PART II

Building Solutions with CEP

Highlights in Part II:

- *The fundamental elements:*
  - *Event patterns and pattern languages*
  - *Reactive event pattern rules and constraints*
  - *Event causality and event hierarchies*
- *The building blocks of CEP applications:*
  - *Event processing agents*
  - *Architectures of agents*
- *Case studies of CEP applications*
- *Steps towards developing an infrastructure to support CEP applications*

Chapter 8

The RAPIDE *Pattern Language*

- *Designing event pattern languages*
- *The* RAPIDE *event pattern language* (RAPIDE-EPL)
- *Classes of events*
- *Pattern matching—the fundamental operation in CEP*
- *Single event patterns*
- *Complex event patterns*
- *Repetitive and recursive event patterns*
- *Event pattern macros*
- *Building libraries of patterns*

So far we have explored the concepts of CEP using a simple event pattern language, STRAW-EPL. We chose the tabular format because it lays out all the elements of an event pattern in a way similar to a graphical user interface. STRAW-EPL is fine for explaining easy examples. But when we deal with how to build real applications of CEP, we need a more powerful event pattern language.

In this chapter, we describe in some detail the features of a particular event pattern language that formed the basis for early research into CEP.

This is the RAPIDE-EPL for specifying patterns of events in causal event executions (posets). The RAPIDE-EPL contains a core of features that are needed to support CEP. We use it extensively in later chapters to illustrate example applications of CEP because it is a more powerful and succinct notation than the tabular format of STRAW-EPL. As the complexity of CEP applications develops over time, we expect that EPLs will include many more features.

8.1 Event Pattern Languages—Basic Requirements

First, we discuss some general requirements that an event pattern language should meet in order to adequately support CEP.

An event pattern language is a computer language in which we can precisely describe patterns of events. It is similar to mathematical language for logical expressions or a Web search language with more than the usual options and some other bells and whistles. It lets us describe, without any ambiguity, exactly the patterns in which we are interested.

A *pattern matcher* for an EPL is a program that processes one or more event executions in real time and picks out all, and only, posets that match a pattern.

An EPL must have the following properties.

- *Power of expression:* It must be powerful enough to specify the kinds of complex patterns that are needed in order to apply CEP to the problems described in Chapters 1 through 4. To do this, it must provide relational operators corresponding to relationships between events (for example, "and," "or," "causes," and "is independent of"), temporal operators (for example, "during," "at," "within") that refer to time bounds and time intervals, and ways to refer to data inside events and to context.
- *Notational simplicity:* It must let us write patterns easily and succinctly. The tabular format of our STRAW-EPL is too lengthy and restrictive for "heavy-duty" use. Some powerful pattern language constructs would be difficult to express in tabular format.
- *Precise semantics:* It must have a mathematically precise concept of *match* so that when we specify a pattern, we know the posets of events that can match it.
- *Scalable pattern matching:* It must have an efficient pattern matcher that can scale to matching large numbers of patterns over high volumes of events in real time. This issue will inevitably influence language design.

When it comes to designing EPLs, we have a "tension" between our requirements: *simplicity, ease of use,* and *efficiency of pattern matching* versus *power of expression.*

There is a fact-of-life reason for this. If an EPL is simple and easy to use, we won't be able to specify some kinds of complex patterns in it. On the other hand, if it is powerful and lets us specify complex patterns, it will contain "advanced" features or options that take time to learn how to use. And pattern matching for complex patterns is computationally demanding and difficult to implement efficiently.

8.2 Features of RAPIDE

The RAPIDE event pattern language is a declarative computer language for writing patterns of events. The patterns can specify sets of events together with their parameters, timestamps, and causal dependencies, and which events are causally independent of each other.

Declarative means that RAPIDE-EPL consists of mathematical expressions that "declare" (or describe) patterns. It does not include any algorithmic programming features like assignment or conditional branches. It is as simple a language as can meet the basic requirements for CEP.

Here is a summary of its main features.

- *Strong typing* to avoid common errors in writing patterns.
- *Basic data types* for specifying the data parameters of events and contexts.
- *Event types* for expressing the types of events in a pattern.
- *Basic event patterns* that allow us to express patterns that match single events—for example, any order from any customer.
- *Pattern operators* for expressing relationships between events. Pattern operators are used to specify complex patterns of many events in precise relationships.
- *Context* that lets us restrict matches of patterns to specific contexts in which events are observed.
- *Temporal operators* that allow us to specify the timing of events that match a pattern, or when a pattern should or should not match.
- *Pattern macros* that let us express complex patterns succinctly and build libraries of patterns.
- *Mathematical semantics.* RAPIDE-EPL has a simple one-page definition of matching that provides a specification for any pattern matcher.

The syntax is easy to understand and is similar to the syntax of object-oriented languages such as Java or C#, with a few small variations. In fact,

RAPIDE-EPL can be viewed as an "add-on" or extension of expressions in these languages.

In the following sections, we describe RAPIDE-EPL and the semantics of pattern matching. We want to tell you, the reader, as simply as possible how to write event patterns in RAPIDE-EPL. This is done with informal, intuitive descriptions of matching. We don't give a mathematical definition of matching here. There are precise formal definitions of matching in other documents about RAPIDE.

8.3 Types

RAPIDE-EPL is strongly typed, much like most modern object-oriented languages. It may seem odd to impose strong typing on patterns, but it turns out to help users avoid all kinds of silly errors in patterns, such as typos. Such errors show up as type mismatches in the pattern. Types also play a powerful role in restricting the context in which a pattern can match. This makes pattern matching more efficient.

There are three kinds of types: *data types, event types,* and *execution types.*

When we write a pattern, we must first declare the types of data in the pattern and the types of events we expect to match the pattern against. A set of type declarations is called a *type context.*

Example 1: *Defining Warning events*

If we want to write patterns that will match Warning events, either about the loads on routes in a network or about the distance between aircraft, we first define the types of these Warning events:

```
// type declarations preceding a pattern.
typedef  Network_Path ...;                  -- data type
typedef  Aircraft ...:                      -- data type
action Warning(Network_Path P1, Real P2); -- event type
action Warning(Aircraft P1, Feet P2);       -- event type
```

This type context declares two types of Warning events. It defines a *global type context* for patterns that follow it. Following these type declarations (that is, in the scope of the global type context), we can write a pattern such as this:

```
// pattern that matches network Warning events.
(Network_Path Route; Real Load) Warning(Route, Load);
```

This a basic event pattern that can match single events. It consists of declarations of the types of parameters in the pattern followed by an event template. The type declarations of the parameters are a *local type context* for the pattern that restricts the types of parameters only in this pattern.

