Event Processing In Action

Opher Etzion
Peter Niblett

MANNING

MEAP Edition
Manning Early Access Program

For more information on this and other Manning titles go to
www.manning.com

Preface

ABOUT THIS BOOK

Event processing is an emerging area, the appearance in recent years of various commercial products and open source offerings, making it the fastest growing segment of enterprise middleware. While interest in this subject is growing, gaining a deep understanding of event processing is still a challenge. As it is a relatively new area it is not surprising that several different approaches to event processing have been evolving in parallel. This means that when trying to understand what event processing is about it can be difficult to see the wood from the trees. This is the main intention of this book, to give a deep dive into what event processing is (the wood) and to provide the reader an opportunity to experience this using some of the existing event processing languages or tools (the trees).

This is book is intended for those interested in understanding what's behind event processing technologies and how they should be employed to design and implement applications. This book is the first comprehensive view about event processing for the technical reader, it looks at "event processing" as a generic technology, in a way that fits all the different implementation approaches, and thus familiarizes the reader with the entire wood, and not just with a specific tree. The book provides a deep dive into all the concepts that need to be understood in order to design event processing applications, and guides the reader through these concepts by showing the construction of a single example application that uses event processing.

The interested reader also gets a unique opportunity to see how to implement this application using representatives of the various programming styles that exist today: SQL extension, rule based, graphical oriented, and script oriented languages. The website that accompanies the book provides examples based on this application, with instructions on how to download trial versions of various commercial and open source event processing products. This allows the reader to see the concepts applied in action, to play further with the code, and to devise further examples.

Besides the design concepts the book also discusses implementation issues and the authors' opinion about the event processing of the future. A survey of existing products is provided, as well as a comprehensive set of terminology definitions.

THE INTENEDED READERSHIP

The book is intended for those who want to gain understanding about event processing concepts in depth; the primary audience is a technical audience consisting of architects, designers, developers and students. The book will benefit designers and architects who wish to know how to design applications that use event processing, and developers who would like to understand the relationships between these concepts and current event processing languages and products. It can also serve as a textbook for an academic or professional course on event processing, and for this reason it provides an "additional reading" list along with a few exercises at the end of each chapter.

THE BOOK'S METHODOLOGY

In this book we have taken a top-down approach by describing the concepts (the wood), and then providing the reader an opportunity to view the trees (representatives of the different approaches) and experiment with them through the associated Website. This approach is somewhat different from the bottom-up approach of describing a single language or product. We have taken this approach as there are several different approaches for implementing event processing applications, and the transfer in thinking from one to another is not easy. We use a general model that consists of seven building blocks, which we believe is an effective way to explain the concepts and facilities of event processing. Moreover, we feel that there are advantages in using this level of abstraction when designing and developing applications.

In order to illustrate these concepts we use a single example which we follow throughout the course of the book. This example, based around flower delivery, can be understood with no prior domain knowledge, but nevertheless contains many of the concepts that we discuss in the book.

Terminology does vary somewhat between different event processing products, so in this book we have tried to define all the terms and concepts that we use, and we provide a summary of these definitions in Appendix A. Our definitions are written in an explanatory style rather than in a rigorously formal style, so as to make them accessible to a broad audience. Where possible we are using definitions that are consistent with the terminology established by the Event Processing Technical Society (EPTS) however, our scope of terms is much larger.

THE BOOK'S STRUCTURE

The book has four parts. The first part consists of chapter 1 and chapter 2 and is an introduction to the subject and to the terms and concepts that we are using. Readers who are already familiar with Event Processing can just browse through it, noting the definitions we use, without reading it thoroughly.

- Chapter 1 is the entry point, with some examples and some basic terms, and introduces the Fast Flower Delivery example.
- Chapter 2 explains basic architectural and programming principles.

The second part of the book goes through the concepts in detail, showing how the seven building blocks can be used to describe an event processing application, illustrating them in the context of the Fast Flower Delivery example.

- Chapter 3 deals with events: types, event schema descriptions, event relationships.
- Chapter 4 deals with event producers: types of event producers, ways events are obtained from a producer.
- Chapter 5 deals with event consumers: types of event consumers, and some current examples.
- Chapter 6 deals with the event processing network, a key concept in event processing and with the associated building blocks, such as: event processing agents, channels and global state.
- Chapter 7 discusses the notion of context and its major role in event processing.
- Chapter 8 looks in greater detail at event processing agents that filter and transform events.
- Chapter 9 provides a deeper dive into "event pattern matching", this being the jewel in the crown of event processing.

The third part deals with additional issues that relate to the implementation of event processing applications in practice.

- Chapter 10 provides a survey of implementation oriented issues, both engineering aspects such as scalability, and software engineering aspects such as programming style and development tools.
- Chapter 11 surveys some of the semantic challenges that developers and users of event processing systems should be aware of, in order to avoid semantic anomalies when building event processing applications.

The fourth part, consisting of chapter 12, summarizes the book and provides the author's views about the future of event processing.

TOPICS FOR ADDITIONAL READING

Some topics are mentioned in the book, but are not thoroughly discussed, since such discussion is not vital to achieve to book's goals, and they are the subject of books in their own right.

Business topics, (such as: a review of the types of applications being used, or analysis of the event processing market and its trends) are beyond the scope of the book. The book provides short motivation for the use of Event Processing by means of some examples in

chapter 1, and talks about trends in chapter 12. It provides an additional reading list for the readers interested to go deeper here.

There are many technologies and architectural concepts that are adjacent to event processing, starting from SOA (Service Oriented Architecture), moving through EIP (Enterprise Integration Patterns), and touching other disciplines: BPM (Business Process Management), BI (Business Intelligence), BAM (Business Activity Monitoring) and more. In chapter 2, we provide a brief survey of the relationship of event processing to each of these areas and then provide an additional reading list for the interested reader.

THE EPIA WEBSITE

The book's Website is being hosted by EPTS,

http://www.ep-ts.com/content/view/74/108/

This Website contains the solution of the "Fast Flower Delivery" example that accompanies this book in six different event processing languages representing six different programming styles. Some code samples from these solutions are embedded inside the book, but if you wish to learn a specific language you should download the documentation from the appropriate link on the Website. There is also a link to the editor that will enable you to create a model of this system using the building block language described in this book.

Comments to the reviewers and MEAP subscribers:

This Website is currently still under construction (will be finalized by the end of January 2010), some of the solutions are already there, and some are being checked. The solutions will be validated by that time.

ACKNOWLEDGEMENTS

TBC

1

Entering the world of event processing

"I am more and more convinced that our happiness or unhappiness depends far more on the way we meet the events of life, than on the nature of those events themselves"

- Wilhelm von Humboldt

Some people say that event processing is the next big thing; some people say that event processing is old hat and there is nothing really new in it. Both groups may be right to a certain extent. As with any field that is relatively new there is some fog around it: some of the fog stems from misconceptions, some from confusing messages by vendors and analysts, and some arises because of a lack of standards, a lack of agreement on terms, and a lack of understanding about some of the basic issues.

This chapter covers

- An explanation of what we mean by events, using examples from daily life
- Various examples of computerized event processing in use
- Different reasons for using event-driven computing systems
- The main concepts of the event-driven computing
- Business consideration in using dedicated event processing software
- The "Fast Flower" delivery use case that will accompany us throughout this book and will be demonstrated using various implementations in the book's website:

- http://www.ep-ts.com/content/view/74/108/
- A description of the Website content

Before we can clear the fog surrounding event processing, however, let's start with a look at its background and some examples of event processing in daily life.

1.1. Event-driven behavior in daily life

Our intention in this book is to send some rays of light to disperse the fog, and show a clear, comprehensive, and consistent view of the event processing area. We shall start by explaining the concept of event-driven behavior in dally life.

1.1.1 The notion of event

Before going any further we should clarify what we mean by the word *event*. In this book we use the following definition:

> **Definition**
>
> An *event* is an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain. The word *event* is also used to mean a programming entity that represents such an occurrence in a computing system.

We are using throughout the book terms explanations using this "Definition" block, appendix A provides all definitions lexicographically sorted; you may use it to recall a definition.

As for this specific definition, you will see that this is in fact two definitions in one. The first meaning refers to an actual occurrence – the "something that has happened" in the real world or some other system – and the second meaning is the programming entity that represents it. It's easy to get caught up into a rather pedantic discussion about the difference, but in practice the word *event* is used with both meanings. It is safe to do this, since it's usually easy to tell which meaning is intended from the context in which it appears. However it is worth noting that a single event occurrence can be represented by many event entities, and secondly a given event entity might only capture some of the facets of the occurrence.

As this is such an important term, it's worth commenting on three of the phrases used in it. The first of these is "external system". We could have used the phrase "real world system" here – we have already noted that event-driven computing is largely about "real world events" – but we didn't want people to think we were excluding events that happen in Virtual Worlds, training simulators or similar virtual environments.

Our second phrase is “or is contemplated as having happened”. That is there because it’s possible to have events that don’t correspond to actual occurrences. To explain what we mean, imagine a fraud detection system being used in financial institution. Such a system monitors financial transactions and generates events when it suspects that a fraud is being conducted. In general such systems can generate “false positives”, so further investigation is usually required before you can be sure whether a fraud has actually taken place or not.

Finally there is the phrase “programming entity”. We could have used the word “object” here, but to some readers this might imply that has to be an object as defined in OO programming. In some contexts you might well find events represented as objects, but you can equally well find events that appear in other forms, such as records in a relational database, structures in a language like C or COBOL, or messages transmitted between systems. We have therefore chosen the vaguer expression “programming entity”.

The word *event* is sometimes used in event processing literature to refer to a type or class of events rather than to a specific event instance, for example `Error` or `Account Overdrawn`. In this book we will generally use the phrase “event type” in such cases, unless it is obvious from the context that we mean the type rather than a specific instance.

The Event concept is very simple yet also very powerful. I am writing this chapter in a coffee shop, and since I entered this coffee shop several events have happened; people came in and out and the waitress brought me coffee - nothing really exciting. However imagine what would happen if a robber were to enter the coffee shop and ask people for their money. This would break the peaceful atmosphere and compel people to react. Suppose that someone reacts by surrendering his wallet to the robber, this reaction would trigger more events; after recovering from the shock, the person who was robbed might call the credit card companies to cancel his stolen credit cards, which in turn would trigger further activities. But let's leave scary scenarios and look at how self-service coffee shops work[1]. Some of them work in a synchronous fashion: a customer asks for coffee and pastry, the person behind the counter puts the pastry into the microwave, prepares the coffee, takes the pastry out of the microwave, takes the payment, gives the tray to the customer and then turns to serve the next customer. Another way to organize things is to have the person behind the counter take the order and payment and then continue with the next customer, while in the background there are other people dealing with the pastry and coffee. When both are ready they'll call the customer, or bring it directly to the table, allowing the customer to sit down, take out the laptop and write books while waiting. Figure 1.1 illustrates these two approaches.

---

[1] Note that Gregor Hohpe discussed the synchronous / asynchronous natures of coffee shops and also used diagrams to illustrate the differences, see: Gregor Hohpe: Your Coffee Shop Doesn't Use Two-Phase Commit. IEEE Software 22(2): 64-66 (2005)
http://www.computer.org/portal/web/csdl/abs/mags/so/2005/02/s2064abs.htm

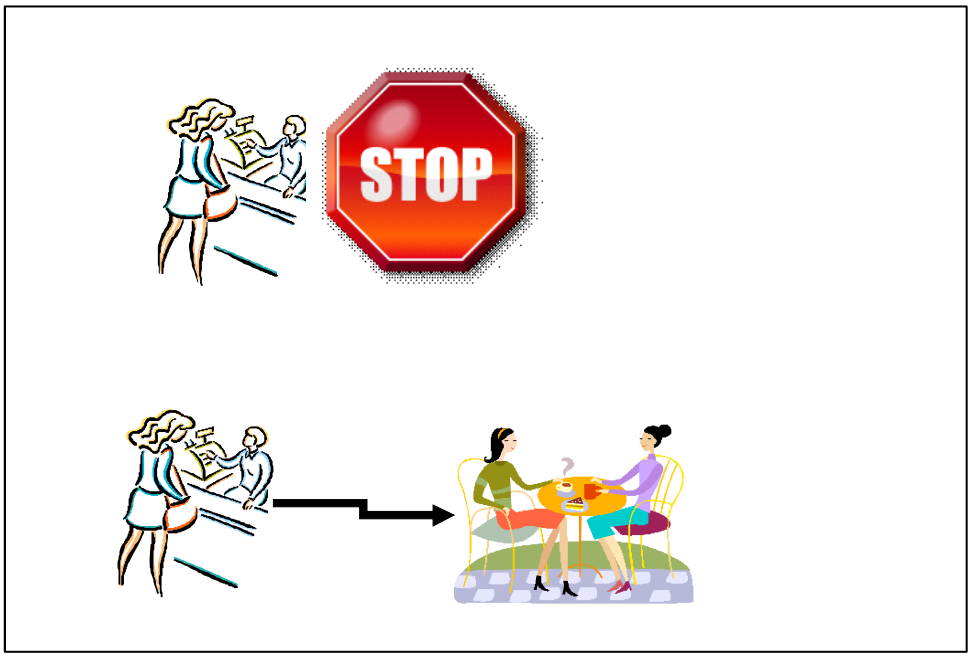


Figure 1.1: Self-service coffee shop examples: the synchronous approach (upper illustration) versus the event-driven approach (lower illustration).

The first approach is more like the way that traditional information systems work; we issue a request, and wait for the response, typically doing nothing in between. The second approach is event-driven. The event – "`order is completed`"- is a combination of two other independent events: "`coffee is ready`" and "`pastry is heated`"... The "`order is completed`" event is a routine event for the coffee shop and customers, while the robbery event is an unexpected event, however both require some reactions. We encounter events all the time in our daily lives; some events are quite basic: the phone rings, an email message arrives, somebody knocks at the door, or the book falls on the floor. Some events may be unexpected: the robbery event mentioned before, coffee is spilled over the laptop and creates a short circuit, a flight is late and we miss the connecting flight. It's also an event when I then find out that my luggage did not arrive. Before the reader gets the impression that all unexpected events are negative, here are some positive ones: winning the lottery, getting a large order from an unexpected customer, finding a significant amount of natural gas under the sea.

Some events can be observed very easily: for example things that we see and hear during our daily activities; some require us to do something first, for example: subscribing to news groups, or reading a newspaper. In other cases we need to do some work in order to detect the event that actually happened, as all we can observe are its symptoms. As an

example, recently my laptop was unable to connect to my wireless home router. This was a symptom of an event which had occurred earlier, and it took 90 minutes of search by a skilled technician before that event could be identified. To take another example, I recently noticed that our consumption of milk at home had gone up and we needed to purchase an additional carton of milk in our weekly grocery shopping. I reached this conclusion when in three consecutive weeks we ran out of milk in the middle of the week, convincing me that this was a consistent phenomenon. In this example "`running out of milk`" is an observable event, while the "`milk consumption increased`" event is a higher level event that can be concluded from observing lower-level events.

The main reason we are interested in knowing that events have occurred is that it gives us the opportunity to react to them. In the previous example I reacted to the "`milk consumption increased`" event by increasing our weekly purchase of milk. There are a lot of events around us, some are outside the scope of our interest, some of them are just background knowledge and do not require any reaction, and some require reaction. We use the word *situation* to refer to this kind of event.

Definition

A situation[2] is an event occurrence that might require a reaction

One of the main themes in event processing is the detection and reporting of situations so that they can be reacted to. The reaction might be as simple as picking up the phone or changing the weekly shopping list, or it might be more complicated. If we miss a flight connection there may be several alternative reactions depending on the time of the day, the airport where we are stranded, the airline policies, the amount of other passengers in the same situation and more.

So we have seen that people quite frequently act as event processors. Let's now move from the world of people to the world of information systems.

1.1.2. Some examples of event-driven computing

Event-driven computing is not new. In the early days of computing, events appeared in the form of exceptions whose role was to interrupt the regular flow of execution and cause some alternative processing to happen. For example if a program tried to divide by zero an exception event would be raised that enabled the programmer to end the program with an

---

[2] The term "situation" has been used by several sources in this context; an example is: Asaf Adi, Opher Etzion: Amit - the situation manager. VLDB J. (VLDB) 13(2):177-203 (2004) http://www.springerlink.com/content/nb1qa1d02vvdre00/

error message, or to perform some corrective action and then continue with the computation process. Later on, events featured in Graphical User Interface systems (such as Smalltalk or Java AWT) where UI components ("widgets") are designed to react to UI events such as mouse clicks or key presses.

In this book we are mainly concerned with computing events that correspond to events that occur in the "real world". Here are some examples that also show the benefits of automated event processing; these examples show different types of processing.

Example 1: A patient is hooked up to multiple monitors that either continuously or periodically perform various measurements on the patient. The measurements take the form of events which are then analyzed by an Event Processing system. A physician can configure this system, on a patient-by-patient basis, so that a nurse is alerted if certain combinations of measurement are detected within a certain time period, and so that if other combinations occur then the physician herself is alerted. This example demonstrates the use of event processing for personalized diagnosis.

Example 2: In an airline luggage handling system an RFID tag is attached to every piece of luggage. There are RFID readers in various places where the luggage moves (sorting device, cart going to the aircraft, aircraft's unloading dock and more). Events from the RFID readers are analyzed to provide luggage control alerts such as: luggage is going on the wrong cart; luggage did not arrive[3] at the aircraft; luggage did not even arrive even at the sorting device; as well as an alert that the luggage approaching the carousel. This example demonstrates the use of event processing for detecting and eliminating exceptions within a processing system.

Example 3: A manufacturing plant with restricted access zones uses RFID tags to monitor compliance with safety regulations. Each person working or visiting the plant carries an RFID tag, and an RFID reader in each zone generates an event when it detects the presence of that person. These events can then be analyzed to detect safety violations: simple ones like a person entering a zone where they are not authorized, or more complex ones like an authorized person working unaccompanied in a zone which requires the presence of two or more authorized people. This example demonstrates the use of geospatial event processing for observation about policy violations.

