and the orderings between them. See page 5.

conjunction A pattern operator matching both of two sub-patterns. See page 16.

consistent cut Informally, a line across the computation, with all the causal arrows crossing the line doing so in the same direction. See page 29.

disjunction A pattern operator matching either of two sub-patterns. See page 15.

guarded pattern A pattern containing a sub-pattern and a boolean guard; it matches if its associated pattern matches and if the guard evaluates to True. See page 23.

guard The boolean expression associated with a guarded pattern. See page 23.

immediate sequence A pattern operator that matches two subpatterns that are ordered in the dependency computation, with no events between them. See page 13.

independence A pattern operator that matches two subpatterns that are unordered in the dependency computation. See page 14.

iteration A pattern operator matching some number of sub-patterns, as determined by its iterator expression and iterator operator. See page 19.

iterator expression The iterator associated with an iteration pattern. See page 19.

iterator operator The binary pattern operator associated with an iteration pattern. See page 19.

join A pattern operator that matches two disjoint sub-patterns. See page 14.

match The fitting of a set of events and placeholder values to a pattern. See page 5.

partition The separation of a set of events into two disjoint subsets. See page 14.

pattern macro instance A pattern obtained by applying parameters to a pattern macro. See page 25.

pattern macro A parametrized pattern abstraction; applying parameters to a pattern macro yields a pattern called a pattern macro instance. See page 25.

placeholder pattern A pattern that defines one or more placeholders; all instances of a placeholder defined in a placeholder pattern must have the same substitution value in any match of the sub-pattern it contains. See page 21.

placeholder A free-variable-like construct for expressing typed "holes" in patterns. See page 6.

sequence A pattern operator that matches two subpatterns that are ordered in the dependency computation. See page 12.

state A set of objects and the dereference values (of the reference objects) determined by a consistent cut. See page 29.

substitution value For a particular placeholder defined in a placeholder pattern, the value which, when replaced for all instances of the placeholder, produces a match of the pattern. See page 21.

timing operator A pattern operator used to derive timing information from events. See page 27.

union A pattern operator matching two possibly overlapping sub-patterns. See page 17.

Bibliography

[Tea94] Rapide Design Team. *The Rapide-1 Executable Language Reference Manual.* Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.

Index