As the example shows, a pattern consists of two parts: first a list in parentheses of the variables in the pattern together with their types, and then the pattern part—the part that describes the events to be matched. Events have names called *action names* followed by a list of parameters. The name of an event is in fact a parameter of the event that we give a special syntactic emphasis by putting it first. It is similar to the subject of a message. The name of an event must match the action name in the event pattern in order for a match to be possible. So, for example, a Warning(...) pattern cannot match a ConnectTime event.

Here, the pattern matches events with the action name, Warning, and parameters Route (which is of type Network_Path) and Load (which is a Real number).

Patterns are checked for type consistency before they are compiled for matching. This is where typos and other errors are caught.

Matching must be consistent with the types of the parameters in the pattern, and this restricts the events that can match the pattern. Our Warning pattern will match events like Warning(London–NewYork, 0.75), which is the kind of event we are looking for. London–NewYork is a Route and 0.75 is a Real number giving a measure of urgency.

If we omitted the typed parameter list from the pattern and just wrote the pattern part, we might get matches of Warning events from a different type of Warning event, like Warning(UA51, 5000), a warning of an aircraft within 5,000 feet. Although this may be an interesting match, it isn't a network warning, which is the kind of event our pattern is intended to match.

A pattern has both a *global type context,* where the types of events it can match are declared, and a *local type context,* where its variables and their types are declared. Because different types of events can have the same action name, the local context is important in disambiguating action names and specifying the types of events a pattern is intended to match. The data types in the local context must be subtypes of data types in the global context.

8.3.1 Predefined Types

RAPIDE-EPL has a set of predefined types. These are very common data types that appear in the events generated by many systems.

The predefined types include Boolean, Character, String, Integer, Float, and so on. Each predefined type comes with a set of operations. For example, Integers have the usual operations "+", "−", "=", and so on. Strings have a rich set of predefined operations—for example, comparison operations, such as S < T; selection operations, such as S [2 .. 4], which returns the substring consisting of the second, third, and fourth characters of S; and concatenation, &, which lets us construct a new string, S&T, from strings S and T.

8.3.2 Structured Types

Structured types are composed from other types. They let us define objects that have other objects as components.

Common kinds of structured types that are in many languages are predefined: **record** (record types), **array** (array types), **enum** (enumeration types), and some special types that we will describe later. Each of these structured types has predefined selection operations that let us select out the components of structured objects. To construct a structured object, we can assign objects as its components.

Structured types are defined by using *type definitions*. A type definition lets us define a type and give it a name. We can then use that name in declaring parameters of that type. The form of a type definition is

typedef *type−expression name*;

Example 1: *The record type definition*

An example of a record type definition is

```
typedef record {Node N1; Connection C;  Node N2} Network_Path;
```

Here we have defined Network_Path to be a record type with three components: two nodes and a connection. We can select components of a record object using the parameters of the type definition and the "." selection operation. So, if Route is an object of type Network_Path, "Route . N1" is the first Node of Route, and so on.

Example 2: *The array type definition*

An example of an array type definition is

```
typedef array [1, 2, 3] of String Triplet;
```

The *index* type in "[,]" must be an enumeration type, such as Integer. All the components of an array type must be of the same type—in this example, String.

Selection of components is done by applying a value of the index type to an object of the array type using "( )" notation. For example, if ThreeSome is of type Triplet, then ThreeSome(1) is its first string component.

8.3.3 Event Types

Events are objects that are tuples of data. An event contains the values of predefined attributes (such as its action name and its timestamps) as well as additional data parameters.

In RAPIDE-EPL there is a predefined *event type,* Event, which is the type of all events. This is the type of all events with any name, any parameter list (which can be empty), and the predefined attributes (which may have undefined values).

It is useful—for example, for efficient pattern matching—to be able to classify events into subtypes. Subtypes of events are declared by *action declarations.*

An *action declaration* specifies a subtype of events. An action declaration has the format

action *identifier* (*list of parameter declarations*);

where the *identifier* is the *action name,* and the list of parameters in paraentheses declares the tuple of data in the events. The parameter list is a list of declarations that consist of the type of a parameter followed by the name of the parameter. The predefined attributes are always implied members of the list of parameters and are not explicitly declared. An action declaration specifies the set of those events that have the action's name as their *action name attribute* and contain a tuple of data parameters that conform to the types of the action's parameter declarations.

Example 1: *A Warning event*

A type of Warning event is

```
action Warning(Network_Path Route; Real Load);
```

Examples of events in this action type are

```
Warning(London−NewYork, 0.75)
Warning(Paris−London, 0.92)
```

However, the following events are not members of this action type:

```
ConnectTime(London−NewYork, 02.56)   −because the action name is not "Warning"
Warning(UA51, 5000)   −because the data parameters are of the wrong types
```

Example 2: *An Order event*

A type of order event in a supply chain system is

```
action Order(Cust_Id Customer, Parts_Order Data, Accnt_no Accnt, ...);
```

Here we specify a subtype of events that contain the customer's Id, the order form in a required Parts_Order format, the account number, and other data. Order events will also contain the attributes they inherit from the RAPIDE-EPL event type, such as timestamps.

Figure 8.1 shows three subtypes of events that can be defined by *action declarations*. Each event subtype defines events from a particular system, or problem domain. The DTP events are the kind of events created by a

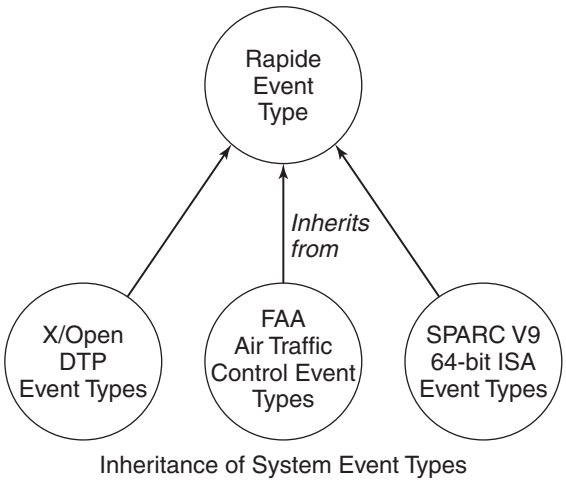

Figure 8.1: Subtypes of the RAPIDE event type

distributed transaction system, the FAA events are the type of events created in air traffic control, and SPARC V9 events are created by a simulation of a CPU architecture. In each subtype, the events inherit the attributes of the basic event type and contain additional data specific to a particular kind of system.