Example 4: A personal banking system that allows bank customers to set up alerts when certain events occur, such as: the sum of money withdrawn from all my accounts within a single day > $10,000; my investment portfolio has gone up for more than 5% since the start

---

[3] The fact that an event did not occur is one of the most common event processing patterns, called: absence pattern, discussed in Chapter 9. Some people refer to it as "non event".

of the trading day. This example demonstrates the use of event processing for personalized information dissemination

Example 5: A financial institution wishes to detect frauds or a financial regulator wishes to catch illegal trading patterns. They collect events from banking or trading systems and analyze them. Certain patterns of activities might suggest that a person is possibly (but not necessarily) in the process of committing a fraud or other illegal activity. This example demonstrates the use of event processing for detection of evolving phenomena.

Example 6: An emergency control system that informs and directs first responders and people at risk in case of an incident (e.g. fire, leakage of hazardous materials). In this case the event is a report on an incident, and the main focus of the system is the dissemination of information: who should be informed about what and at what time, given the nature of the incident.

Example 7: An on-line trading system that matches buy requests and sell requests in an auction keeping fairness practices (e.g. first person to make the bid has first chance). In this case the complexity is in the matching itself. The events are the buy and sell requests and the matching process is required to match using patterns that apply fairness criteria, such as priority based on order, matching conditions, and prior information about the level of risk of the buyer and seller based on trade history. This example demonstrates the use of event processing to dynamically manage business processes,

Example 8: A manufacturing plant management system that diagnoses mechanical failures based on observable symptoms; in this case the events are symptoms, describing something that does not work properly, and the main function is to find the root cause of these symptoms. This example demonstrates the use event processing for problem determination and resolution.

Example 9: A road tolling system that detects the entry and exit point of a vehicle to a toll road and bills the owner. Vehicles are detected based on analysis of a video stream that captures their license plates. Here the main difficulty is extracting and interpreting the vehicles' license plates from video stream in order to generate the events themselves. This example demonstrates the use of event processing to trigger business processes, where the events need to be obtained as a result of some analysis.

Example 10: A social networking site which starts a multi-party chat when five people from a group are on-line. In this case an event occurs when a person goes on-line or off-line. Event Processing is used to analyze these events so as to decide when to start a chat session. This example demonstrates the use of event processing for real-time collaboration.

These examples are somewhat different from one another, but all of them follow the same pattern: events are reported, sometimes by multiple sources, and some processing of the events is performed. This processing can be made up of several phases and at the end it creates additional events that are consumed by event consumers. The producers of the events may be different from their consumers; In Example 2 (he luggage management system) the check-in process emits an event when luggage is initially deposited and various RFID readers then emit events about the movement of the luggage in the system. The consumer could be either the luggage controller or even the passenger.

In the next section we provide a classification of event processing systems according to their business use.

1.1.3. What are the reasons for using event-driven computing systems?

The examples given in the previous section and many others may be classified as shown in Figure 1.2

- *Real-Time operational behavior*—a common reason for using event-driven computing systems is to be able to change the behavior of the system dynamically in order react to react to incoming events. Matching auction buyers and sellers is an example of this type, the result of the match then determines the subsequent flow of the system. Another example is automatic re-routing of luggage when a passenger's itinerary changes.
- *Observation*—another reason to use event-driven computing systems is to look for exceptional behavior and generate alerts when such behavior occurs. In such cases the reaction, if any, is left to the consumer. The job of the event processing application is just to produce the alerts. Examples of observation are regulation compliance systems, as well as the patient monitoring system described above.
- *Information dissemination*—A third reason for using event-driven computing systems is to deliver the right information to the right consumer in the right granularity at the right time, in other words personalized information delivery. Examples of this type are personalized alerts from banking systems, and the emergency system sending alerts to first responders.
- *Active diagnostics*—here the goal of the event processing application is to diagnose a problem, based on observed symptoms. The mechanical failure case is such an example; a help-desk system is another example.
- *Predictive processing*—Here the goal is to indentify events before they have happened, so that they can be eliminated or at least have their affects mitigated. The fraud detection example is of this kind.

These different classes of use are not exclusive to each other; a specific application may fall into several of the categories that we have listed.

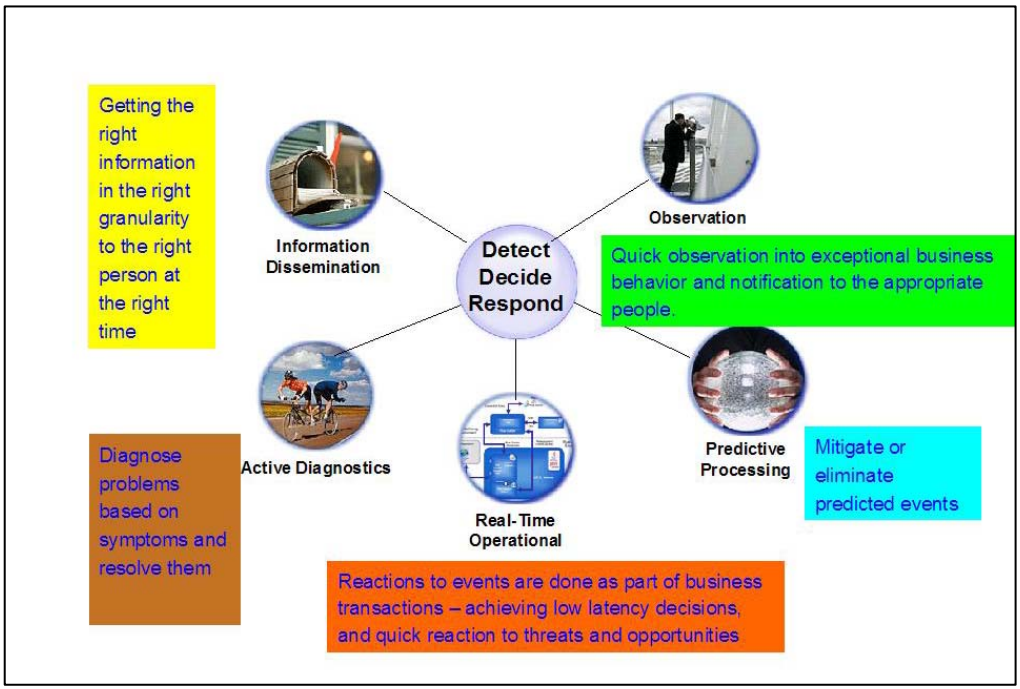


Figure 1.2: Classification of applications that employ event-driven computing for distinct reasons

Now that you know why people utilize event processing, let's discuss some of the main concepts involved with it.

1.2 The main concepts of event-driven computing

In the previous section we introduced the idea of events in real life and their representation in computing systems, and we took a look at the sorts of things that event-driven computing can be used for. We are now going to give some slightly more formal definitions of the main terms that we will be using in the remainder of this book, and the concepts that they describe.

We'll get to some of these definitions shortly, but first you will notice that we have already been using the term e*vent driven computing*. As that's what this book is all about, it's worth taking a minute to explain what we mean by it. We are using this phrase to cover two related ideas:

- *Event-based programming*—Designing or coding applications that make use of events, either directly or indirectly.

- *Event Processing*—Operations that you can perform on events, in particular operations that take a set of one or more events as input and generate further events from them as output. Event Processing operations are used in the analysis of low-level events which featured in many of the examples in the previous section.

It's possible, of course, to write event-based programs without using Event Processing and as we noted earlier people have been doing event-based programming for many years. The three things that distinguish event driven computing, and which open up such a rich range of possibilities, are:

- *Real-world focus*—It deals with events that occur, or could occur, in the real world
- *Decoupling*—the events detected and produced by one particular application can be consumed and acted on by completely different applications. There's no need for producing and consuming applications to be aware of each others' existence, and they may be distributed anywhere in the world. An event emitted by a single producing application can be acted on by many consuming applications. Conversely you can arrange for an application to consume events produced by many different producing applications
- *Abstraction*—The operations that form the Event Processing logic can be separated out from the application logic, allowing them to be modified without having to change the producing and consuming applications

Let's move to an overview of event-driven architecture, which will help organize many of these key concepts.

1.2.1 Event Driven Architecture

Don't be put off by the rather grandiose title of this section. What we are going to look at is the general way that event-driven computing applications are constructed, and the building blocks, or components, that make them up. We will be going into all the details of these components later on in this book

Not all event-driven computing applications are the same of course, but by and large most of them look something like this:

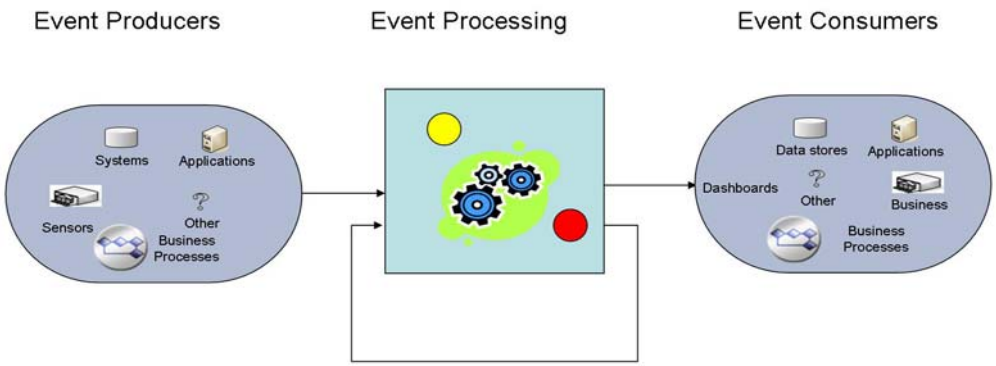


Figure 1.3: The major architectural components of event-driven computing

Somewhere in the application there will be one or more components that generate the Events. We refer to these as *Event Producers* and they are shown on the left hand side of this picture. Producers can come in a wide range of shapes and sizes, for example they might be hardware sensors that produce events when they detect certain physical occurrences, they might be bits of software instrumentation that produce events when certain error conditions are detected, or they might be explicit bits of application programming logic. Event producers are discussed further in Chapter 4.

The counterparts of the Event Producers are called *Event Consumers*. These are the components that ultimately receive the events and generally do something with them. Again they vary a lot in what they do – they might for example store events for later use, display them in a User Interface, or take action as a result of receiving them. Event consumers are discussed further in Chapter 5.

Event Producers and Event Consumers are linked by some kind of event distribution mechanism, illustrated by the arrows in the figure, and there is frequently some additional Event Processing between the Event Producers and the Event Consumers. Event distribution is frequently performed using some kind of asynchronous message passing technology, so we will often talk about the Producers "sending" events, and the Consumers "receiving" them. However other mechanisms can be used, for example the Event Producer might simply write its events to an event log file, which is subsequently read by consumers. The distribution mechanism is usually one-to-many, so that an event, once sent, can be received by multiple Event Consumers.

We'll be talking a lot more about the intermediary Event Processing as this book progresses. In simple cases this processing may just involve routing and/or filtering of the events sent by the Event Producers, however a lot of the richness in event-driven architectures comes from the fact that the Event Processing can generate additional events (in other words it can act as an Event Producer itself). These events can then be distributed to the Event Consumers, but they can also be subjected to further Event Processing (the Event Processing components can also act as Event Consumers). Our figure shows Event Processing as a monolithic component. In practice Event Processing systems often allow this processing to be specified as a sequence of sub-components which we will refer to as *Event Processing Agents*.

One important point to note in all this is the de-coupling of the Event Consumers and the Event Producers. It is the event that takes center stage here. The Event Producer has a relationship with each event that it produces, rather than a direct relationship with the Event Consumers. It is unaware of how many Event Consumers there are, and has no idea of action (if anything) the Consumer is going to take when it receives the event. Likewise the Event Consumer reacts to the event itself rather than the specific Event Producer (although in some cases the identity of the Event Producer forms part of the data that makes up the event).

We will summarize this section with some formal definitions of the new terms that we have been using:

Definition

An *Event Producer* is an entity that emits events.

This definition might look a little strange at first – you might imagine that it would also have to detect the occurrence that gave rise to the event, create the object that is going to represent it, and populate that object with data. We have deliberately chosen the definition to be as general as possible because although an Event Producer may do all those things it doesn't necessarily do so (it might be some kind of proxy relaying events in from somewhere else). More importantly, the rest of the system (the distribution mechanism, Event Processing and Event Consumers) can't actually tell how an Event Producer gets to generate its events, what's important to them is that it emits them.

Having seen the definition of Event Producer, the definition of Event Consumer should hold no surprises:

Definition

An *Event Consumer* is an entity that receives events.

We introduced Event Processing in the previous section – here is a more formal definition:

Definition

Event Processing is computing that performs operations on events. Common Event Processing operations include reading, creating, transforming and deleting events.

As we just noted, the intermediary Event Processing in our figure can be made up of a number of Event Processing Agents.

Definition

An *Event Processing Agent* is a software module that processes events.

We also distinguish between two types of event: raw events and derived events

Definition

A *Raw Event* is an event that is introduced into an event processing network by an event producer

The definition of raw event relates only to its source and not to its structure, a raw event may or may not be composed of other events.

Definition

A Derived Event is an event that is generated as a result of event processing that takes place inside the Event Processing Network

Note that this definition is relative to the system being considered. An event object can be generated in one event processing network (and therefore be a derived event in that network) and can then passed to a second event processing network where it would be viewed as a raw event.

In the sections that follow we will look more closely at the concept of events and event distribution, we will look at the ways in which intermediary Event Processing can be described and the kinds of processing that it can perform.

1.2.2 Events and event distribution

In the previous section we noted two important things about events. The first of these was that events represent occurrences in some system, and the second one is that they serve to decouple Event Consumers from Event Producers.

In some systems, it is possible for an event to represent the underlying occurrence completely, for example consider a temperature sensor that sends an event every minute. In this case the event could contain all there is to know: the location of the sensor, the time of day and the temperature value. However in many cases an occurrence may have several observable features (and even some that aren't directly observable) and an event can represent this occurrence without necessarily containing all this information. As an example here, a medical monitoring device might be able to perform multiple measurements, but only reports those that are relevant to a particular patient's treatment.

Furthermore it's possible that a single occurrence could be represented by multiple event instances. This could be the case if there are multiple Event Consumers and they happen to be interested in different aspects of the occurrence. Consider, for example, a "new employee hired" occurrence. A payroll application would be interested in the employee's name, serial number, job level and starting salary, whereas a physical security application would require the employee's office location and information on his or her job responsibilities.

There's an important use of events to represent changes of state, and this is used in many monitoring-style applications. The system being modeled is represented as a set of "resources" (these could be physical things like sensors or logical things like business processes) each of which is associated with some state information. There's usually some way to query each resource directly in order to read its state (for example the current temperature in the case of the temperature sensor) but the resources also act as Event Producers and send events whenever one or more of their internal state values change. This allows monitoring applications to be notified immediately when something happens, without them having to continually poll all the resources.

The idea of using the event itself to decouple the Event Producer and Event Consumer is a significant difference between event-based programming and the request-response invocation pattern, found in function or procedure calls. There are several reasons to consider using event decoupling:

- It may have a more natural fit to the real-word scenario that is being modeled by the application architecture.
- It supports one-many and many-one message exchanges, in addition to the one-one exchange found in the request-response pattern
- It allows further intermediary Event Processing to be added in a straightforward fashion
- It allows event processing to be preformed asynchronously to event arrival, and so is well suited to applications where events happen in an irregular manner. If there is a sudden spike of event activity then it may be possible to defer some event processing

to a subsequent, quieter period of time.

There's one area of possible confusion that it is worth clearing up at this stage, and that is the distinction between an event and a message. As we mentioned in the previous section, message passing systems are often used to pass events between Event Producers and Event Consumers, and people sometimes equate events and messages. Indeed we ourselves use the phrase "send the event". This is just convenient shorthand for "send a message containing the event", since what actually gets sent is a message containing some "serialized" form of the event. However there are other ways in which messages can relate to events:

- We have already noted that events can be distributed by other means, for example by being written to a log or database, but you might still want to send a message so that the consumer knows to retrieve the event itself. In this case the message won't contain an event at all, but it might contain a reference to that event so that it can be retrieved.
- It's also possible for a single message to carry several events – this can be convenient when there are many events to be transferred, for example when retrieving events from an event log.

1.2.3 Event processing networks

In an earlier section we remarked that the Event Processing shown in Figure 1.3 is typically not monolithic, but instead is composed of a number of Event Processing Agents. Agents are specified in an Event Processing language, and there are a number of styles of Event Processing Language currently in use. These include:

- Rules-based languages
- Script-based languages
- SQL extensions
- Other relational algebra languages

We will be looking at these in more detail later on, but for now, there's another term for us to define.

Definition

An *Event Processing Network* is a collection of event processing agents, producers, consumers and global states, connected by a collection of channels.

The event processing network is depicted in figure 1.4.