The subtype of XML events—that is, events having an XML format—is another example of an event subtype.

The idea behind actions is very simple. We think of an activity in the target system as leading to the creation of an event. We call such an activity an *action*. We give it a name—its *action name*—and a list of parameter declarations. The action declaration defines the subtype of the forms of events[1] that signify the system activity.

8.3.4 Execution Types

An execution is a poset of events. Its type is called an *execution type*. An execution type is a set of action declarations specifying the types of events that can happen in an execution. We can define execution types using the keyword execution:

typedef **execution** {*list of action declarations*} *Name*;

As we shall see, execution types are useful in ensuring the correct use of event processing agents, particularly connecting them to work together.

Example 1: *The NetMngmt execution type*

If a network monitoring system generates load warnings, connect times, and messages on various topics, we can specify its execution type as consisting of the following event types:

```
typedef  execution {
    action Warning(Network_Path Route; Real Load);
    action Alert(Node_Type Node; Real CPU_Load, Memory_Allocation;
                                  Int 1 .. 5  Severity; Time_Type Time);
    action ConnectTime(Network_Path Route; Time_Type Time);
    action Send(Subject_Type Subject; String Message, Id; Time_Type Time);
    action Acknowledge(String Id; Time_Type Time)
  }  NetMngmt ;
```

[1] See Section 5.1 for a definition of "event."

Suppose NetMngmt is the type of execution a network monitor is expected to deal with. Its events are classified into five subtypes, with the action names Warning, Alert, ConnectTime, Send, and Acknowledge. These are the types of events it expects to deal with and should be programmed to handle.

Example 2: *The ATM-Use execution type*

A similar example is a simple automated bank teller system (ATM). Its execution type might be specified according to three actions that it allows customers to perform at the ATMs:

```
typedef execution {
    action Deposit(Dollars Amount; Account_Type Accnt);
    action Withdraw(Dollars Amount; Account_Type Accnt);
    action Transfer(Dollars Amount; Account_Type From_Accnt, To_Accnt)
  } ATM−Use;
```

If an ATM's execution type is ATM−Use, we know that its executions can consist only of events with the action names Deposit, Withdraw, and Transfer.

Example 3: *The SupplyChainEvents execution type*

An execution type for a supply chain

```
typedef execution {
    action RFQ(RFQId Id; ProdSpec Spec; Dollars Price; Quantity Num; Schedule ...);
    action Bid(Vendor VId; BidId Id; ProdSpec Spec; Dollars Price; Quantity Num; ...)
    action Order(OrderId Id; CustId Customer; PartsOrder Data; AccntNo Accnt; ...);
    action Confirm(OrderId Id; PartsOrder Data; AccntNo Accnt; Schedule Dates; ...);
    ...
}   SupplyChainEvents;
```

The supply chain execution type defines the types of events that we used in Chapter 2 to illustrate the global event cloud. In real life, of course, it will have many more action types in it. Execution types will eventually be the subject of standardization for particular industries, analogously to the standards for sets of message types today—for example, the ISO 15022 standard for for messaging to execute transactions in financial markets.

8.3.5 Subtyping of Executions

Execution types have a simple subtyping rule:

type *T1* is a *subtype* of *T2* if $T2 \subset T1$ —*T1* *contains* *T2*.

This rule means that T1 is a subtype of T2 if the set of actions in T1 contains all the actions in T2.

This is very similar to the object-oriented subtyping rule: type *colored point* is a subtype of type *point*. A colored point contains the X and Y coordinates of a point and the additional color component. The importance in object-oriented programs is that any function that computes on *point*s also computes on *colored point*s because colored points contain all the data it needs—but not conversely. The O-O rule is that you can always evaluate a function on an object of a subtype of a function's input type.

Execution types turn out to be useful when we build networks of agents. Event processing agents (EPAs) are typed with their input and output execution types. This lets us analyze whether or not it is sensible to feed the output of one agent into the input of another. Here's how we do it. Composition of EPAs follows the same rule as function composition in typed languages. Namely, if EPA1 outputs to EPA2, the execution type of the output of EPA1 must be a subtype of the execution type of input expected by EPA2. So, it's just as if EPA1 is pumping out colored points and EPA2 is computing on points.

It is easy to see that we shouldn't try, for example, to hook up a network monitor agent and an ATM agent, because they process entirely different types of events. Such a hookup would be a complete waste of time.

8.4 Attributes of Events

RAPIDE-EPL events have a set of *predefined attributes* that are common to all events. Not all attributes need to have defined values in an event.

Attributes give basic information about events. This includes, for example, their timestamps, their *origin* (the system component that generated them), their *destinations* (the components that received them), and information about what events caused them. Attributes give us an ability to write patterns that more precisely match the events we want to match, and to trace through event hierarchies.

There are *public* and *private* attributes. Public attributes can be named and used in patterns just like the declared parameters of actions (examples come later). Private attributes cannot be named in patterns—at least, not explicitly. They are used by tools such as the semantic checker to do type

Table 8.1: Predefined Attributes

Attribute Name	Meaning
name	Action name of the event
origin	Object in the target system whose execution created the event
thread	Thread in the target system that created the event
countervalue	Counter value of the thread that created the event
archstamp*	Target architecture information related to the event, such as destinations and connections to destinations traveled by the event
timestamp start, end	Start and end times of the event according to clocks in the target system
causality	References to immediate predecessors in the causal history of the event
point-of-creation*	Method call in the target system that created the event
declaration*	Reference to the action declaration defining the subtype of the event
trigger set*	References to events that were aggregated to create this event

checking or the pattern matcher to compute causal history, or animation tools to display event movements on architectural diagrams.

Table 8.1 shows a complete list of the predefined attributes. The ones that are starred (*) are private and cannot be used to write patterns.

The meaning or interpretation of event attributes must be defined for each target system. Interpretations of most attributes are defined in *adapters,* whose job is to monitor the target system and translate the target system's events into RAPIDE-EPL events (we will discuss adapters in detail later).[2] In some cases, an attribute may not be meaningful for a particular system. Attributes all have default initial values. So, if an attribute isn't defined for some target system, its value in all the events will be its default value.

The interpretation of attributes such as origin and archstamp can vary greatly in different target systems. For example, an origin could be a module in one system, an application in another, and a thread in a third. The timestamp attribute can give an interval consisting of the start and end times of an event with respect to a clock. If there is more than one clock, there may be multiple timestamps, one for each clock, giving a time interval from the reading of one of the clocks together with a reference to the clock.

[2]Adapters are sometimes also called *loggers*. This is a historical aberration in CEP.

The causality attribute refers to the events that had to happen in the system for this event to happen. It is often a vector of event Ids. The interpretation of "had to happen" is system dependent and will be incorporated into the adapter for events in that system. The trigger set is a set of references to events that triggered an aggregation rule to create this event. A trigger set encodes hierarchical aggregation (or *vertical causality,* Section 1.3.2) and is available to tools that make drill-down tracking possible.

The interpretation of event attributes may depend upon the programming language in which the target system is written, its components (such as the networking middleware), and its architecture.

8.5 Basic Event Patterns

The simplest event patterns are called *basic patterns.* A basic pattern matches single events. The syntax of a basic pattern is

```
(list_of_variable_declarations)  action_name ( list_of_parameter_expressions )
```

The variables are declared in a list, followed by an action name and a list of parameters, each of which can be either a declared variable or an expression. The parameters must have types corresponding to the formal parameter declarations of the action.

A basic event pattern can match events that have the same action name. To match an event, the variables in the pattern must be replaced by data values in the event to make an instance of the pattern that is identical to the event.

Example 1: *A basic pattern for money transfers*

A basic pattern matching all money transfers from a given account is

```
(Dollars  X,  Account_Type  A)  Transfer (X,  Accnt#100,  A)
```

The action name is Transfer, and the variables are X and A. The Transfer action is shown in the ATM–Use example (see Section 8.3.4, Example 2). Accnt#100 is a constant account number for the From_Accnt parameter. To make a match, we can replace X by any dollar amount and A by any account number. This will give us an *instance* of the pattern that must be identical with the event we are trying to match.

For example, the pattern matches Transfer(10, Accnt#100, Accnt#5) if X is replaced by 10 and A is replaced by Accnt#5. Similarly, the pattern matches Transfer(105, Accnt#100, Accnt#21505) if X is replaced by 105 and

A is replaced by Accnt#21505. But the pattern cannot match Transfer(105, Accnt#5, Accnt#21505) because Accnt#100 ≠ Accnt#5.

Example 2: *A basic pattern for engine parts*

A basic pattern matching all orders for engine parts is

```
(OrderId Id, CustId Customer, AccntNo Accnt) Order(Id, Customer, EngineParts, Accnt);
```

The Order action is shown in the SupplyChainEvents example (Section 8.3.4, Example 3). All parameters in this pattern are variables except the PartsOrder which is a constant called EngineParts. So this pattern has an instance which matches any Order event for engine parts—replace each variable by the corresponding data parameter in the event.

The simplest case of matching is when the pattern does not contain any variables—called a *constant pattern*. In this case, a basic pattern matches an event if it has the same action name as the event, and each expression in its data parameters evaluates to the corresponding value in the event's data parameters.

Example 3: *A constant pattern*

A constant pattern is

```
Deposit(1000, Accnt#123)
```

This basic pattern matches any single event in which the name of the action is Deposit, the Dollars amount is 1,000, and the account is Accnt#123. So it matches events like the following:

```
Deposit(1000, Accnt#123), Deposit(500 + 500, Accnt#123), ...
```

8.6 Placeholders and Pattern Matching

The variables in patterns are called *placeholders* because they occupy the places in a pattern that are "open." Matching is a game of trying to fill the open places in a pattern with values (also called "objects") so that the instance of the pattern is identical to the event or poset that we are trying to match.

8.6.1 Matching Basic Event Patterns

A basic event pattern *matches* an event if when its placeholders are replaced by objects, the resulting instance of the pattern is identical to the event. Replacing placeholders with objects to make a match is subject to two conditions.

- The type of object that replaces a placeholder must be the same as, or a subtype of, the type of the placeholder.
- A placeholder must be replaced by the same object at all of its positions in a pattern in any one match.

For a basic pattern to match an event, it must have the same action name.

8.6.2 Placeholder Bindings

The result of a successful match of a pattern to an event (or more generally, a poset) is an association of placeholders with objects that replaced them in the instance that matched the event. This is called a *binding* of placeholders to objects.

A binding is usually represented as a set of pairs consisting of a placeholder and an object, <placeholder ← value>, meaning "replace the placeholder by the value."

Example 1: *Matching bids in an RFQ process*

```
// basic pattern
(Vendor VId, BidId Offer) Bid(VId, Offer, EngineSpec#10, $2,000, 5000)

// event
Bid(Vendor#5, RF#20, EngineSpec#10, $2,000, 5000)

// binding that results in a match
{<VId ← Vendor#5>, <Offer ← RF#20>}
```

The pattern has the action name Bid in our supply chain events (see the declaration of Bid line in Section 8.3.4, Example 3). It has a placeholder, VId, for the Vendor parameter and a placeholder, Offer, for the BidId parameter of the Bid action. The other parameters have constant values for the ProdSpec, Price, and Quantity, so the pattern will match Bid events that have those constant values. The example shows an event and the placeholder binding that results in a match. If we replace VId by Vendor#5 and Bid by RF#20, we get an instance of the pattern that is indentical to the event.

Notice in this example that constants in patterns are very restrictive. Often we want to match a range of values, say, for price or quantity, rather than one value. We will see how to do this with context guards later.

Example 2: *Using a predefined attribute in a pattern*

```
// pattern
   (ATM_Machine M; Dollars D) Deposit(origin is M, D, Accnt#123)

// event
   Deposit(origin is ATM3, 1000, Accnt#123)

// binding resulting in a match
   {<M ← ATM3>, <D ← 1000>}
```

This pattern matches Deposit events from our ATM–Use actions (see Section 8.3.4, Example 2). The pattern contains a placeholder, M, for the prefedined attribute, origin. It uses a parameter-naming notation (**is**), which we will discuss later.