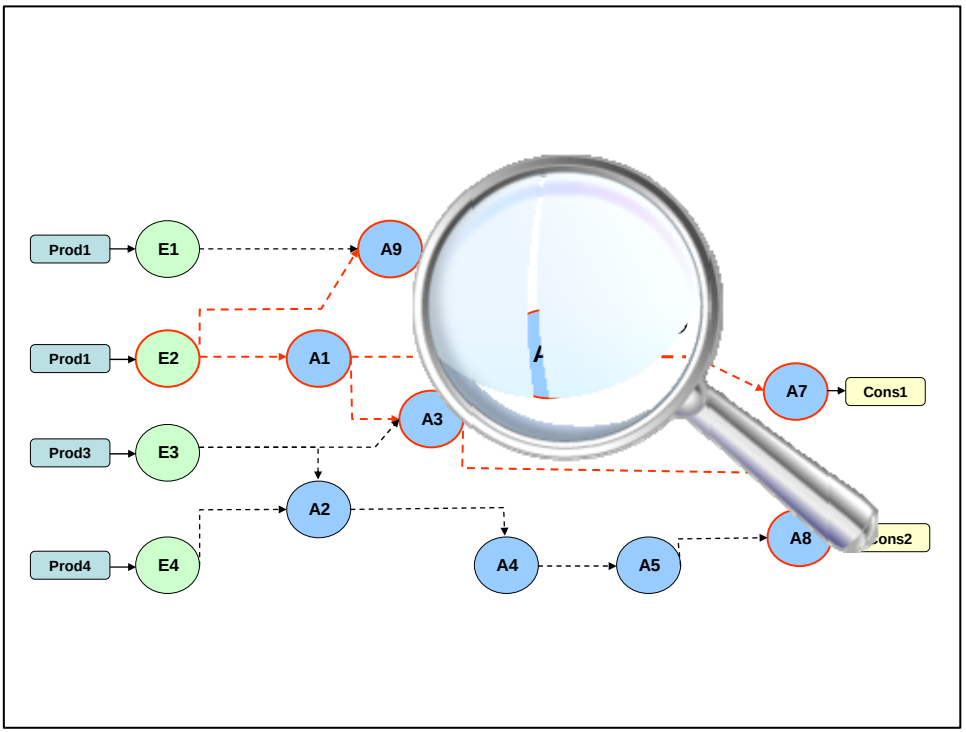


Figure 1.4: An illustration of a specific instance of event processing network

Here we see a collection of Event Processing Agents (labeled A1 through A9) linked together to form an Event Processing Network. The wiring lines show the flow of events between the various agents, referred to as *Channels*. Each agent accepts a stream of one or more input events delivered to it through a channel, and it in turn emits further events which are carried through one or more channels to further agents or to Event Consumers.

1.2.4 Types of intermediary event processing

As we saw earlier, one of the powerful ideas in event-driven computing is the abstraction (separating out) of Event Processing logic so that it sits in between Event Producers and Event Consumers rather than being hardcoded into them, and introduced the idea of Event Processing Agents. In this section we will provide brief introductions to the types of intermediary processing that are most commonly performed by such agents.

The simplest form of such processing simply takes events from the Event Producers and distributes them to the Event Consumers. This often involves some kind of filtering of the events, since not every event will be of interest to every Event Consumer (and in some cases there may be sensitive events that a particular consumer is not authorized to receive).

Filtering and routing of this sort is sometimes performed by the same infrastructure that is providing the Event Distribution, particularly if that infrastructure has *publish/subscribe* capabilities.

Alongside simple basic filtering and routing, the intermediary processing might include logging events for audit purposes in an event store.

Moving up in sophistication, the intermediary processing might involve *translating* the events – either to change their representation or to add information to them (*enrichment*) or remove information from them (*projection*). The techniques used here are very similar to those used in Enterprise Integration and can be implemented using the same tools.

1.2.5. Streams and stateful event processing

The processing that we have described so far has all been of the form one-event in/one-event out, however it is also possible to have event processing agents that take collections of events as their input or that create one ore more new events as their output:

- An incoming event may be *split* into multiple events each containing a subset of the information from the original event.
- A stream of multiple incoming events may be *aggregated* to produce a sequence of new events.
- Multiple streams of incoming events may be *composed* to produce one or more streams of output events

You will see that we have introduced a new term here – *stream* – so let's define this now.

Definition

An event stream is a set of associated events. It is often a temporally totally ordered set (that is to say that there is a well-defined timestamp-based order to the events in the stream). A stream in which all the events must be of the same type is called a *homogeneous* event stream; a stream in which the events may be of different types is referred to as a heterogeneous event stream.

Streams can be a convenient way to think of and model the processing performed by an event processing application, and some event processing systems make the stream their major abstraction. It can be more natural to think of an event processing agent as operating on an entire stream of events, rather than as operating on each event one by one. The stream concept can be particularly useful in applications that are concerned with time series events, such as periodic reading of a sensor or periodic quotes of a stock price.

Agents such as `filter` or `transform` can be described as being stateless. They simply process each incoming event and the events that they emit, if they emit any events at all, are each derived from a single input event (in the case of the *filter* agent this is the original

unchanged input event). In contrast, agents that aggregate or compose event streams exhibit something we call stateful event processing.

Definition

Stateful event processing agent; an event processing agent is said to be *stateful* if it can generate derived events whose content is influenced by more than one input event

In addition, a stateful agent does not necessarily emit a derived event every time it receives an input event.

Let's look at a simple example to illustrate the idea of a stateful agent. Suppose we have an application that receives an event every time a quantity of a given product is sold, but we are interested in knowing the total quantity that has been sold. We could use an aggregate agent to compute a running total. Each time it receives a `sale` event, the agent emits a new event containing the updated total. This meets our definition of a stateful agent, since each of the events emitted by the agent depend on all the events that is has previously received. If the volume of incoming `sale` events is high, we might decide that we don't want an `updated total` event to be emitted every time a new `sale` event comes in, so we arrange for the agent to emit derived events less frequently, for example every tenth time it sees an input event, or at certain times of the day.

In this running total example, the total starts at zero when we start up the agent and carries on increasing during the lifetime of its associated input stream, so each `total` event that is emitted is influenced by all the `sale` events that have been received up to that point. This might be what is wanted in this particular example, but it's easy to come up with cases where we only want the agent to consider a subset of the input events. Suppose that, instead of computing a running total, our aggregate agent computes an average sale size. We might then be interested in seeing a rolling average of just the last ten or last hundred `sale` events so as to be able to spot emerging trends.

To allow for requirements like this, stateful agents can be defined so that they operate only on subsets of the input events that they receive. In stream processing terminology, such a subset is often referred to as a *window* into the stream. We will return to the idea of windows and their generalization *event context* in chapter 7.

We will conclude this section with a comment on the way stateful processing is handled in implementation languages. We have already noted that there's a variety of different event processing languages, and they have different ways of handling state. There are two approaches that are commonly used:

- Some languages take an event-driven approach where the function performed by the agent is defined as an operation on a single event (just like in the stateless case) but this operation can also read or write items of state data that are associated with the agent. This state data is then carried forward from an operation performed on one

event to the operation performed on the next event that is received by the agent. Our running total example could be implemented in this manner. To do so the agent would retain a piece of state data representing the current value of the total. This would then be updated each time a new event is received, and then used when the output event is generated.

- Some languages, particular those that emphasize the stream abstraction, have an explicit window concept. In these languages incoming events are gathered into a window and the language then defines an operation to be performed upon all the events in that window. This is a convenient approach to use for our rolling average example. To implement this you could set up a *sliding window.* Each time a new event arrives the oldest event is evicted from the window and the new event is added. The average can then be recomputed using all the events that are in the window at that time.

As the running total and rolling average examples show, the event-driven approach is more natural in some cases, and the window-driven approach in others, and so some languages offer a mixture of both approaches. When we return to this topic in Chapter 9 we will show patterns using both approaches side by side.

1.2.6 Event Processing and its relationship to the real world

There is a common theme to the examples that we listed in section 1.1.2, they all use event processing to detect or report on situations, events that occur in the real world that may require some reaction, human or automated. In this section we use a couple of further examples to explore the relationship between events in the real world and their representation in an event processing system. These examples illustrate two kinds of relationship:

- Deterministic: there is an exact mapping between a situation in the real world and its representation in the event processing system;
- Approximate: The event processing system provides an approximation to real world events.

Our first example is a system that detects violations of toll payment on a toll bridge. In this case the situation we are interested in occurs when a vehicle crosses a highway toll booth without paying.

Figure 1.5.A toll booth with multiple lanes

- This can happen in two cases:

1. A vehicle uses an automatic payment system lane without having the device that identifies the car;
2. A vehicle uses a manual lane and somehow manages to sneak through without paying.

In both cases the system detects the situation and sends a picture with the vehicle license plate to the officer on duty in the other side of the bridge. From the event processing perspective we can devise a simple EPN that detects the fact that a vehicle did not pay and sends this observation along with the picture to a consumer (the officer). This is a deterministic example; the derived event that flows to the consumer implies that the situation has occurred. Conversely when the officer does not receive an event, he or she can assume that no violation has occurred.

Our second example is one in which events in the event processing system only approximate to the real world. The setting is a service provider's helpdesk; the situation that concerns us is on where a customer gets so frustrated with the service that he or she is in danger of deserting. The service provider wishes to detect this so that it can assign a skilled customer relations officer to call the customer.

Figure 1.6 A frustrated customer

In this example things are not so straightforward. A human agent can detect frustrated customers by the tone of their voice (or electronic message) but this might not catch all frustrated customers, so in addition we might use an event processing system to look at patterns of user activity, for example one that detects when a customer contacts the helpdesk three or more times within a single day. While we can construct an EPN that detects this pattern and sends a notification to the CRM officer, the detection of this pattern is neither a necessary nor a sufficient indication that the situation has occurred. In this case the derived event generated by the system is an approximation to the situation, as there may be false positives and false negatives. Dealing with uncertainty in event processing will be briefly discussed in part III of this book.

These are the basic concepts that will accompany us throughout this book. They include the basic architectural concepts as well some of the major types of event processing. Next we move to discuss event processing software and its business value.

1.3. The business value of event processing software

In Figure 1.2 we have illustrated some of the classes of applications that are event-driven applications. At the end of this chapter we list some sources that further explain the type of applications in which event processing is employed, and the business value of using these applications. One question that is frequently asked is whether there is a business value in employing dedicated event processing software (sometimes called: event processing platforms) to construct such applications, or is it sufficient to understand these concepts and just apply them using regular programming languages and existing software tools. From

observing the current market, and reading the relevant analysts' reports one can get two observations about the current state of using event processing software:

- The use of event processing software is growing rapidly; analysts claim that this is the fastest growing segment of enterprise application middleware.
- The use of event processing software covers only a small fraction of the potential.

In this Section we briefly explain some of the criteria of when event processing software might have business value to the customer.

1.3.1 Effectiveness issues

The use of event processing software may substantially reduce the total cost of ownership of event-driven applications. In a similar fashion to the use of Database Management Systems over using file systems, the benefit is in the abstraction level. Event processing software typically provides higher level abstractions relative to what programming languages provide for handling events. This may decrease the cost of development and maintenance, and thus the total cost of ownership. In some cases the higher level abstractions can also enable semi-technical persons to author event processing rules. We return to this point in Chapter 12, when talking about the event processing of tomorrow.

Another effectiveness issue is business agility; in systems where the event processing functionality change relatively quickly, it is much easier to make quick changes for functions that are expressed using higher level programming and are detached from the regular applications code.

1.3.2 Efficiency issues

In some cases the processing of events requires to scale up and meet high performance requirements. We discuss this issue more in chapter 11, when talking about implementation issues. In these cases, the use of software optimized for this purpose might be crucial to achieve this goal, as it may not be easily achievable using regular programming. This is true for other non functional properties as well.

1.3.3 When event processing software should not be used?

As in any "build vs. buy" decision, there are still cases that using event processing software may not be cost-effective. In some cases the event processing functionality required is quite simple, no special performance requirements, and the usage of event processing within the enterprise is limited, it might not be cost-effective to purchase, learn and assimilate event processing software. It may also be more reasonable to use self-built solution in the case that the required functionality is unique and is not expressible naturally within the languages provided by event processing software.

- The reader interested to read more on the business aspects of event processing is

referred to some sources at the end of this chapter.

Next we move to the Fast Flower Delivery example that accompanies this book and is demonstrated on the book's related website.

1.4. Fast Flower Delivery: an example that accompanies this book

We have many more concepts to explain, definitions to define, and details to provide but before doing that it's time to introduce a comprehensive example. This example is used throughout the book to explain and demonstrate the concepts and facilities of event driven computing. This example is demonstrated through the book's website:

http://www.ep-ts.com/content/view/74/108/

The website provides you an opportunity to see how this example is being implemented in practice, by the ability to experiment in various implementations, and see how these concepts are applied in practice.

In this section we provide a specification of this example - the "Fast Flower Delivery" application - in a detailed, yet informal, type of specification. As we proceed through the book we will examine each aspect of this application in some detail. You will find a more formal specification of the application, expressed using our building block notation, in appendix B. Figure 1.5 summarizes the main event flows in the application:

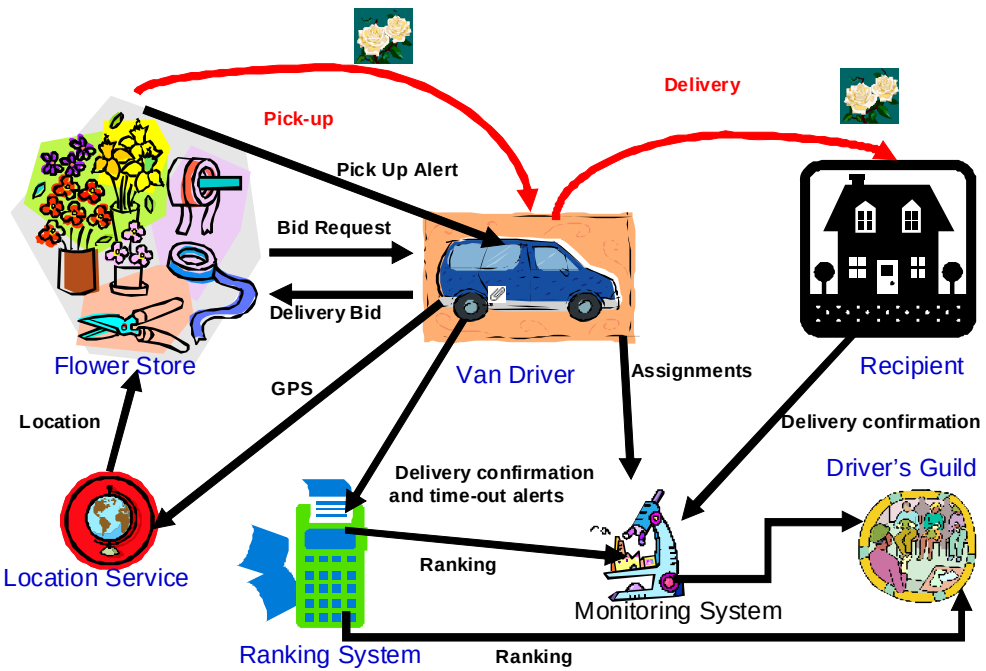


Figure 1.7 An illustration showing the various parts of the "Fast Flower Delivery" example

In Figure 1.7 the black arrows represent event flows, the pictures represent the various entities, labeled in blue, and the red curved arrows represent an actual driver's journey from a flower store to a recipient. Next we describe the example.

1.4.1 General description

The flower stores association in a large city has established an agreement with local independent van drivers to deliver flowers from the city's flower stores to their destinations. When a store gets a flower delivery order it creates a request which is broadcast to relevant drivers within a certain distance from the store, with the time for pick up (typically now) and the required delivery time if it is an urgent delivery. A driver is then assigned and the customer is notified that a delivery has been scheduled. The driver picks up the delivery and delivers it, and the person receiving the flowers confirms the delivery time by signing for it on the driver's mobile device. The system maintains a ranking of each individual driver based on his or her ability to deliver flowers on time. Each store has a profile that can include a constraint on the ranking of its drivers, for example a store can require its drivers

to have a ranking greater than 10. The profile also indicates whether the store wants the system to assign drivers automatically, or whether it wants to receive several applications and then make its own choice.

1.4.2 Skeleton Specification

Let's go through the various phases of the skeleton specification.

PHASE 1: BID PHASE

The communication between the store and the person who makes the order is outside the scope of the system, so as far as we are concerned a delivery's life-cycle starts when a store places a `Delivery Request` event into the system. The system *enriches* the `Delivery Request` event by adding to it the minimum ranking that the store is prepared to accept (each store has different level of tolerance for service quality). Each van is equipped with a GPS modem which periodically transmits a `GPS Location` event. The system *translates* these events, which contain raw latitude and longitude values, into events which indicate which region of the city the driver is currently in. When it receives a `Delivery Request` event the system matches it to its list of drivers. A filter is applied to this list to select only those authorized drivers who satisfy the ranking requirements and who are currently in nearby regions. A `Bid Request` event is then broadcast to all drivers that pass this filter.

PHASE 2: ASSIGNMENT PHASE

A driver responds to the `Bid Request`[4] by sending a `Delivery Bid` event designating his or her current location and committing to a pick up time. Two minutes after the broadcast the system starts the assignment process. This is either an automatic or a manual process, depending on the store's preference. If the process is manual the system collects the `Delivery Bid` events that match the original `Bid Request` and sends the five highest-ranked of these to the store. If the process is manual, the store makes the assignment and creates an `Assignment` event that is sent to the system; if the process is automatic then the first bidder among the selected drivers wins the bid, and the `Assignment` event is created by the processing system. The pickup time and delivery time are set and the `Assignment` is sent to the driver.

There are also some alerts associated with this process: If there are no bidders an alert is sent both to the store and to the system manager; if the store has not performed its manual assignment within one minute of receiving its `Delivery Bid` events then both the store and system manager receive an alert.

---

[4] Note that the term "Request" here means a message that requests drivers to bid; it should not be confused with a service request issued in the "request-response" protocol.

PHASE 3: DELIVERY PROCESS

When the driver arrives to pick up the flowers from the store, the store sends a `Pick up Confirmation` event; when the driver delivers the flowers, the person receiving them confirms by signing the driver's mobile device, and this generates a `Delivery Confirmation` event. Both `Pick-Up Confirmation` and `Delivery Confirmation` events have time-stamps associated with them, and this allows the system to generate alert events. A `Pick-Up Alert` is generated if a `Pick-Up Confirmation` was not reported within five minutes of the committed pick up time. A `Delivery Alert` is generated if a `Delivery Confirmation` was not reported within ten minutes of the committed delivery time.

PHASE 4: RANKING EVALUATION

The system performs an evaluation of each driver's ranking every time that that driver completes 20 deliveries. If the driver did not have any `Delivery Alerts` during that period then the system generates a `Ranking Increase` event indicating that the driver's ranking has increased by one point. Conversely if the driver has had more than five delivery alerts during that time then the system generates a `Ranking Decrease` to reduce the ranking by one point. If the system generates a `Ranking Increase` ***for a driver whose previous evaluation*** had been a ***Ranking*** `Decrease` then it generates an `Improvement Note`.