We are assuming that in the ATM–Use system, the actual ATM can be an origin recorded in the predefined origin attribute in the events it creates. This pattern will match events from any ATM that deposit any amount to a fixed account, Accnt#123. The event in the example is a deposit orginating at ATM3 of $1,000 to Accnt#123. The binding shows that M must be replaced by ATM3 and D by $1,000 to make the pattern match the event.

The golden rule about matching is that in order to match a pattern, a placeholder can be bound to only one object in all its occurrences in the pattern. So, if a placeholder occurs more than once in a pattern, a matching event or poset must have the same data at those positions.

Different matches of a pattern usually (but need not) result in different placeholder bindings.

Here are some examples of basic patterns using placeholders.

Example 3: *Placeholders in basic patterns*

```
// 1. Any transfer of any amount from and to the same account
     (Dollars D, Account_Type A) Transfer(D, A, A);

// 2. Any event originating from ATM3
     (event E) E(origin is ATM3);
```

The first pattern has the same placeholder, A, as both the From_Accnt and To_Accnt parameters of a Transfer action in ATM–Use (see Section 8.3.4, Example 2). So, it will match events in which some unspecified amount of money is transfered from any account to the same account. It will match events such as the following:

```
Transfer(10, Accnt#123, Accnt#123),
 if the binding is {<D ← $10>, <A ← Accnt#123>},

Transfer(25, Accnt#47, Accnt#47),
 if the binding is {<D ← $25>, <A ← Accnt#47>},
```

The second pattern shows a powerful use of a placeholder of the predefined **event** type. It matches any event generated by ATM3. E will be bound to the event, whether it is a Deposit, Transfer, or Withdraw. This is a succinct way to write a pattern to monitor a particular ATM. If the event is

```
Transfer(origin is ATM3, 10, Accnt#47, Accnt#123),
```

the binding is

```
{<E ← Transfer(origin is ATM3, 10, Accnt#47, Accnt#123)>}
```

8.6.3 Notation to Aid in Writing Patterns

To emphasize the role of placeholders, RAPIDE-EPL allows a "?" as a prefix to a placeholder. The use of "?" is optional. It helps distinguish the variable parts of a pattern from the constant parts. Some of the previous examples of basic patterns can be written as follows:

```
(Dollars ?D, Account_Type ?A) Transfer(?D, ?A, ?A);
(Vendor ?VId, BidId ?Offer) Bid(?VId, ?Offer, EngineSpec#10, $2,000, 5000)
```

Naming Parameters

A common error in writing a basic pattern is misordering the placeholders in the list of parameters of the action name. That is, the order of the placeholder parameters in the pattern is not consistent with the order of the parameters in the action declaration. To prevent this kind of error, each parameter in a basic pattern can be explicitly associated with the name of an action's formal parameter. You just use the parameter from the action

declaration to name the parameter in the pattern. The notation for doing this is

action parameter name **is** *pattern parameter*

This is called *naming* the parameters in the pattern and is an optional notation.

Example 1: *A basic pattern written in named parameter form*

```
(Dollars ?D, Account_Type ?A, ?B) Transfer(To_Accnt is ?A, Amount is ?D,
                                           From_Accnt is ?B );
```

Look at the Transfer action declaration in ATM–Use (see Section 8.3.4, Example 2). The placeholder parameters in the pattern here are written in a different order from the order in which the action's parameters are declared. But it doesn't matter because we have associated each placeholder with the action's parameter that it corresponds to. So in this pattern, the To_Accnt is ?A, the Amount is ?D, and the From_Accnt is ?B.

Omitting Parameters

A useful feature in writing patterns is to omit a parameter whose binding is irrelevant to the matches you want. That means that you don't care about the omitted parameters, so any value will match them. But to do this without ambiguity, you must name the parameters you do use in the pattern so that it is obvious which of an action's parameters you want to include.

Example 2: *Omitting action parameters*

```
  Deposit(account is Accnt#123);
(Account_Type ?A) Transfer(From_Accnt is ?A, To_Accnt is ?A);
```

Here the Deposit pattern matches any Deposit event to account Accnt#123. We are not interested in the amount. The Transfer pattern matches any transfer from and to the same account. We are not interested in the amount of the transfer, but only the accounts where such a transfer happens.

Using an Event's Public Attributes

The public attributes of events (see Section 8.4) can be used to write more precise patterns. The attribute name is used in the named parameter form. For example, there's an attribute called the origin of an event. It denotes the component in the target system that generated the event. That component may be an object or a module or a thread, depending upon the system. The origin attribute's value can be either a name or a reference to that component.

Example 3: *Monitoring all* withdrawals *generated at a particular ATM*

```
Withdraw(origin is ATM3)
```

Another useful public attribute is the timestamp of an event. Because timestamps are used frequently in patterns, there are special notations such as **at** and **after**, for referring to timestamps that are described later in Section 8.8.3.

Example 4: *Filtering out supply chain events according to their timing*

```
(OrderId ?Order) Order(Id is ?Order, end is 12:00)
(RFQ ?R) RFQ(Id is ?R, Spec is EngineSpec) after 12:00
```

The Order pattern will match those order events that happen over a time interval that ends at 12:00. The binding will contain the OrderId of those events.

The RFQ pattern will match those RFQ events for engine specifications that happen after 12:00 and will bind the Id of the RFQ.

8.7 Relational Operators and Complex Patterns

Relational operators let us write patterns that specify two or more events and a relationship between them. Relational operators are needed to write patterns that match complex behavior in a system.

In the simplest case, relational operators specify how two events are related—for example, whether the events must happen independently or

one must cause the other, whether they must happen one before the other or at the same time, and so on. In general, we can use relational operators to specify how two posets are related. So we can start with basic patterns and build more and more complex patterns.

Relational operators are binary operators. A binary relational operator expresses a relationship between two posets. Patterns written with relational operators are called *complex patterns* to distinguish them from *basic patterns,* which specify single events.