PHASE 5: ACTIVITY MONITORING

The system aggregates assignment and other events and counts the number of assignments per day for each driver for each day on which the driver has been active. Once a month the system creates reports on drivers' performance, assessing the drivers according to the following criteria:

- A *permanent weak driver* is a driver with fewer than five ***assignments*** on all the days on which the driver has been active.
- An *idle driver* is a driver with at least one day of activity which had no ***assignments.***
- A *consistent weak driver* is a driver, whose ***daily assignments*** are at least two standard deviations lower than the average assignment per driver on each day in question.
- A *consistent strong driver* is a driver, whose daily ***assignments*** are at least two standard deviations higher than the average assignment per driver on each day in question.
- An *improving driver* is a driver whose ***assignments*** increase or stay the same day by day.

As we have said, this use case accompanies us throughout the book, and provides us with a good view into the various functions performed by an event processing system. We'll cover exactly how to tackle this use case on the book's website in the next section.

1.5 How can you utilize the book's website?

The book's website is a valuable resource for the reader, since it bridges the concepts explained in this book with some of the products and open source offerings in the event processing domain.

- First we remind you the website's URL: http://www.ep-ts.com/content/view/74/108/

The Website contains information in two different categories:

- Languages-based: Each of the participating languages has links to a site from which you can download the language, documentation, and the language's implementation to the `Fast Flower Delivery` example.
- Topic-based: For each of the different topics discussed in this book, a short description of the appropriate book's chapter, and some code samples from all the languages to get a topic based view of the examples.

In this section we briefly explain both parts of the website.

1.5.1 The languages-based part of the website

In order to experience in event processing programming you need to choose the language that you would like to use. We are providing a glimpse into each of the six languages, you'll have to read the book further in order to understand the content of these code samples, but it serves as a first glance. You may chose any of these languages and go deep, you can even chose multiple of them and get better view into various ways to do event processing. In the website there are six languages detailed below.

ALERI

Aleri is represented with the CCL language, originally from Coral8, which was acquired by Aleri. The language is SQL extension, and belongs to the stream based language. The following example shows CCL query, this looks like SQL, but has some additional clauses such as: STREAM and KEEP.

```
CREATE STREAM Vwap_s SCHEMA (Symbol STRING, Vwap FLOAT);
INSERT INTO Vwap_s
  SELECT Symbol, sum(Qty * Price)/sum(Qty)
  FROM Trades_s KEEP 30 MINUTES
  GROUP BY Symbol;
```

APAMA

Apama is a division of Progress Software. Its programming style is imperative. A snippet of Apama code is shown below, as you can see, this looks like an imperative programming language.

```
455     action watchForPickUp(DeliveryRequest dr, string driver, float committedPickUpTime) {
456         // 5 minutes after the committed pick up time
457         on wait(committedPickUpTime-currentTime+PICKUP_TIMEOUT)
458         and not PickUpConfirmation(requestId=dr.requestId) {
459             //send out PickUpAlert
460             route PickUpAlert(integer.getUnique(), dr.requestId, dr.store, driver);
461         }
462     }
```

ESPER

ESPER is an open source. Its language is based on SQL and has some extensions, such as: on pattern, and it is embeddable component in Java application.

```
on pattern[every b=BidRequest(storeManual=true) -> timer:interval(2 min)]
insert into AssignmentManual
select d.* from DeliveryBidW d where requestId=b.requestId order by ranking
desc limit
5;
```

ETALIS

ETALIS is also an open source. Its rule language is based on Logic programming as seen in the following snippet.

```
check_manual_assignment/4
exceptionAlarm(check_manual_assignment(DeliveryRequestId,StoreId,ToCoordina
tes,DeliveryTime),Time-:(
     store_transmit_highest_five_delivery_bids(DeliveryRequestId,StoreId,ToC
oordinates,DeliveryTime,HighestFive) where
(check_manual_assignment_time(Time.((
print_trigger(check_manual_assignment/4.(
 %no_choice_alert/1
no_choice_alert(DeliveryRequestId-:(
     check_manual_assignment(DeliveryRequestId,StoreId,ToCoordinates,Deliver
yTime) fnot
store_select_delivery_bid(DeliveryRequestId,_DriverId,_PossiblePickupTime.(
print_trigger(no_choice_alert/1
```

RULECORE

Rulecore employs an Event Condition Action rule style. Listing 1.1 is an example of such rule is shown below, the rule is defined using XPath.

Listing 1.1 Example Event Condition Action rule style in Rulecore Code

```
<Rule name="CreateAutomaticAssignments" limit="?" evalMode="once"
level="2">
  <Description>This is rule CreateAutomaticAssignments</Description>
   <Initialize>
   <Assert>
     <Event>
     <base:XPath>sim:EventDef[@eventType="BidRequest"]</base:XPath>
     </Event>
     <Expression>
      <Property name=”Store”>
      <InList name=”AutomaticAssignmentStore”/>
       </Property>
     </Expression>
   </Assert>
</Initialize>
   <Views>
     <ViewRef name="CreateAutomaticAssignments">
      <base:XPath>sim:ViewDef[@name=”CreateAutomaticAssignments”]
      </base:XPath>
    </ViewRef>
  </Views>
<Situations>
   <SituationRef name="CreateAutomaticAssignments">
     <base:XPath>sim:SituationDef[@name="CreateAutomaticAssignments"]
        </base:XPath>
  </SituationRef>
</Situations>
<Actions>
     <SituationDetected situationName="CreateAutomaticAssignments">
        <ActionRef name="CreateAutomaticAssignments"
                   eventVisibility="external">
           <base:XPath>sim:ActionDef[@name="CreateAutomaticAssignments"]
          </base:XPath>
        </ActionRef>
    </SituationDetected>
  </Actions>
</Rule>
```

STREAMBASE

Streambase is using graphical studio to define the application. Here is a screenshot using Streambase studio. On the bottom you can see the control flow, and on the top, a definition of a particular node in this flow.

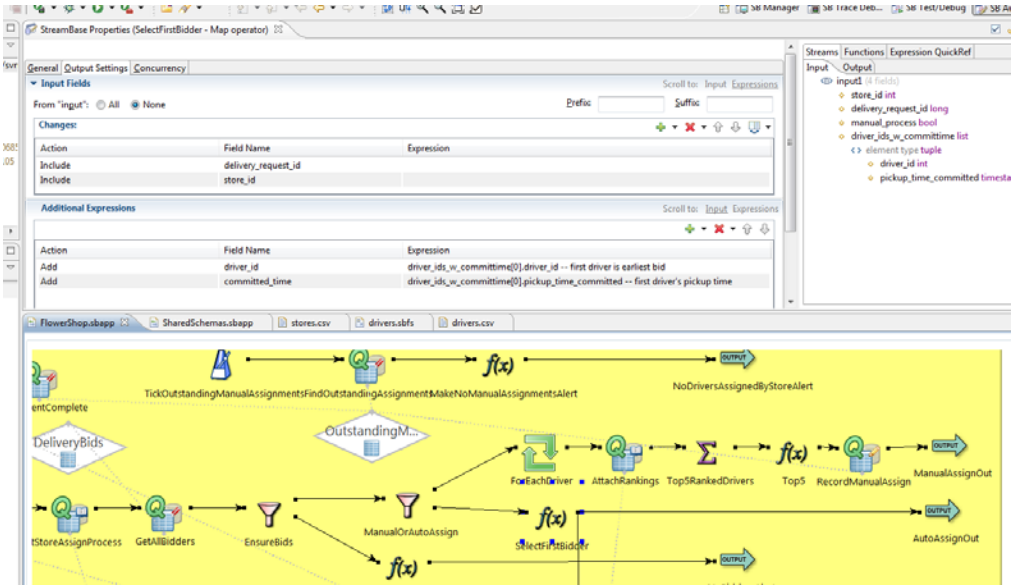


Figure 1.8: The graphical interface for Streambase.

After selecting a language, you can drill down to it using the links on the website. In order to get vertical view of the different functions you can use the topic-based part of the website.

1.5.2 The topic-based part of the website

The topic-based part of the website follows the various topics discussed in the book.

- The building block topic described in chapter 2, is the modeling meta-language that is described throughout this book. In the website, there is a link to an editor for this meta-language, as well as examples of its output.
- The event type topic described in chapter 3, including some code examples from various products.
- The Event Processing Network topic described in chapter 6, including some graphical examples from various products.
- The context topic described in chapter 7, including some code examples from various products
- The transformation and filtering topics described in chapter 8, including some code examples from various products

- The pattern matching topic described in chapter 9, including some code example from various products.

Note to the reviewers and MEAP subscribers: the website is still under construction.

1.6 Summary

We process events all the time in our daily lives; however traditional software paradigms have not been oriented towards event-driven functionality, but instead have focused on more synchronous request-driven interactions. Imagine that you have an employee that sits idle and works only when explicitly told exactly what to do - that does not sound like a very effective way to handle a business. Likewise there are cases in which software is more effective if, rather than waiting to be told what to do; it can detect when an event has happened and can decide whether and how to react.

In this chapter we have discussed the following topics:

- The concept of event and event-driven computing;
- The motivations behind event-driven computing using some examples,
- Some of the basic concepts behind event driven computing,
- The "Fast Flower Delivery" use case the accompanies this book
- A glance into the book's website.

By now you should be familiar with the basic concepts at a high level. Reading the rest of the book will provide you with much more detail about these concepts, and give you a deep understanding of their proper use in constructing event-driven applications.

Additional reading

K M Chandy, W R Schulte: Event Processing: Designing IT Systems for Agile Companies McGraw-Hill Osborne Media; 1 edition (September 24, 2009)
http://www.amazon.com/Event-Processing-Designing-Systems-Companies/dp/0071633502/ref=sr_1_1?ie=UTF8&s=books&qid=1258816511&sr=8-1

This book is an excellent reference book for the business motivation for event processing, as well as the positioning of event processing vs. SOA and other related concepts. It also provides some terminology discussion, e.g. various interpretations of the term "event".

David Luckham: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Professional May 2002.

http://www.amazon.com/Power-Events-Introduction-Processing-Distributed/dp/0201727897/ref=sr_1_2?ie=UTF8&s=books&qid=1258816511&sr=8-2

The first event processing book that introduced early versions of many of the concepts discussed in this book, such as: Event processing network and event processing agent.

Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, Jennifer Widom: STREAM: The Stanford Stream Data Manager. SIGMOD Conference 2003: 665
http://www.sigmod.org/sigmod03/eproceedings/papers/dem09.pdf

This article describes the STREAM project in Stanford, which introduced the term data stream management.

Detlef Zimmer, Rainer Unland: On the Semantics of Complex Events in Active Database Management Systems. ICDE 1999: 392-399
http://www.informatik.uni-trier.de/~ley/db/conf/icde/ZimmerU99.html
This article is a good survey of related concepts in active databases; one of the ancestors of is an early paper that discusses event derivation and patterns in the context of active databases.

Exercises

1. Provide some more examples of real-life activities which involve real-world events and reactions to them.
2. Classify the ten examples given in section 1.1.2 into the categories given in section 1.1.3. Remember than an example can fit into more than one category.
3. Provide three more examples of the use of automated Event Processing, and analyze the benefits brought by automation.
4. Provide an example of an application in which an Event Producer also serves also as an Event Consumer**.**
5. Can events be distributed using a Request/Response (Remote Producer Call) message exchange pattern? Give an example or, alternatively, explain why this is not possible.
6. List all event types used in the "Fast Flower Delivery" application.
7. List all the event processing agents in that application
8. List the event types consumed and emitted by each Event Processing Agent in the “Fast Flower Delivery” application.
9. Describe three more functions that could be added to the “Fast Flower Delivery” application.

2

Event Programming Principles

"An apprentice carpenter may want only a hammer and saw, but a master craftsman employs many precision tools. Computer programming likewise requires sophisticated tools to cope with the complexity of real applications, and only practice with these tools will build skill in their use."

- Robert L. Kruse

In the previous chapter we introduced the basic concepts and our example use case. In this we explore these concepts in more detail. In particular we look at:

- Event-driven interactions and event distribution patterns
- Event-driven applications and architecture and their relationship to Service Oriented Architecture
- We introduce the notion of building block and describe the various types of event processing building block, the event processing network concept and the graphical notation we use to depict it

The focus of this chapter is on the general principles underlying these topics. In part III we will show them being used in a worked example.

2.1 The background: request-response interactions

Everyone who has used a web browser (figure 2.1) will have had first-hand experience of the request-response interaction pattern. When you use a browser you formulate a request, usually by clicking on a link or filling in a form, which is sent across the Internet to a server. You then wait while a response is constructed by the server. This response is returned across the Internet and then displayed by the browser.

In a request-response interaction the request can be a request for information (sometimes called a query) or it can be a request for something to happen as is the case, for example, when ordering a product (this is sometimes called an update). In the query case the response returns the information asked for; in the update case it usually carries some sort of confirmation that the update has been made successfully. The response in an update case can contain some extra information, for example if the request was a request to place an order then the response might give an indication of the time when that order will be fulfilled.

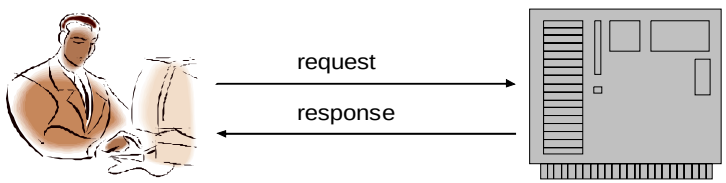


Figure 2.1. Request-response in the World Wide Web

Request-response[1] interactions are used very frequently in distributed computing, and form the basis of most Service Oriented Architectures. In the general interaction pattern one application or application component, termed a *service requestor* (or sometimes a *client*) sends a request to another, known as a *service provider* or *server,* and sometime afterwards the requestor receives a reply. Figure 2.2 shows a Sequence Diagram of such an interaction, where the requestor, an Order Processing application, sends a stock query request to an inventory system to determine whether a particular item is in stock or not.

[1] Request-response as well as some of the other terms are discussed in depth in the book: Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003 http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Deploying/dp/0321200683/ref=sr_1_1?ie=UTF8&s=books&qid=1258829949&sr=1-1

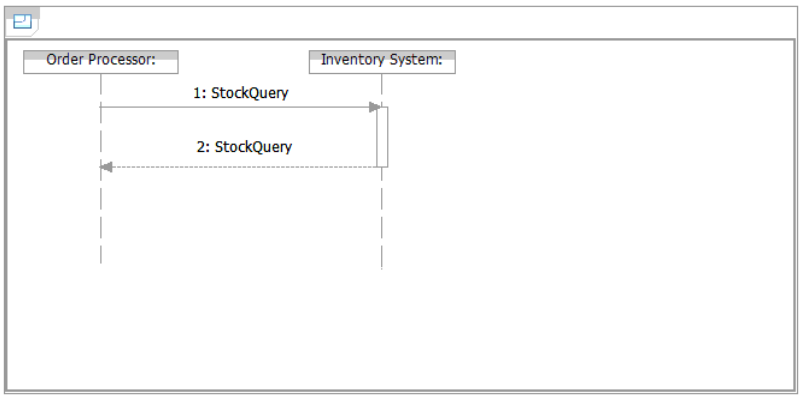


Figure 2.2 Request-response in distributed computing

The request-response interaction pattern is a convenient programming technique. It is used when interacting with all REST-style web services[2] and most SOAP-style[3] ones and it forms the basis for the Remote Procedure Call (RPC) invocation pattern used in most distributed object systems. It has the advantage of being familiar to any programmer who has made a procedure or function call in a procedural programming language or who has invoked a method in an object-oriented language. It often fits naturally into the design of an application. When writing a program you reach a point where the program needs a piece of additional information, or where it needs to make sure that something takes place that is external to it, and the natural thing to do at such a point is to invoke an external component or service to return the information or to perform the operation. As we noted earlier the arrival of the response indicates that the request has been received by the service provider and has been acted upon[4].

Although there is an asynchronous version of the request-response pattern, the great majority of request-response interactions are synchronous in nature. In a synchronous interaction the provider is expected to send a response back fairly promptly. This usually means a few tenths of a second at the longest, though response times of several seconds are sometimes encountered in web applications. As a consequence, it is reasonable for the

---

[2] Roy T. Fielding, Richard N. Taylor: Principled design of the modern Web architecture. ACM Trans. Internet Techn. 2(2): 115-150 (2002) http://portal.acm.org/citation.cfm?doid=514183.514185
[3] http://www.w3.org/TR/soap/
[4] This is only part of the story, since an application might still have to cope with the situation where a failure of some kind means that no response is received. In such circumstances the requestor cannot immediately tell whether the request was acted upon or not.

requestor program's thread of execution to wait for the response to arrive without doing anything else in the interim[5].

2.2 Event-driven interactions and the principle of decoupling

Events differ from the requests as an event producer does not, in general, expect consumers to take any specific action when they receive its events, and it does not expect to get any responses in reply to these events. One reason for this is that responses would relate to the action taken by the consumer, and as we have just seen the producer is not aware of what this going to be. Another reason is that the number of such consumers is often not known when the event producer is being designed and implemented (and in some cases the number can vary dynamically while the application is running).

This means that events are often sent as "one-way" messages from producers to consumers, in the manner shown in Figure 2.3.

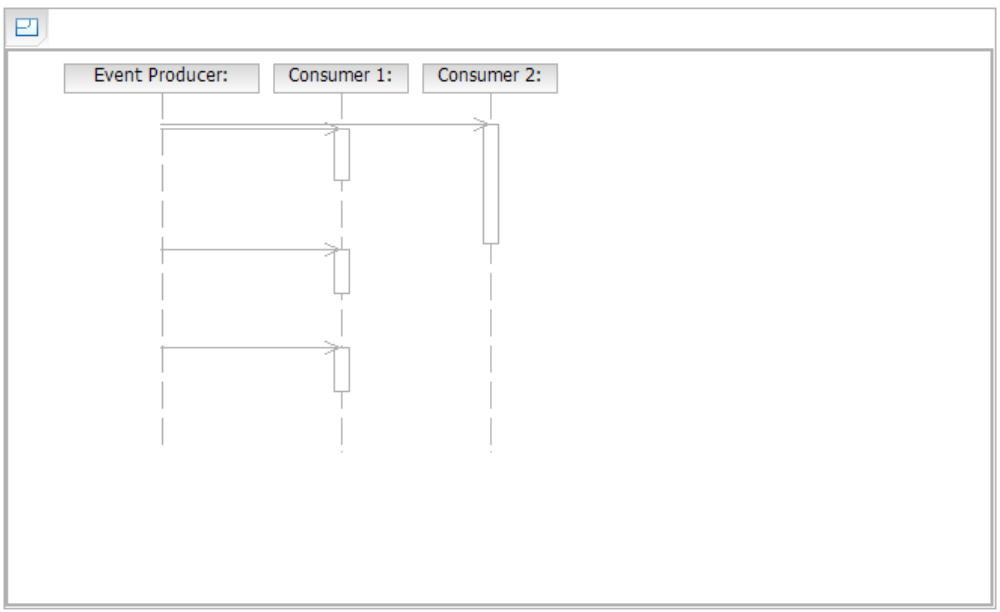


Figure 2.3 Typical Push-style Event distribution

[5] In practice there is often some kind of time-out mechanism so that the requestor thread waits only for a finite period before deciding that there must have been some kind of failure and that the response is never going to arrive.

In this diagram (a UML sequence diagram) time travels down the page, the various entities that communicate are shown as vertical "lifelines" and their interactions by horizontal arrows. You can see that, in this style of distribution, communication is initiated by the producer when it has an event to distribute. It "pushes" the event to a consumer (or consumers) as a one-way message. In the illustration you can see that the producer sends a copy of its first event to two separate consumers, but then sends two more events, this time just to consumer 1.

This push approach can reduce processing latency since the producer can send an event as soon as it has one to distribute (this is in contrast to the "pull" style which we'll look at in the next section).

Figure 2.3.illustrates three commonly-found characteristics of event distribution:

- The consumer does not send a response to the producer, other than possibly an indication that it has received the event. This means that the consumer can process the event *asynchronously* to the producer.
- A given event instance may be delivered to more than one consumer, and each consumer can process that event in a different way.
- A producer may produce a sequence (stream) of similar events over time.

To see why these characteristics come about we need to step back and look at the fundamental difference between an event and a request. An event is an indication of something that has already happened whereas a request, as it name implies, expresses the requestor's wish that something specific should happen in the future, for example that a provider will provide it with some piece of information or will perform a particular service on its behalf.

This distinction means that event producers and event consumers can be completely decoupled from each other. There is of course a degree of decoupling in a request-response interaction as when you implement a service provider you generally code it to provide that particular service taking no regard of the nature or purpose of the service requestor. However in a request-response interaction the service requestor is dependent on the service provider performing some agreed function for it – whereas in Event Processing, a producer can be decoupled from a consumer and the consumer decoupled from the producer. We can summarize this as the "Principle of Decoupling":

Principle of Decoupling

In a decoupled Event Processing System an event producer does not depend on any particular processing or course of action being taken by any event consumer. Moreover an event consumer does not depend on any processing performed by the producer other than the production of the event itself.

So in a decoupled system there can be more than one consumer of any event and the action taken, if indeed any action is taken at all, can vary significantly between consumers. It can also vary during the lifetime of the application. As an event producer does not know what an event consumer is going to do with an event, or even how many consumers there are, it usually does not make sense for the event producer to expect a response in reply to its events.

At this point we should point out that even though a producer itself does not expect a response, there are situations where a producer forms part of a larger application component that does expect to receive incoming events. Let us consider the Flower Delivery use case that was introduced in chapter 2. In that use case a flower store has an event producer that produces a `Delivery Request` event which is submitted to the delivery scheduling system. Shortly after this, if it has chosen to do manual assignment, the store will receive some `Delivery Bid` events. These `Delivery Bid` events are indirectly caused by the `Delivery Request` but we view them as events in their own right rather than direct responses to the request event. As such they are handled by an event consumer portion of the store's application rather than by the event producer responsible for the original `Delivery Request`. This distinction may look unnecessarily subtle at first, but by keeping a clear separation of producers and consumers and viewing the `Delivery Bids` as separate events we are able to build more flexible and adaptable applications.

One other way of viewing the *principle of decoupling* is that an event can, and usually does, have meaning outside the context of any particular interaction between its producer and consumers. In contrast a request in a request-response interaction generally has meaning only within that interaction.

2.3 Further event distribution patterns

In the simple push configuration shown in Figure 2.3 each event is sent directly from the producer to its consumers, and in some cases this means that the producer has to send multiple events.. One way to achieve this is to make the event producer responsible for this distribution, so that the event producer has knowledge of the relevant consumers. This knowledge can be configured statically into the event producer, or alternatively the event producer can find out their identities from a dynamic external source.

Alternatively the event producer can delegate this distribution function to another entity, as shown in Figure 2.4.

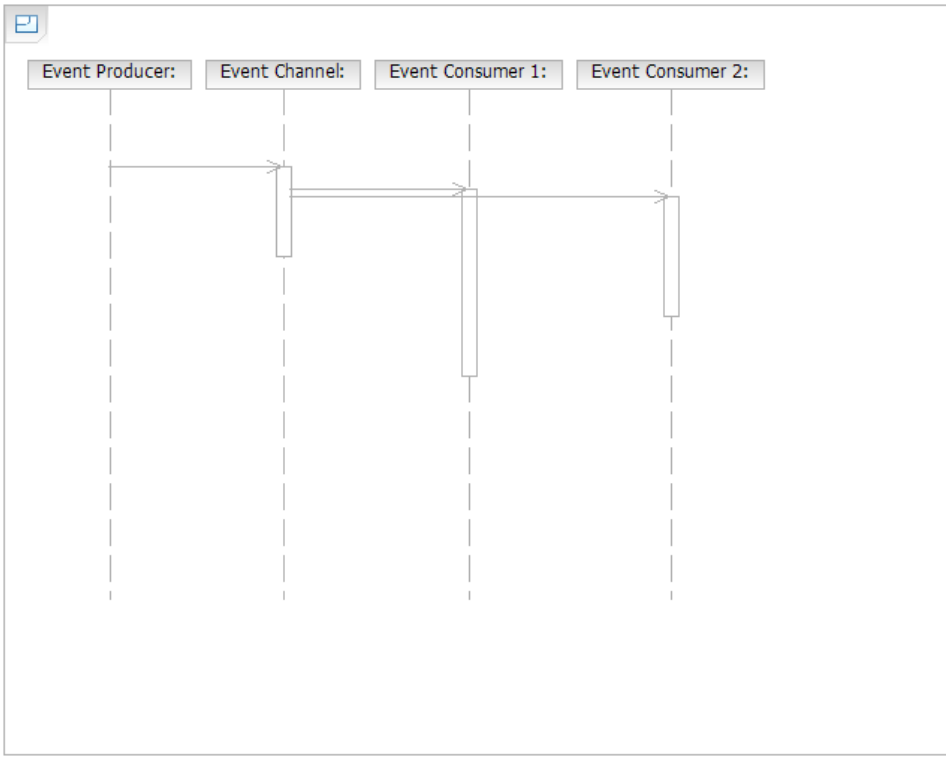


Figure 2.4 Push-based event distribution with an intermediary event channel

Here you can see that the event producer just sends a single copy of the event to the intermediary event channel, and that this channel then takes care of forwarding it on the event consumers.

In practice an event channel can be implemented in a number of different ways:

- It could be an intermediary service or other piece of software (sometimes called a *broker*)
- It could be implemented using a multicast protocol, such as IP Multicast
- It could be implemented using Message Oriented Middleware (MOM), such as a Java Message Service (JMS) provider

How does the channel know which consumers it should forward the event to? That partly depends on how it is implemented. In the IP Multicast case, the event is automatically forwarded to any consumer that is listening on the particular multicast group address that the producer has used. In the broker or MOM cases, the channel could be statically

configured with the list of consumers, or the consumers could register themselves (or be registered by someone else) with the channel. If dynamic consumer registration is involved then the pattern is known as *Publish/Subscribe* and the consumer registrations are referred to as *subscriptions*.

We will now, as promised earlier, take a look at situations where the request-response interaction pattern is used along with events. In all these examples an event is passed as a kind of parameter on either the request or the response part of the interaction. Events are passed in the response part in "pull" distribution of events, illustrated in Figure 2.5.

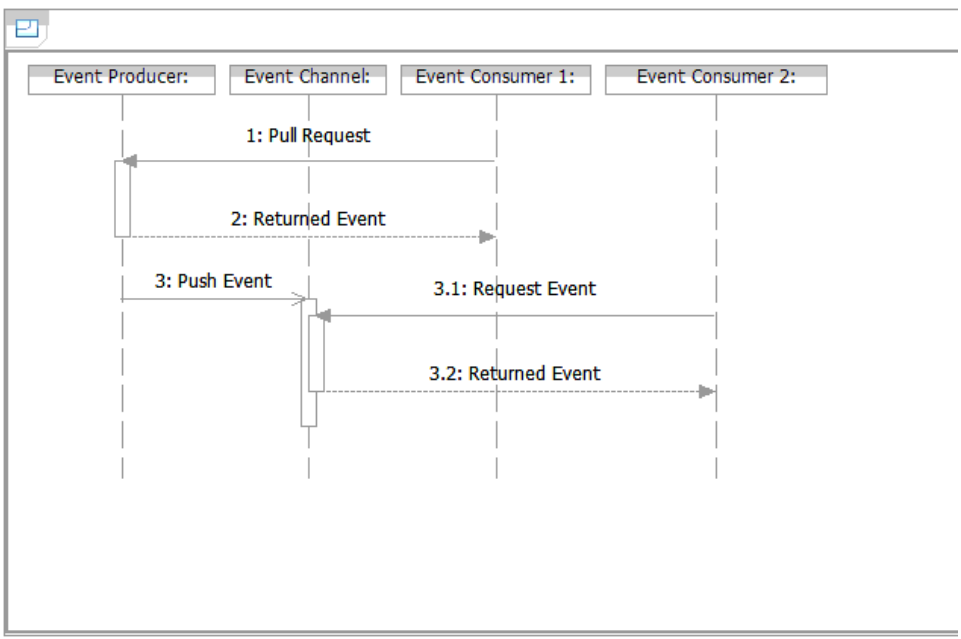


Figure 2.5 Pull-based event distribution

In pull-based distribution, the consumer uses the standard request-response pattern to request an event from a producer, or an intermediary, and receives that event in the response part of the interaction. In the examples in figure 2.5, we see Event Consumer 1 sending a pull request directly to the Event Producer (message 1) which then responds by returning an event in message 2. To avoid having to hold on to events and to avoid having to service requests from multiple consumers, an event producer may choose to delegate this job to an intermediary known as an *event channel*. Figure 2.5 also shows this: the producer uses a regular push (message 3) to send the event to the channel, and Consumer 2 then requests it (message 3.1) from the channel. Our examples only show a single event being

returned, but this approach can be extended to send multiple events in the response messages. This is useful as there could be more than one undelivered event pending at the producer or channel when the Pull request is received.

Although push distribution is widely used, there are situations where pull is used instead. These include situations where there is:

- A consumer, such as a mobile device, which is only available occasionally, or is only intermittently connected to the distribution system
- A consumer which wants to regulate its processing of events and have control over exactly when it receives them
- A consumer which is physically unable to receive unsolicited incoming events, for example a computer system behind a firewall
- A producer which is unwilling or unable to distribute events

A common example of the last of these (the producer that is unable to distribute events) occurs where there's a system that is writing events to an Event Store or Log. This is a file or database system that is used for medium or long term event storage, and can provide a pull-style producer interface, allowing other entities to query the log using request-response style queries.

A store might also provide an interface that allows events can to be written to it using a request-response pattern. This is an example of a case where the event is supplied as part of the request message.

2.4 Benefits of using the event-driven approach

One immediate question you might have is why you might want to use an event-driven approach in the first place. Here are some reasons:

- Your application might be naturally centered on events. Many of the examples we gave in section 1.1.1 are like this. They involve some kind of sensor that detects and reports events and the purpose of the application is to analyze and react to these events.
- Your application might need to identify and react to certain situations (either good or bad) as they occur. An event-driven approach, where changes in state are monitored as they happen lets an application respond in a much more timely fashion than a batch approach where the detection process runs only intermittently.
- Your application might involve analysis of a large amount of data in order to provide some output to be delivered to a human user or some other application. By treating the input data as events you can use an event-driven approach to distribute this analysis across multiple computing nodes.
- The event-driven approach can give you a way of extending an existing application in a flexible, non-invasive manner. Rather than changing the original application to add

the extra function it's sometimes possible to instrument the original application by adding event producers to it (for example by processing the log files that it produces). The additional functionality can then be implemented by processing the events generated by these event producers.

- There are potential scalability and fault tolerance benefits to be gained by using an event-driven approach. These benefits can be even greater for a system that minimizes or eliminates request-response functionality.

We are not claiming that event-driven computing is some kind of universal solution, and that existing batch, on-line transaction processing, or database-centric applications should be re-written using event processing technology. However it does have an important part to play in complementing these approaches. As we have just seen there are some applications where an event-driven approach will give you a more timely response, better throughput or greater flexibility through the separating out of event processing logic from the mainstream application code. It doesn't have to be a case of "either-or"; many applications incorporate a mixture of event-driven and other approaches.

2.5 Event processing and its connection to related concepts

Now we have covered some of the basics, we move to look at some of the interactions between EDA and Event Processing and other concepts that exist in the IT environment. We will discuss the following areas: Service Oriented Architecture (SOA), Business Process Management (BPM), Business Activity Monitoring (BAM), Business Intelligence (BI), Business Rules Management Systems (BRMS), Network and System Management (NSM), Message oriented Middleware (MOM) and Stream computing. As this is a very wide topic, we provide some references for the interested reader at the end of this chapter.

2.5.1 Event-Driven Architecture and Service Oriented Architecture

The terms "Event-Driven Architecture" (EDA) and "Service Oriented Architecture" (SOA) both end with the word "architecture", so you might imagine that they are alternatives and that you have to choose which architectural style to adopt. It is our contention that this is not the case and that it is perfectly possible to use an EDA approach within an overall SOA.

To see why we say this, we need to look at what we mean by SOA. If you were to take a narrow definition of SOA that says that every application has to be built out of request-response oriented services (and only out of request-response oriented services) then it is true that you would be limited to request-response interactions and wouldn't be able to benefit from all of the advantages of event-driven computing. However we claim that there is nothing fundamental to SOA that dictates the exclusive use of request-response.

The main idea in SOA is to move away from a monolithic way of designing applications to one where applications are composed of reusable shareable components. For this to be possible the components need to have well-defined interfaces that are independent of their

implementations and it must be possible to incorporate a given component into more than one application. The emphasis is on the flexibility that comes from this reuse, so that IT systems can be adapted quickly and easily to respond to new business conditions. Much of the focus of SOA is on the design principles behind this and the lifecycle and governance issues that come along with such an approach – for example identifying what services are available and who is responsible for developing and maintaining them.

There is nothing in the last paragraph that implies that the SOA components (services) have to be exclusively request-response oriented, although the reusability requirement does require a degree of decoupling between components. As we saw in section 2.2, event-processing components, such as Event Producers and Event Consumers, are in fact more decoupled than request-response components are, so there is no difficulty using them within an SOA.

There are two main ways in which the event-driven approach can be mixed with request-response components in an SOA:

- It is possible for a component to implement both approaches. In other words it can provide or consume a request-response interface AND also be an event producer or event consumer. As an example consider a service that fulfils orders via a request-response interface and produces an event when it detects that stock levels are low.
- The SOA infrastructure that hosts the SOA components can provide instrumentation that produces events on behalf of request-response style services. For example you could instrument the order fulfillment service just mentioned so that an event is produced each time an order is fulfilled. These events are then processed by event-processing components.

So you can view event-processing components as extending the repertoire of components used inside an SOA, and an event processing network as being a kind of SOA composite service. The lifecycle and governance aspects of SOA apply equally well to these components, although of course their interfaces are defined slightly differently - an event producer is defined in terms of the events it produces rather than the request-response interfaces that it relies on, and an event producer is defined in terms of the events it consumes rather than the request-response interfaces that it implements. The term "Event-driven SOA" is now used by some analysts and vendors to denote the combination of EDA and SOA[6].

2.5.2 Event-driven Business Process Management

Business process Management deals with computerized support of modeling, managing, orchestrating and executing some or all of an enterprise's business processes. BPM software has emerged from evolution of workflow systems, and is now frequently included in SOA

---

[6] See the Wikipedia entry: http://en.wikipedia.org/wiki/Event-driven_SOA for more discussion and some references.

platforms. There are two major standards related to BPM software: BPMN (Business Process Model and Notation)[7], an OMG standard that deals with the modeling side, and BPEL (Business Process Execution Language)[8], an OASIS standard that deals with the execution side. There are various synergies between these two areas:

- The BPM system can serve as an event producer, generating events that report on state changes within the BPM system, which are then analyzed by the event processing system, with the resulting derived events either being returned to the BPM system, or disseminated to other applications.
- The BPM system can act as an event consumer, reacting to situations detected by the event processing system after it has analyzed events from outside the BPM system. There are several ways in which an event sent from an event processing system could interact with the BPM system: the event could trigger a new instance of business process, it could affect a decision point within the flow of a business process that is already running, or it could cause an existing process instance to stop running. We will give an example from human resources management. Our fictional enterprise uses a managed business process to make management appointments. This process consists of a number of activities, advertising the position, identifying candidates, evaluating them and reaching a decision. There are several events that could trigger this process to start such as retirement or promotion of the current manager. Once the process has started it could be affected by a number of external situations, for example the event processing logic could detect that fewer candidates have applied than expected and so it could cause the process to launch an activity to encourage more applications, such as additional publicity or the use of a recruitment agency. During this process, there might be an organizational change that eliminates the position, in which case the process should be canceled, or an organizational change that alters the set of skills required for the job, which might remove some candidates from consideration.