Example 1: *Complex patterns illustrating use of relational operators*

```
1.  (Dollars X) Withdraw(X, Accnt#123) ⟶ Deposit(X);

2.   Withdraw  ||  Withdraw;

3.  (Event E, E') E(origin is "Bonnie") ~ E'(origin is "Clyde");
```

The first pattern uses the causal operator, ⟶. It matches whenever a Withdraw event from account Accnt#123 causes a Deposit of the same Dollar amount (to any account). So the pattern matches posets consisting of two causally related events, a Withdraw from Accnt#123 and a Deposit of the same sum of money. Whenever the pattern matches, X is bound to the Dollar amount. Figure 8.2 shows a poset that contains exactly one match of this pattern.

The second pattern uses the parallel operator, ||. It matches any two independent Withdraw events. The parameter values in the events do not matter; only their independence determines whether they match the pattern. Figure 8.2 shows a poset that contains two matches of this pattern. If the relational operator in pattern 2 was ~ instead of ||, there would be three matches (see the third example).

The third pattern uses the "any" relationship operator, ~, and placeholders that have the most general type, the Event type. This pattern matches any two events, provided "Bonnie" performs one of them (is its origin) and "Clyde" performs the other. Since we don't know what these desperadoes might do, looking for any action rather than specific actions is the best strategy. The events may be in any relation to one another. This means that the pattern can match events that are causally related or that are independent. Whenever the pattern matches, E and E' will be bound to the events. There are six matches in Figure 8.2. The poset shows "Bonnie" and "Clyde" as separate threads of control that generate events and synchronize at two points.

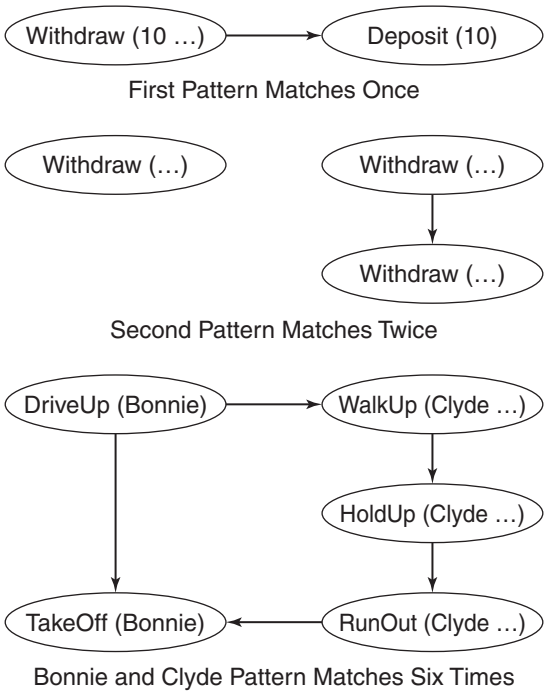

Figure 8.2: Examples of matches of complex patterns

8.7.1 Relational Operators

There are three categories of relational operators: *structural, logical,* and *set* operators. Table 8.2 shows a complete list of the relational operators in RAPIDE-EPL and what they mean. P and Q are patterns, either basic patterns that match single events or complex patterns that match posets.

Structural operators specify the causal structure and timing of matching posets. For example, $P \longrightarrow Q$ tells us that the events matching P must all be in the causal history of all the events in the match for Q. Note, by the way, that the events matching P don't have to immediately precede those matching Q. For example, a grandfather is a causal ancestor of a grandchild.

The independence operator, $P \parallel Q$, requires that all the events matching P must be independent of all the events matching Q.

The timing operator, $P < Q$, specifies that all the events matching P must have timestamps less than any timestamp of an event in the match for Q. So it specifies a similar structure as $\longrightarrow$ but for timing instead of

Table 8.2: Relational Operators in RAPIDE-EPL

Operator	Name	Description
Structural operators		
$P \longrightarrow Q$	causes	A matching poset consists of two subposets, one matching P and one matching Q so that all events in the match of Q are caused by all the events in the match of P.
$P \parallel Q$	independent	A matching poset consists of two subposets, one matching P and one matching Q so that each event in the match of P is independent of every event in the match of Q, and conversely.
$P < Q$	before	Timing: a matching poset consists of two subposets, one matching P and one matching Q so that any event in the match for P has an earlier timestamp than all events in the match for Q. If there are multiple clocks, a particular clock, C, can be referenced as a parameter of $<$.
Logical operators		
P **and** Q	and	The events in a matching poset must match both P and Q.
P **or** Q	or	The events in a matching poset match P or Q.
P **not** Q	not	A matching poset must match P and not contain any subposet that matches Q.
Set operators		
$P \cup Q$	union	A matching poset consists of two subposets, one matching P and the other matching Q.
$P \sim Q$	disjoint union	A matching poset consists of two disjoint subposets; one matches P and one matches Q.

causality. If events have timestamps from more than one clock, the relevant clock is an explicit parameter of the $<$ operator, written as $<_C$.

Logical operators require a poset to match a logical combination of two patterns. It must match *both* patterns, *either* pattern, or one pattern and *not* the other.

The *set* operators, *union* ($\cup$) and *disjoint union* ($\sim$), require a poset to consist of subposets that each match one of two patterns but don't require any causal or timing relationship between the events in the subposets (that is, no structure). For example, P **and** Q requires the poset to match both patterns, whereas $P \cup Q$ requires the poset to consist of two subposets, each matching one of the patterns.

RAPIDE-EPL contains a rich set of relational operators because it was developed for research into CEP. It is unclear which of the logical and set operators are the most useful. Implementation of efficient pattern matching for some of these operators is challenging and demands smart algorithms.

8.8 Guarded Patterns

We can rarely write a pattern without wanting to restrict the parameters of the events. As we saw in STRAW-EPL context tests, in Chapter 6, we might want to deliver messages according to the data they contain or whether the sender passes a credit check. We can do this with a feature of RAPIDE-EPL called *guards*.

A guarded pattern has this syntax:

pattern **where** *Boolean_ test*

The guard is a Boolean valued test following **where**. Its meaning is that it restricts the matches of the *pattern* to those matches for which the Boolean test is true. We often call the guard in a pattern the *where clause*.

8.8.1 Content-Based Pattern Matching

Guards can refer to data in events (the *content* of the events). This is called *content-based matching*.

Example 1: *Testing the content of messages*

```
(Dollars ?LatestPrice, ?LastQuote)
    StockQuote(IBM, ?LatestPrice, ?LastQuote) where ?LatestPrice > ?LastQuote + $5;
```

This pattern will match stock quotes for IBM stock only if the latest price is $5 more than the last quote. The guard uses the content of the StockQuote event to make the test. Typically, these kinds of tests are used in content-driven message delivery.