As the time of writing, there are some BPM products that embed their own ad-hoc event processing capabilities. In the future we might see more loosely-coupled integration between BPM and event processing products. We provide some references at the end of the chapter for readers interested in pursuing this topic further.

2.5.3 Business Activity Monitoring (BAM)

Business Activity Monitoring is a term coined by Gartner[9]; BAM software typically tracks Key Performance Indicators (KPIs), for example a book publisher might want to use the number of copies sold per month of an important E-book as a KPI. BAM software is considered today

---

[7] http://www.omg.org/spec/BPMN/
[8] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
[9] http://www.gartner.com/resources/105500/105562/105562.pdf

as a category of enterprise middleware software. The BAM concept, in principle, is wider than KPI tracking and may include any observation of business related events.

BAM software systems typically contain some event processing functionality. Systems that are centered on KPI tracking perform filtering, transformation and especially aggregation of events. This can be done in batch mode, calculating KPIs at the end of each day/week/month as appropriate KPI, or in online mode so that the current value of the KPI can be continually tracked, typically on some sort of dashboard[10].

There are two trends in the evolution of BAM software that will require more event processing functionality: the drive to provide richer types of observation, which necessitates more event processing functions such as event pattern detection, and the need to provide more on-line observations where data comes from multiple event sources. . We anticipate that these trends will lead to more use of event processing software in tightly or loosely coupled integration with BAM software.

2.5.4 Business Intelligence (BI)

Business Intelligence (BI) is a collection of analytics techniques and software used to help organizations make decisions based on data that they have collected.

Today's BI systems differ from event processing systems in that they are request-driven. A typical BI system takes as its input a set of data that has previously been collected in a data warehouse. It analyzes the data retrospectively, and does not respond to events as they happen. This approach is so different that it makes sense to think of Business Intelligence and Event Processing as being separate disciplines, dealing with different problems.

However we are beginning to see the appearance of software that provides online analytics for decision making. This is a kind of event-driven BI, since it involves versions of the BI analytics that can run online, triggered by events. The term "operational intelligence"[11] is sometimes used to describe this area. While mainstream BI functionality will remain request-oriented for the foreseeable future, event-driven BI is useful to some segments of the BI marketplace, and so we expect to see this functionality being included in BI software.

2.5.5 Business Rules Management Systems (BRMS)

Business Rule Management Systems (BRMS) are software systems that execute rules, typically in the form of "condition-action" or "if-then" which are kept separate from mainline application code, so that they can be modified without requiring change to the application

---

[10] The concept of dashboard is further discussed in Chapter 5, while discussing event consumers.
[11] The Wikipedia entry for operational intelligence provides some explanation: http://en.wikipedia.org/wiki/Operational_intelligence

code itself. These rules are expressed in declarative languages and are managed by dedicated software platforms.

There is often confusion between the concepts of BRMS and event processing, mainly due to the fact that some event processing software is based on rule oriented languages, and the term "rule" is sometimes used for the function that, in our model, is represented by an EPA. BRMS and EP systems have some fundamental differences:

- Event processing is invoked by the occurrence of events; business rules are invoked by requests made from application logic.
- Business rules operate on states; event processing operates on events and can consult state.
- The essence of business rules is inference, the main functionality of event processing is event filtering, transformation and pattern matching.
- These differences result in different functionality, different execution mechanisms and different types of optimization. There have been attempts to implement business rules' functionality using event processing software, and event processing functionality using BRMS software, for example: adding temporal extensions to BRMS to express the equivalent of event processing temporal patterns[12], however neither of these usually lead to optimal implementations.

There are also some synergies between event processing and business rules:

- From event processing standpoint, business rules can be used for routing and filtering decisions made by the event processing software.
- Event processing functionality can be expressed in a rules-based programming style.
- The occurrence of events or situations can trigger a request to invoke a BRMS business rule.
- In the future, we might imagine BRMS business rule evaluations that query the internal state of an event processing pattern matching process, or that evaluate an event processing retrospectively. However this is beyond the current state-of-the-art.

These synergies, coupled with growing quantity of applications that require both event processing and business rules functionality, may serve as a motivation for tighter integration between BRMS and event processing, including common programming models and common product packaging.

2.5.6 Network and system management

The area of network and system management is event-driven. One of its major goals is to monitor error events (sometimes called *symptoms*) and analyze them to find their root

---

[12] Chapter 9 in this book explains event processing patterns in general, and temporal patterns in particular.

causes. An underlying problem may give rise to many symptoms, for example a failed network router could cause a number of services to fail. Network and system management software uses a technique called event correlation[13] to examine symptoms and identify groups of symptoms that have a common root cause. These systems also look for particular patterns among the events that they monitor.

Network and systems management products pre-date more general purpose event processing software, and these two kinds of product have evolved alongside one another. They have been focused on different kinds of application, management applications in one case and business applications in the other.[14] They also have different type of users (network and systems management users are typically system administrators) and different non-functional assumptions.

Some applications (such as Business Activity Monitoring) may require a combination of systems management and event processing applications, and this might be a motivation for more synergy between the two areas, but we expect that they will continue to move along separate tracks for at least the near future.

2.5.7 Message Oriented Middleware (MOM)

Message Oriented Middleware (MOM) complements event processing, but also partially overlaps with it. MOM provides a transport layer that event processing may employ as an infrastructure to implement event channels. There is also some overlap, the filtering and some of the transform functionality of event processing is similar to the filtering and transform functionality in MOM. There are also some differences:

- While an event object may be represented as a message, a message does not necessarily represent an event, for example if I am sending a picture as a message using MOM system, the picture does not represent event
- While messages in Message Oriented Middleware can have temporal semantics, such as timestamps, expiry and ordering, these are not hard requirements. In contrast temporal properties are fundamental to events[15].
- MOM typically handles each message separately from every other, while event processing usually includes functions that operate of a collection of events, for example aggregation and pattern detection.
- Event processing may be implemented using a combination of MOM and dedicated event processing components, using the MOM to route event messages between event processing components and to perform some filtering. You can find a good description of the MOM pattern and related concepts in Hohpe and Woolf's Enterprise Integration

---

[13] A Computerworld article described event correlation in network and system management http://www.computerworld.com/s/article/83396/Event_Correlation?taxonomyId=16&pageNumber=1

[14] For an interview with Tom Bishop, who is BMC CTO while the book is written about this topic see: http://complexevents.com/wp-content/uploads/2008/11/an-answer-to-a-question3.pdf

[15] Chapter 3 discusses the structure of events, and in particular the temporal dimensions of events.

Patterns book.[16] This book also includes several of the concepts that we cover; they provide a message oriented view, whereas we describe them from an event processing viewpoint.

2.5.8 Stream computing

The term "event stream processing" is used by some people as alias for event processing, while others use it to refer to a subset of event processing. Stream computing is much wider, encompassing streams that contain data which we would not normally view as events, for example video streams, and audio streams. Stream computing is an infrastructure based on dataflow model, which may be used to run various types of applications

Event processing has some intersection and some synergies with stream computing:

- Event stream processing may be considered as a subset of stream computing, it can be extended to full event processing functionality.
- Functions implemented on top of stream computing framework can serve as event producers, for example in a security application you might use a stream processing platform to extract events out of video streams which are then forwarded to an event processing platform for further analysis.

Additional reading on stream computing is detailed at the end of this chapter.

For the remainder of this chapter we will look more closely at the way we define event-driven components, and the way we can use *event processing networks* to assemble them into composite applications or services.

2.6 Event Processing Building Blocks

Many computing books explain concepts by providing examples in a particular programming language or using a particular product implementation. We could have chosen to use such an “implementation-up” approach; however there is quite a variety of implementations available and by picking one we would have restricted our thinking to the constructs that that implementation chose to use as primitives.

Instead we follow the model driven architecture (MDA) approach. We will be describing event processing applications as collections of platform-independent *definition elements.* Each definition element is an instance of a *building block*. A building block denotes an abstraction that is presented to an application designer and is distinct from the system-level artifacts that implement it. You will recall that in Chapter 1 we introduced event processing agents, event producers and consumers, event channels and event types, and these are all terms that are used in an application description and each has an associated building block.

---

[16] http://www.eaipatterns.com/

A building block abstraction allows us to talk about the important features of a concept without getting caught up with the details of a particular implementation. It also acknowledges the fact that there may be more than one way to implement a particular concept. For the benefit of the reader who wants to acquire hands-on experience with event processing tools, this book's website provides an opportunity to download and experiment with various implementations of these concepts. The website also includes a link to a graphical editor that lets you to experiment by modeling the building blocks. The book's website can be found at:

http://www.ep-ts.com/content/view/74/108/

The direct link to the building block editor is:
http://code.google.com/p/epdleditor/

2.6.1. What is a building block?

An Event Processing *building block* represents an Event Processing concept and is used to create platform independent definition elements, which are implementation-neutral instances of this building block. For example, we can use the `Event Type` building block to create platform independent and implementation-neutral representations of the event types needed by an application - the `Delivery Request` event type we met earlier being one such definition element. Each application is made up of a collection of these definition elements, customized to perform a particular role and connected together to form the complete application.

When the application is implemented these platform-independent definition elements have to be translated into one or more *platform-specific definition elements*. In this book we are not concerned about the representation of these platform-specific elements, which will depend on the tool that is being used to develop the application (they could for example be statements or classes in a programming language, they might be configuration files or entries in a database table). Once it has been developed, the application will be run and this will result in the creation of one or more runtime instances of these definition elements.

The relationship between building blocks, element definitions and runtime instances is shown in Figure 2.6.

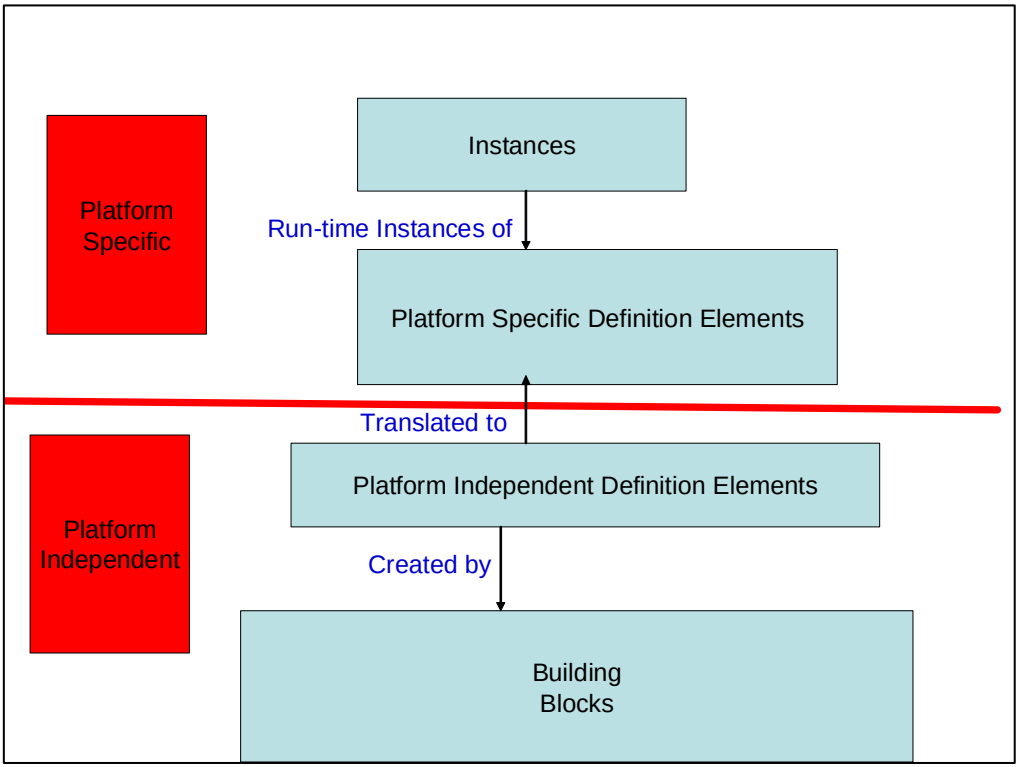


Figure 2.6 Building blocks, definition elements and instances, and how they relate to each other

To illustrate this relationship, let's look at the concrete example shown in Figure 2.7; in this example we are using the *event type* building block to define the event types needed for our application. In particular we have created a `Delivery Request` *event type* definition element which is then translated into the platform-specific definition element representing that event type. There will be many instances of this event type that occur when the application is running, in figure 2.7 we see one of these: `Delivery Request 3329` is a specific instance of the `Delivery Request` *event type* created by the Great Flower Shop. It contains some specific attributes that match the schema of the event type defined by the `Delivery Request` definition element.

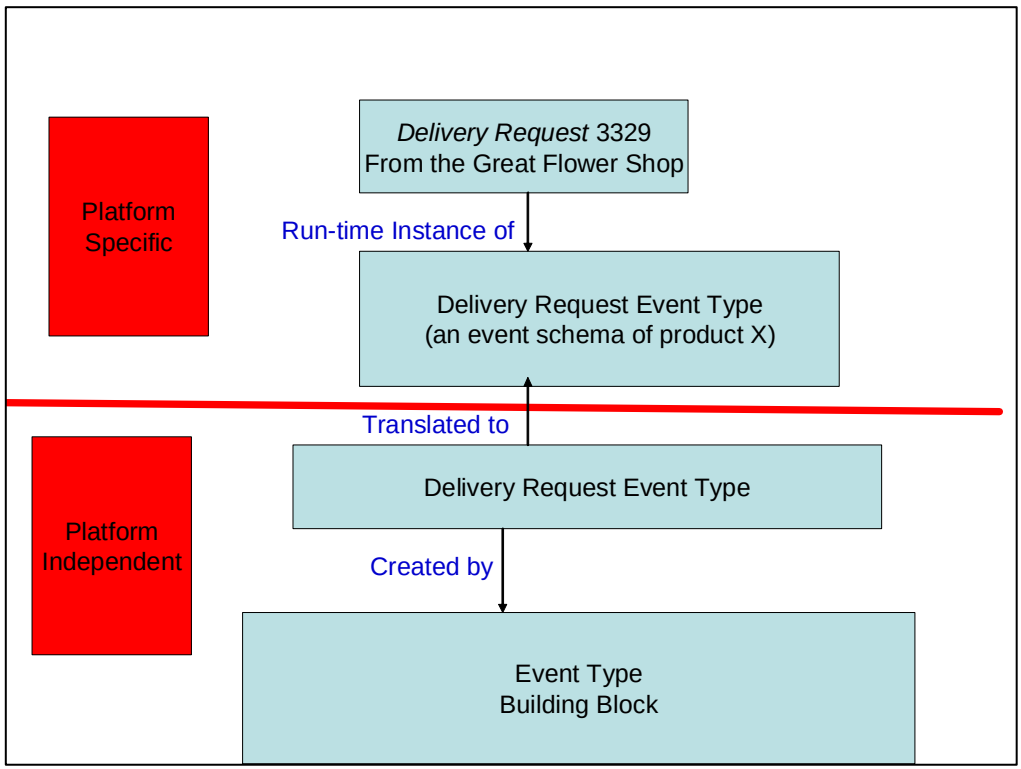


Figure 2.7 The *Event Type* Building block and corresponding definition elements

2.6.2. What information is contained in a building block?

There are several different types of building block, but they all have a similar structure. We will illustrate this by looking at a specific example, the *event channel* building block shown in Figure 2.8.

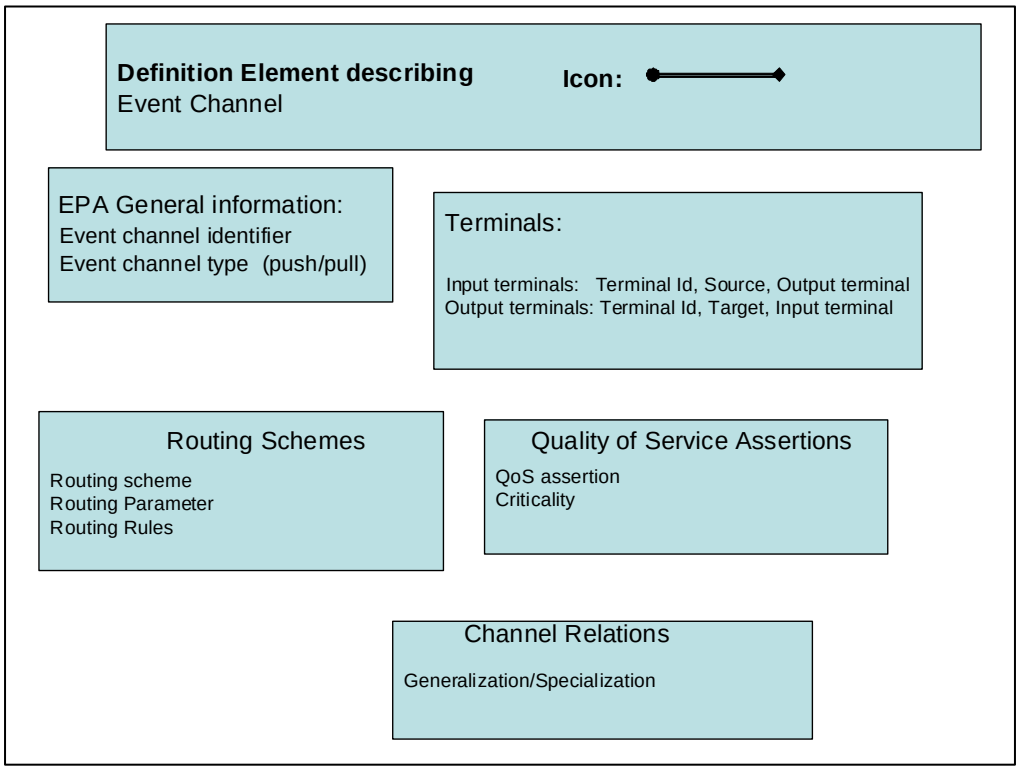


Figure 2.8 Information contained in the Event Channel building block

The full details of this specific building block are explained in Chapter 6, this figure is intended to demonstrate the various parts of a building block.