Example 2: *Good banking behavior*

```
(Dollars ?X, ?Y; Account ?A)
     (Deposit(?X, ?A) ⟶ Withdraw(?Y, ?A)) where ?Y < ?X;
```

The pattern, Deposit(?X, ?A) ⟶ Withdraw(?Y, ?A), will match any pair of causally related Deposit and Withdraw events on the same account. The guard restricts the matches to those pairs for which the amount withdrawn is less than the amount deposited.

When the pattern matches, the placeholders ?X, ?Y are bound to the data values for the amounts of the deposit and the withdrawal. Those values are then used to evaluate the guard. If its value is true, the guarded pattern matches.

8.8.2 Context-Based Pattern Matching

Guards can also refer to information outside the events, such as database queries or values returned by method calls (that is, the *context* in which the pattern matching happens).

Example 1: *Context-based message filtering*

```
(OrderId ?Id, CustId ?C, AccntNo ?A)
    Order(?Id, ?C, ?A) where CreditCheck(?C) = Pass and Active(?A);
```

This pattern matches Orders from customers if they pass a credit check and their accounts are active. Both of these tests in the guard refer to the *context* in which the Order is received—the status of the customer's credit and the status of their account at that time.

The Boolean test in the guard may refer to the values of placeholders in the pattern and objects from the context in which the pattern is matched. Matching a guarded pattern proceeds in two steps. The first step is to match the unguarded pattern; if there is a match, all the placeholders in the guard must be bound to values by the match. Then the guarded pattern matches if the guard is true when those values are substituted for the placeholders. The guard is evaluated after the unguarded pattern matches. An error results if there is an unmatched placeholder in a guard when it is evaluated—except in some special circumstances, which we mention later.

Context-based pattern matching is more difficult to implement efficiently to allow high throughput of events than content-based matching. Context references in guards make it more difficult to organize large numbers of patterns for sublinear searches for matches. Details are beyond the scope of our discussion here.

8.8.3 Temporal Operators

RAPIDE-EPL provides some operators that simplify writing guards that refer to the timestamps and start and end times of events. They are called *temporal operators*.

- **at**: Applies only to basic event patterns. For example, if P is a basic event pattern, P **at** 3:00pm matches an event E if E matches P and its timestamp is 3:00pm.

  An unbound variable can be a parameter of the **at** operator. If T is a variable of type Time, P **at** T matches an event E that matches P, and a result of the match is to bind T to the timestamp value of E.
- **after**: Applies to complex patterns. For example, P **after** T matches posets that match P and contain events all of whose start times or timestamps are greater than T.
- **during**: Applies to complex patterns. For example, P **during**(T1, T2) matches posets that match P and contain events all of whose start times or timestamps are greater than T1, and all of whose end times or timestamps are less than T2.

Each of these operators is equivalent to writing guards that refer to the timestamps or start and end times of the basic events in the pattern. Normal use of these operators assumes a single global clock in the system. If there are multiple clocks, a particular clock whose readings are to be used can be named as an argument of any of these operators.

8.9 Repetitive Patterns

Many systems repeat the same behavior over and over. A typical example is found in communication protocols that repeat behaviors such as "if no acknowledgement, time out and resend"—lack of success requires the protocol to keep on attempting to send a message. So event patterns must be able to express repetitive behavior.

In RAPIDE-EPL we first express the pattern of the behavior that is repeated. Say this is a pattern, P. Each repetition is another poset that matches P. Next, we need to express *how* P repeats—that is, the causal or timing relationship between one poset that matches P and the next poset that matches P. It can be any of the *structural* relational operators. We do this by writing a prefix to P that specifies the number of repetitions and the relationship between each repetition and the next one—called the *repetition profile*.

So, to express a pattern consisting of repetitions of P, the syntax is:

[*number of repetitions* **rel** *relational operator*] P;

The part inside the square brackets is the repetition profile. It expresses the number of repetitions and the structural relationship between each match of P. P is the body that is being repeated.

The repetition profile can have a counter variable that counts the number of repetitions, and that counter can be used as a parameter of the pattern body—just like a **for** loop.[3]

A repetitive pattern matches a poset if the poset consists of the specified number of subposets, each matching the body, and each of the matches is related by the relational operator in the prefix. The number of repetitions can be specified in various ways: a specific number or any number. We use "*" for any number.

Example 1: *Some repetitive patterns*

1. [∗ **rel** ⟶] Deposit
2. [1..10 **rel** ⟶] Deposit
3. [I **in** 1 .. 10 **rel** ∼] Deposit(I)
4. [∗ **rel** ∼] (Msg ?M)(Send(?M) ⟶ (Ack(?M.header) **or** Time_Out))
5. (Msg ?M)([∗ **rel** ∼](Send(?M) ⟶ (Ack(?M.header) **or** Time_Out)))

Each of these examples illustrates a different feature of repetitive patterns.

The body of the first pattern matches any Deposit event. The repetition profile specifies any number (*) of matches related to one another by ⟶. So this repetitive pattern expresses "any number of Deposit events, where each event is causally related to the previous one." A matching poset must be a causally ordered chain of Deposit events.

The (*) repetition pattern has what we call *maximal match* semantics. This means that it will match only the chain of Deposit events that consists of the maximal number of events, not a subchain consisting of some of those events.

---

[3]For this reason, RAPIDE-EPL specifies the repetition in a prefix profile rather than in postfix notation like regular expressions.

The second pattern differs from the first one only in specifying exactly ten Deposit events. It uses a common range notation, $1\,..\,10$, to specify a finite number of matches.

The third pattern uses a repetition counter variable, I, as a parameter of the body. So, the pattern body Deposit(I) changes for each repetition. The repetitive pattern matches a poset consisting of Deposit(1), Deposit(2), ... up to ten, with any relationship between the events. Some could be dependent and some independent.

The fourth pattern is one that is repeated an arbitrary number of times. But it illustrates some "fine points." The pattern body is

(Msg ?M)(Send(?M) ⟶ (Ack(?M.header) **or** Time_Out))

which matches a poset consisting of two events, a Send(O) event with a message object, O, that binds to ?M, which causes either an Ack event with the header of O as its parameter or a Time_Out event. The body can be matched repeatedly. Because ?M is declared in the body, it can be bound to a different message O on each of the matches. So the pattern matches a poset consisting of any number of pairs of events, either Send(O) ⟶ Ack(O.header), or Send(O) ⟶ Time_Out; each pair can be dependent or independent of other pairs (the $\sim$ relation) and can have a different message O.