- Each building block has a name and this name determines the type of platform-independent definition elements that it describes. In this example the name is *event channel*. It also has an icon which can be used to depict these definition elements in graphical representations.
- In order to create a definition element from a building block you need to provide certain pieces of information. The building block describes what this information is. In this example the required information includes a name for the event channel.
- A building block will also have a number of relationships with other building blocks. In this example there is a relationship between *event channel* and *event type* denoting the (one or more) event types that can flow on the channel.

In part III we will explain all the concepts used to build an event-driven computing application in terms of these building blocks, and we will demonstrate this by using building blocks to specify the "Fast Flower Delivery" application.

In section 2.6.1 we introduced the idea of an event processing *building block*. These building blocks give us the “primitive components” which are put together to form event-driven applications. To use a metaphor from chemistry, the building blocks are like chemical elements, definition elements are like atoms and the applications built from them correspond to molecules. In this section we review the different types of building block (the “periodic table” in our metaphor) and in the next section we will look at the event processing network – the mechanism used to build applications from these atoms.

There are seven fundamental building blocks[17] and these are shown in Figure 2.9. Some building blocks contain references to others, and some of these relationships are shown by arrows in the figure.

We will give a detailed description of each of these types of building block in part III, where you will also be able to see how they are used in the “Fast Flower Delivery” application, but we will give a brief summary of each of them here.

Any event-driven application will involve one or more different types of event and, as its name suggests, the *event type* building block allows us to describe these types. This building block defines the structure of an event (this is sometimes called an “event schema”) along with some of its semantics.

The *event producer* and *event consumer* building blocks are used to represent the concepts of the same name that we defined in Chapter 1. The event producer represents an application entity that emits events into the event processing network, and the event consumer an application entity that receives them. It is important to note that we are using these building blocks to model just the projection of the producer and consumer onto the event processing system, that is to say those bits of the behavior of the event producer or consumer that are visible to other components of an event processing network. So the *event producer* building block does not specify how an event producer instance actually comes to emit an event, and the *event consumer* building block does not specify what an event consumer instance does when it consumes an event.

---

[17] There are a few additional building blocks that complement these fundamental ones. They will be discussed later in the book.

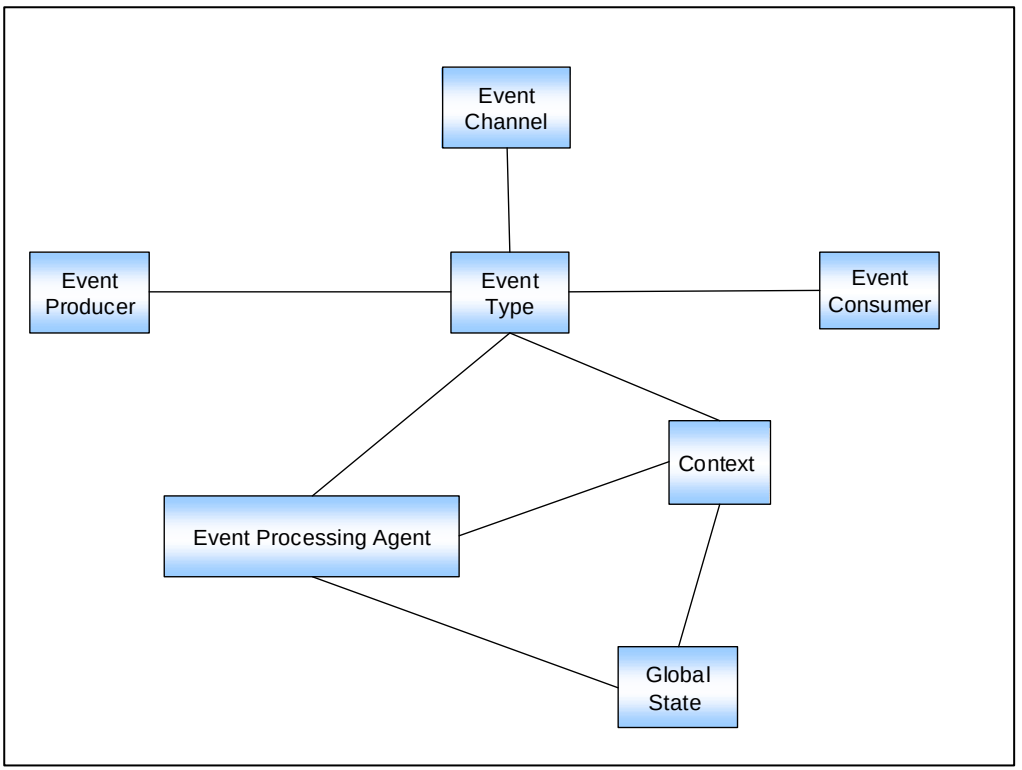


Figure 2.9. The seven fundamental building blocks

The *Event Processing Agent* building block represents a piece of intermediary event processing logic inserted between event producers and event consumers. In contrast to the event producer and event consumer, the *Event Processing Agent* building block does model the behavior of the agents built from it. There are various different kinds of event processing agent and we will look at them in a minute.

An *event channel's* principal job is to route events between event producers and event consumers (recall that we saw an example of a channel being used in figure 2.4). We will look at channels a bit more when we examine event processing networks in the next section.

The five building blocks we have mentioned so far should be familiar to you if you have read chapter 1 since they represent concepts that we introduced there.

We add two additional building blocks, the *context* building block, and the *global state* building block.

A context element collects together a set of conditions from various dimensions (temporal, spatial, segmentation oriented, state oriented) that partition the set of event

instances so that they can be routed to appropriate agent instances. For example, our "Fast Flowers Delivery" `Bid Collection` agent has a context with a semantic dimension that refers to a certain delivery request, and with a temporal life-span that starts with the `Bid Request` event and ends two minutes later. Contexts are further discussed in Chapter 7.

A global state element refers to data that is available for the use both by event processing agents and by contexts. This data may be system-wide global variables, reference data for enrichment, and event stores that hold past events. Global states are further discussed in Chapter 6.

2.6.3 Event processing agent building blocks

There are several different kinds of event processing agent, and so the event processing agent building block has a number of different subclasses as shown in Figure 2.10.

Again we will be covering these in detail in later chapters so we will confine ourselves to a few brief comments here:

- *Filter agents* – These are used to eliminate uninteresting events. A filter agent takes an incoming event object and applies a test to decide whether to discard it or whether to pass it on for processing by subsequent agents. The Filter agent test is stateless, in other words a test based solely on the content of the event instance. An example would be a test that discards a `transaction reported` event if its value is less that $100.
- *Pattern Detection agents* - These perform tests that involve some state or context in addition to the content of the incoming event object, for example a test that discards the first two failed `logon attempt` events, but passes through subsequent ones. Pattern detect agents can emit synthetic events that describe the pattern that they have detected instead of, or in addition to, passing through the incoming event objects.
- *Transformation agents* – These modify the content of the event objects that they receive. They can be further classified based on the cardinality of their inputs and outputs.
- A *translate* agent takes each incoming event object and operates on it independently of any preceding or subsequent event objects. It performs a "single event in, single event out" kind of operation.
- A *split* agent takes a single incoming event and emits a stream of multiple event objects, in other words it performs a "single event in, multiple events out" operation.
- An *aggregate* agent takes a collection of incoming event objects and produces an output event that is a function of the incoming events. It performs a "multiple events in, one event out" operation.
- A *compose* agent takes multiple collections of incoming event objects and operates on them to produce one or more output events. This is similar to the join operator in relational algebra.

Two special kinds of translation appear sufficiently frequently for it to be worth assigning them their own building blocks. They are the *enrich* agent which augments an event object with additional information (for example a customer address derived from a customer identifier in the incoming event), and the *project* agent which deletes information from the incoming event.

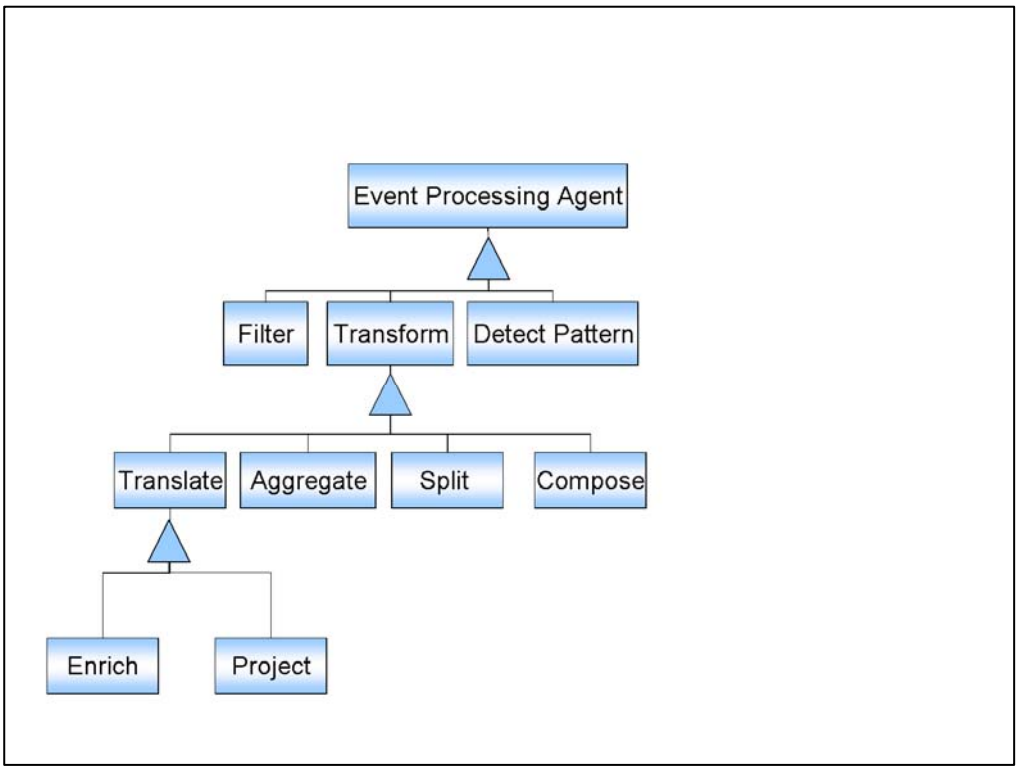


Figure 2.10 Different kinds of Event Processing Agent

You may notice that these descriptions have referred to "incoming events" and have talked about emitting passing events on to subsequent agents for further processing. This leads us naturally to the question of how event processing agents are connected together to form applications – and that's what we are going to look at in the next section.

2.7 Event processing networks

In this section we discuss how to represent an event processing application using an event processing network and we introduce the graphical notation that we use to describe event processing networks. This section serves as an introduction; we will, as usual, be filling in more of the details in Part III.

We have already met the main ingredients of an event processing network – they are none other than the *Producer*, *Consumer*, *event Processing agent* and *Channel* definition elements. Before we get on to networks, we need to take a look at the externals of these definition elements. We will start with the *Event Processing Agent*, illustrated in Figure 2.11, which also serves as an introduction to our graphical notation.

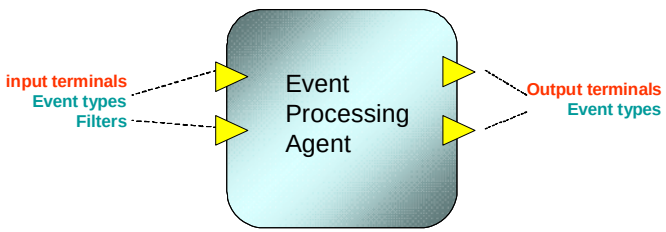


Figure 2.11 The external appearance of an Event Processing Agent

We represent each agent definition element as a rounded square box containing one or more named "input terminals" which can receive events from other entities, and one or more "output terminals" on which it emits events. The number of terminals varies depending on the kind of agent it is, so for example a *filter* agent might have a single input terminal and a single output terminal, whereas a *compose* agent might have two input terminals (one for each of two input streams) and a single output terminal.

The input and output terminals can be marked up with a set of event types to show the types of event that they are prepared to receive (input terminals) and the types of event that they are able to emit (output terminals). The terminals also indicate whether they support push or pull distribution. The default is push, meaning that an output terminal can choose when to emit an event and an input terminal undertakes to accept unsolicited input In

addition input terminals can be assigned filter conditions which restrict the set of event instances that will be accepted by the agent.[18]

The definition elements for *event producers* and *event consumers* have similar graphical representations and are shown in figure 2.12. They also have terminals – although producers only have output terminals, and consumers only have input terminals.

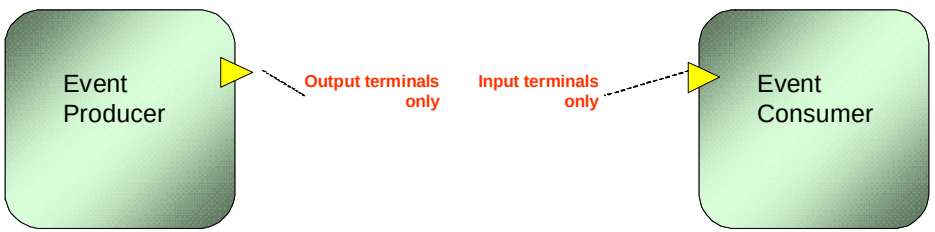


Figure 2.12 Event Producer and Consumer definition elements

So now we are ready to assemble a collection of event producers, event consumers and event processing agents together to form an event-driven application. We do this by specifying an *event processing network*. An event processing network gives us a way to define the set of event producers and event consumers that participate in the application, to specify the possible routes by which events from the producers can find their way to the consumers, and to specify the intermediate event processing (if any) to be applied to these events *en route* between producers and consumers. In addition an event processing network is a vehicle for specifying non-functional characteristics (including ordering constraints, reliability, security, performance requirements).

An event processing network could be completely static, containing a fixed set of producers, consumers and event processing agents, or it can be more dynamic:

- Producer or consumer instances may come and go dynamically. or
- The set of event processing agents and the interconnections between them may vary dynamically

[18] A purist might argue that the ability to specify a filter is not strictly needed, since one could achieve the same effect by inserting an explicit Filter agent upstream from the agent in question. The reason for including a filter in the specification of the agent is that the logic of the agent proper might be dependent on the filter, and it therefore makes sense for the processing logic and the filter to be specified in a single entity.

It is important to note that an event processing network is an abstraction. An underlying implementation does not have to slavishly reproduce the artefacts in the event processing network. What is important is that an implementation should reproduce the behavior specified (as far as it goes) by the event processing network. An implementation also has to worry about a physical provisioning of artefacts to computing nodes (something that looks like a single agent in an event processing network may in fact be realised by multiple processes running on multiple computers).

An event processing network is a directed graph made by taking a collection of producers, consumers, agents and connecting together their input and output terminals in an appropriate manner. Figure 2.13 shows the components of a very simple event processing network. We have left some details out of this picture, such as markup attached to the terminals, so as to make it easier to see the main features of the representation.

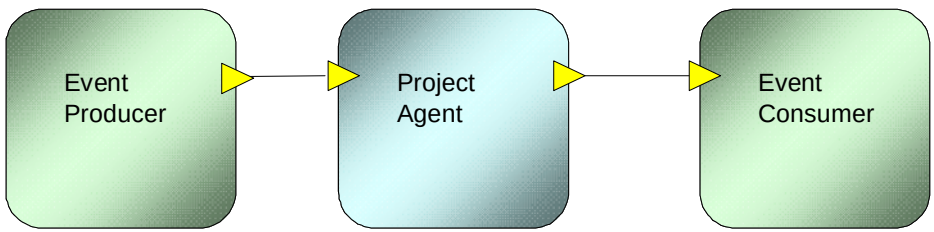


Figure 2.13 The graphical representation of a very simple event processing network

The picture in figure 2.13 looks quite straightforward, but it's worth taking a few minutes to walk through it.

The presence of the Event Producer box indicates that the application consists of just one class of Event Producer. We use the word "class" here deliberately. In some applications there might be just one instance of the producer (for example if the producer is a firewall router raising alert events); in other cases there might be many instances (for example smoke detectors in a building). Where there are many instances it would be tedious to require every one to be represented in the diagram; moreover making the boxes stand for classes rather than instances makes it easier for a diagram to represent a dynamic network. So in the smoke detector example there is no need to modify the diagram each time a new detector is added. Similarly the Event Consumer box implies the presence of a class of consumer, and this allows for dynamic consumer registration to be modeled (you will recall in section 2.3 that we noted that dynamic consumer registration is a feature of publish/subscribe systems).

The link between the event producer and the project agent indicates that any event instance emitted by the event producer is distributed to the project agent. As we noted earlier there might be a filter associated with the input terminal to the project agent, in which case a particular event instance might not actually be processed by the agent. Also the word “distributed” should not be taken too literally – the EPN diagram is a just a logical representation, and an implementation could choose to optimize the event flow so as to filter out events further upstream. The direction of flow of events is implied by the terminals involved. In this case events flow from the output terminal on the Event Producer to the input terminal on the project agent. The presence of a link does not necessarily imply that events are distributed in a push fashion – as we noted earlier the terminals themselves indicate whether they support push or pull and so the combination of the two terminals determines which approach is used. Moreover a link does not prescribe a particular mechanism for transporting events, though there might be some quality of service requirements associated with it which influence this.

An Event Processing Network graph can include a more complex distribution pattern such as that shown in figure 2.14.

In this example there are two links emerging from a single output terminal on the Event Producer. This means that when the producer emits an event through this terminal, two copies of the event instance are distributed, one to projection agent 1 and the other to project agent 2 (subject of course to any filters on the input terminals of these two agents). The network does not impose any constraints on the order in which this happens – in one implementation the two events could be distributed concurrently, while in another the distribution could be serialized, so that distribution to one of the two producers does not start until some time after the event has been distributed the other one.