The fifth pattern matches similar posets to the fourth one, except that the placeholder ?M is declared before the repetition profile. Therefore, it must be common to all the repetitions of the body. So a single binding for M must be common to all the repeated matches.

Example 2: *A pattern that matches a supply chain bidding process*

```
(RFQId ?Id, Time ?T1)(RFQ(?Id) at ?T1 ⟶
   [* rel ~] (Time ?T2) Bid(RFQId is ?Id) at ?T2 where ?T2 < ?T1 + Bnd);
```

This example uses the RFQ and Bid actions in the supply chain example in Section 8.3.4 (Example 3). It matches the kind of electronic bidding process that might be expected in B2B activites of the electronic enterprise (see Chapter 2).

Let's look at the pattern in detail. It matches an RFQ event with an ?Id that happens at time ?T1 and a poset of Bid events that are caused by the RFQ event. There can be any number (*) of Bid events, in any relation ($\sim$) to one another, provided they all occur within a time bound, Bnd, of the RFQ event. The Bid events must all contain the ?Id of the RFQ event.

So this pattern "picks out" from the global event cloud all the Bid events in response to an RFQ that happen within a time limit.

8.10 Pattern Macros

Writing patterns is made much easier by an abstraction feature. An abstraction feature is useful in various ways.

1. First, it lets us abstract commonly used patterns and name them. This lets us shorten the notation for patterns and write readable patterns.
2. Second, we need to build up libraries of patterns for each application domain, say, network protocols, control systems, distributed transaction systems, supply chains for various industries, and so on—each domain has its own common patterns.
3. Third, when we specify hierarchical systems, we need to organize our patterns hierarchically too.

Pattern macros are a simple abstraction feature that helps with all these practical problems.

If we want to define a pattern macro called PM, we write:

pattern PM (*parameter list*) {*pattern*};

PM is the name of the macro. It names the part in braces, '{' ... '}', which must be a pattern. This is called the *body* of the macro.

The way we use pattern macros is to call them in patterns. So, in some pattern, we can write a call like this:

... PM(*actual parameter list*) ...

During pattern matching, a point is reached at which a macro call such as PM(...) must be matched. At this point in the matching, the parameters of the call have certain values, either objects or placeholders that haven't been bound yet. The macro call is replaced by an instance of the macro's body. To do this, the parameters in the body are replaced by the corresponding values. The resulting instance of the macro's body replaces the macro call in the pattern. This is called *macro expansion* because the call is "expanded" into an instance of the body, which is usually a lot bigger.

We could do many macro expansions with a text editor except when the macro is recursive—that is, the macro contains a call to itself. So macro expansion takes place at runtime—during pattern matching. And macro expansion is *lazy*—expansion takes place only if a match of the macro call is needed to match the pattern containing the macro call.

Example 1: *A pattern macro to shorten notation*

```
pattern Reply(Msg X) { Ack(X.header) or Time_Out };

[ * rel ~] (Msg M)(Send(M) ⟶ Reply(M))
```

We want to shorten the "send causes an acknowledge or time out" pattern in an example in the previous section. So we define the "acknowledge or time out" piece of the pattern to be a macro called Reply. Now we can specify the pattern more succinctly with a macro call to Reply. It is shorter and more readable. The rewritten pattern specifies that each send causes a reply, which happens to be "..." (an instance of the Reply body).

Example 2: *Another pattern macro to shorten notation*

```
pattern Transaction() { (Msg M)(Send(M) ⟶ (Ack(M.header) or Time_Out)) };

[ * rel ~] Transaction();
```

This second pattern macro shows that if we think of the "send causes an acknowledge or time out" pattern (the part that is being repeated) as a transaction, rather than a send and a reply, we can write the example even more succinctly.

As we said before, we have to be just a little careful about how we define macro expansion of macro calls, because pattern macros can be recursive. If we just dive in and do naive macro expansion, a recursive macro call will keep on being expanded, and we will never stop. This happens in all macro facilities that can be recursive. So, macro expansion is *lazy*. A macro call is expanded during matching of the pattern containing the call, "as needed" to do the match.

Example 3: *A recursive pattern macro*

```
pattern Saving() is Deposit ⟶ Saving() or Empty();

  Saving()
-- matches the same finite posets as:
[ + rel ⟶] Deposit
```

Saving is recursive. It says, "Match a Deposit that causes either another match of the pattern Saving or the empty poset—that is, it causes no events." It matches posets consisting of one or more Deposit events, all in a causal chain.

Empty is a predefined pattern macro that matches the empty poset. Empty is useful for defining other patterns, as here, where it defines the termination case in a recursive macro.

Macros can be used to define other relational operators. Here is an example of a macro defining a new structural operator, ▷ (*immediate cause*). This expresses a relationship between events P and Q, whereby P is an immediate cause of Q—for example, father and son, but not grandfather and grandson. That is, P causes Q and there is no event, E, such that P causes E and E causes Q.

Example 4: *Immediate cause operator*

```
pattern P ▷ Q is (P ⟶ Q) not (P ⟶ Any ⟶ Q);
```

A new relational operator, ▷(P, Q), is defined using the operators ⟶ and **not**. It matches a poset if P ⟶ Q matches the poset, and there is no nonempty subposet of the matching poset that matches Any and is causally between the matches for P and Q. So, the match for P must be an immediate cause of the match for Q.

8.11 Summary

One of the earliest examples of pattern languages for specifying computer programs is Path Expressions [22], which is conceptually similar to Regular Expressions.[4] Historically, RAPIDE-EPL evolved experimentally from an event pattern language for specifying and monitoring multitasking programs, called Task Sequencing Language [7], which was also rooted in Regular Expressions.

RAPIDE-EPL could be viewed as being designed by taking Regular Expressions of basic event patterns as a basic event pattern language and then adding new features, including the causal (⟶) and independence (||) event pattern operators, predicate guards over complex event patterns, timing operators similar to the ones usually found in simulation languages, strong typing with inheritance, and pattern macros. Today, several pattern languages could be added to in similar ways to be suitable for CEP.

[4] Web search on "regular expressions."