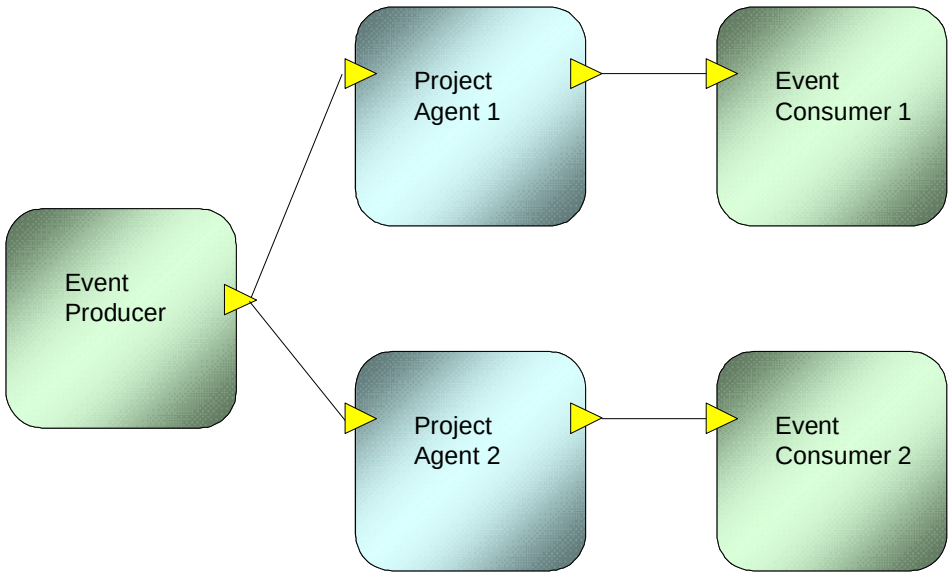


Figure 2.14. A producer with an output terminal connected to two agents

It is also possible to have two different output terminals linked to a single input terminal as shown in Figure 2.15.

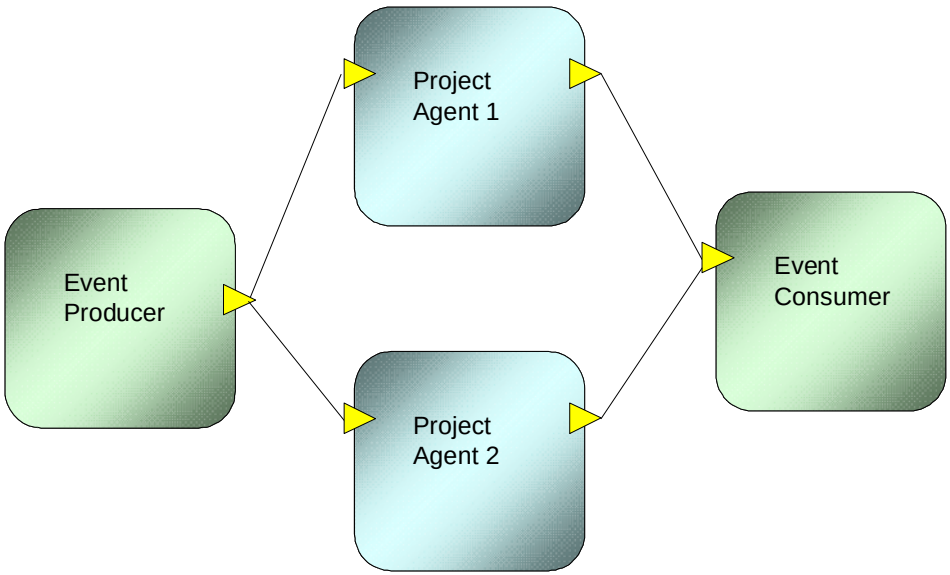


Figure 2.15. Two agents connected to a single consumer input terminal

In this somewhat contrived example we have events from a single producer being processed in two different ways by two different agents after which they are then delivered to a single consumer. The fact that two links converge on a single input terminal just implies that events from the two agents are interleaved in some order. It says nothing about the particular order of interleaving – if the order is important then the application should use an agent to perform the interleaving explicitly.

The links that we have been talking about, shown as lines in the graphical notation, are special cases of event channels. Simple channels like these do not need to be explicitly modeled using definition elements since each one only connects a single input terminal to a single output terminal and their behavior is fully specified by the constraints and requirements attached to those terminals. However there are occasions where more sophisticated routing is required, and for these we can use a fully modeled event channel definition element (using the channel building block) as illustrated in Figure 2.16.

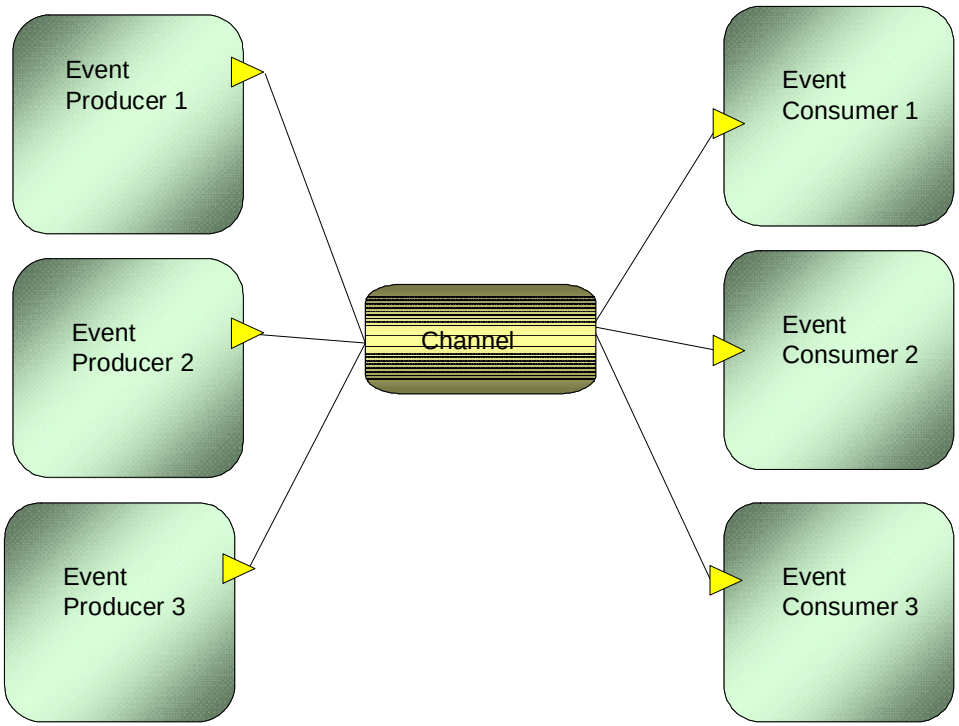


Figure 2.16. Use of an explicitly modeled channel to route events

We saw one use of such a channel earlier – to convert between push and pull distribution and thus allow an input terminal that only supports pull to be connected to an output terminal that only supports push. Modeled channels have some further advantages:

- They have a name, which means that producers, consumers and agents can be linked to a channel rather than to each other. Not only can this reduce the complexity of the event processing diagram (compare figure 2.16. with figure 2.17. which shows a similar topology that does not include a channel), it also means that you can add or remove producers, consumers or agents without having to know the names of other producers. consumers of agents in the network
- They have their own configuration parameters. This means that you can specify particular routing behavior or associate particular quality of service requirements with a modeled channel and have that apply to all events that pass through the channel.

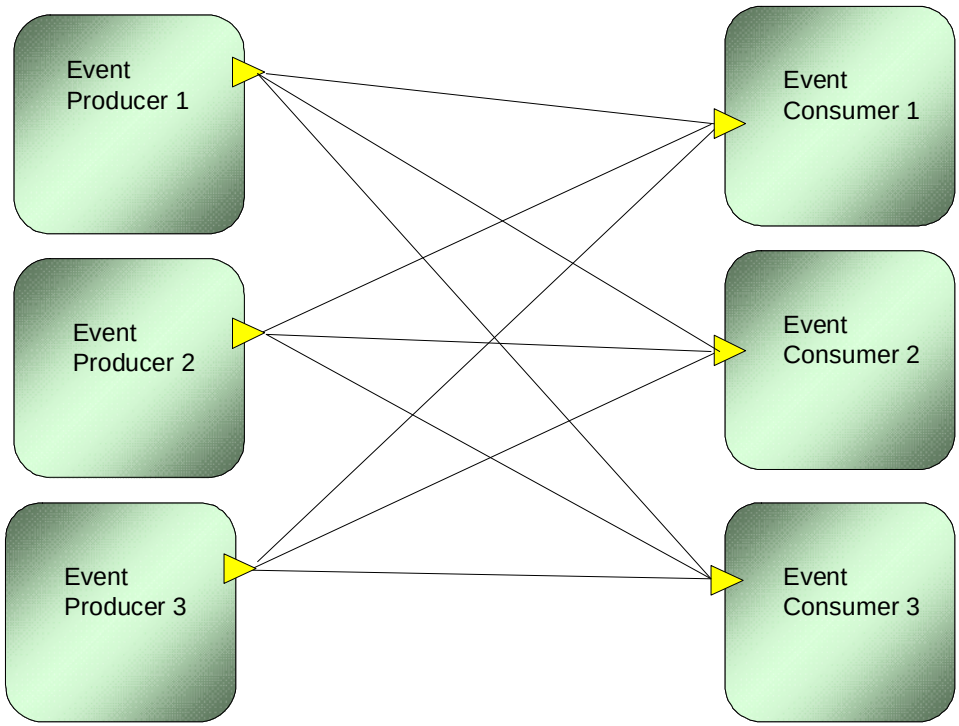


Figure 2.17. The additional interconnects required if an explicit channel is not used.

As this figure shows, to connect N producers to M consumers, you need NxM interconnecting links if you don't use an explicit channel, whereas with an explicit channel you can reduce this to N+M links. If N or M is large this can be significant.

2.8 Summary

In this chapter we have discussed the basic concepts of event driven architecture, positioned the event processing area within the IT world by looking at some related concepts and technologies, and we have introduced the seven event processing building blocks.

We have now introduced all the concepts that we will be using in the rest of the book and can move to the in-depth discussion about each of them.

Additional reading

Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003

http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Deploying/dp/0321200683/ref=sr_1_1?ie=UTF8&s=books&qid=1258829949&sr=1-1

This book explains in depth many of the concepts discussed here, such as: request/response, it is also the major book that describes the message oriented functionality.

Bud Smith, Push Technology for Dummies, For Dummies, 1997.
http://www.amazon.com/Push-Technology-Dummies-Bud-Smith/dp/076450293X/ref=sr_1_2?ie=UTF8&s=books&qid=1258830719&sr=1-2

This book describes the fundamentals of push technology.

Hugh Taylor, Angela Yochem, Les Phillips, Frank Martinez : Event-Driven Architecture, Howe SOA Enables the Real-Time Enterprise. Addison-Wesley, 2009.
http://www.amazon.com/Event-Driven-Architecture-Enables-Real-Time-Enterprise/dp/0321322118/ref=sr_1_1?ie=UTF8&s=books&qid=1258889333&sr=1-1

This book explaining through collection of use cases what is EDA and how it is related to SOA.

Judith Hurwitz ,Robin Bloor , Carol Baroudi , Marcia Kaufman
Service Oriented Architecture For Dummies (For Dummies (Computer/Tech))
For Dummies, 2006
http://www.amazon.com/Service-Oriented-Architecture-Dummies-Computer/dp/0470054352/ref=sr_1_2?ie=UTF8&s=books&qid=1259174835&sr=8-2

This is one of many books that explain what SOA is.

Rainer von Ammon: Event-Driven Business Process Management. Encyclopedia of Database Systems 2009: 1068-1071
http://www.springerlink.com/content/p267j78082568086/

This is an encyclopedia entry explaining the notion of Event-Driven Business Process Management.

Kun-Lung Wu, Philip S. Yu, Bugra Gedik, Kirsten Hildrum, Charu C. Aggarwal, Eric Bouillet, Wei Fan, David George, Xiaohui Gu, Gang Luo, Haixun Wang:
Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S. VLDB 2007: 1185-1196
http://www.vldb.org/conf/2007/papers/industrial/p1185-wu.pdf

This article provides an example of general stream computing platform

David S. Frankel: Model Driven Architecture: Applying MDA to Enterprise Computing Wiley, 2003.
http://www.amazon.com/Model-Driven-Architecture-Enterprise-Computing/dp/0471319201/ref=sr_1_1?ie=UTF8&s=books&qid=1258889376&sr=1-1-spell

This book explains the principles of model driven architecture

Exercises

2.1 Give examples of two applications where you would use pull distribution to retrieve events from an event producer, one example using periodic pull and one using ad-hoc pull.

2.2 Draw the event processing network for an application that can produce or consume Twitter events and that uses at least five different types of event processing agent.

2.3 Give an example of an application in which a channel needs to be programmed explicitly.

2.4 Give an example of an application which contains a component that is both an event consumer and an event producer. Is the fact that both producer and consumer are combined in a single component significant to the event processing system? If so, what purpose this information can be used for?

2.5 Give some examples of other classification schemes that use a four level classification similar to that shown in Figures 2.6 and 2.7.

2.6 Try to define a platform independent definition element of one of the event channels that exist in the Fast Flower Delivery application.

2.7 An "Event Processing Network" diagram looks similar to a workflow diagram (such as BPMN or a diagram used to represent WS-BPEL) as both of them make use of directed graphs. Can you explain the differences in the semantics of nodes and edges between these two kinds of diagram?

2.8 Do you think that it is possible, or makes sense, to create a unified graph that contains both workflows and event processing networks?

3

Defining the events

"When I can't handle events, I let them handle themselves."
- Henry Ford

We now start our deep dive journey by explaining how event types are defined. This will give you the opportunity to see definition elements in use. In this chapter we discuss the following topics:

- *Event type* definition elements
- Temporal and spatial characteristics of events
- Relationships between events and other things: event generalization and other relationships between events; references from events to application entities
- The first step in building the "Fast Flower Delivery" application, which is to create definition elements for all the event types that we will be using in the application
- How event formats are represented in some event processing systems.

We will start with a discussion of event types and attributes, moving to the definition elements that describe them. This is our first in-depth discussion of a definition element; we will be following a similar approach in the following chapters as we work through all of the components of our application.

3.1. Event types

You will recall that in chapter 1 we discussed two meanings for the word *event.* One meaning refers to something that has happened (the "event occurrence")and the other refers to the programming entity used to represent it (the "event object") . It's the second of these that we are mainly concerned with in this chapter, and we will be talking about the design of

these event objects. However when designing an event object you need to think about the event occurrence and decide which aspects of it should be included in its representation.

Although we talked about "designing an event object" in the previous paragraph, in practice you don't have to design each individual object, as in an event processing application you usually encounter many event instances that have a similar structure and a similar meaning: consider for example the stream of events coming from a temperature sensor, all the events contain the same kind of information though of course each event will have a different time stamp and will be reporting a different temperature value. So instead of defining the structure of each event, you just need to specify the structure of this entire class of events. Since this is similar to defining a reusable data type in a programming language, we refer to this specification as an *event type*.

Definition

An *event type* is a specification for a set of event objects that have the same semantic intent and same structure; every event is considered as an instance of an event type.

Each event type has a unique event type identifier. In this book we will be using simple descriptive text strings for these identifiers, and we will write them in `this typeface`. We will also be using the event type identifier as an adjective to describe an event instance; the phrase "`Delivery Request` event" reads more easily than the clumsy alternative "event of type `Delivery Request`". If you revisit the section in chapter 2 where we introduced our Fast Flower Delivery application you will see that we have already mentioned several event types in this way.

There are systems that use un-typed events. These are systems which do not impose any particular structure on their event objects, for example they might simply represent each event object as a single character string. However, the applications running in such systems usually impose their own typing systems on these events. In the context of this book, therefore, we are assuming that events do have types associated with them, and that any event objects that start off life as un-typed events are translated to typed events before being processed in an event processing network; we discuss this further when we look at event consumers in the next the chapter.

3.1.1. The Logical structure of an event

An event type should help answer questions such as: "what happened?", "when did it happen?", "where did it happen?", "what other information is associated with its happening?" As we will see, the answers to these questions can be recorded at various levels of precision, and you need to take the requirements of your application into consideration. For some applications all these questions are relevant, and for others only some of them will be. While lack of information is a problem, surplus information is also a burden, so when designing an

event type you need to consider the amount of information that will be required by the application.

We distinguish between two kinds of information carried in an event object. The first, which we call header data, consists of some meta-information about the event carried in the form of *header attributes*, the second – the payload - contains specific information about the occurrence itself.

Many event processing systems offer a set of system-defined header attributes and provide a mechanism for application designers to add further attributes. In keeping with the implementation-neutral approach that we are following in this book, we are going to describe some platform-independent header attributes, which can be mapped to the platform-specific ones found in actual event processing system implementations.

The header attributes that we are going to define fall into two categories:

- Attributes that describe the Event Type itself: these include an attribute that gives the event type identifier, as well as an attribute that indicates whether this is composite event or not, and one that gives the precision of any time values given in the event object.
- Event instance attributes: these are additional header attributes that carry information specific to this instance of the event, such as an instance identifier and time stamps. Some of the attribute values may be inserted into the event instance by the event processing system rather than by the event processing application.
- As we mentioned earlier, you don't necessarily need every event object to carry the full complement of header attributes, so we use the event type definition to specify which of the event instance attributes are required, and which are permitted in an event object.
- The event payload consists of a collection of attributes. Each attribute has a data type given by the event type definition element. The data type can be a simple type such as a string or numeric value, or (in some systems) it can be a complex data structure. The definition element can also indicate how many times a given attribute is required or permitted to appear in the payload. For simplicity all our examples will be of types where each payload attribute appears once and only once.
-

3.1.2. The event type definition element

In this section we give a quick summary of the event type definition element. In the sections that follow we will work through all the header and payload parts of an event, describing the attributes that they contain and the corresponding entries that appear in the definition element.

Figure 3.1 shows the structure of the definition element. As we saw in chapter 2 every definition element has a type (in this case it's an *event type* definition element) and icon, some information specific to the kind of definition element (in this case the attributes that

make up the event type organized into header and payload). It also contains a list of references to other definition elements; in this case these are event-to-event relationships including retraction events, event generalization and event specialization.

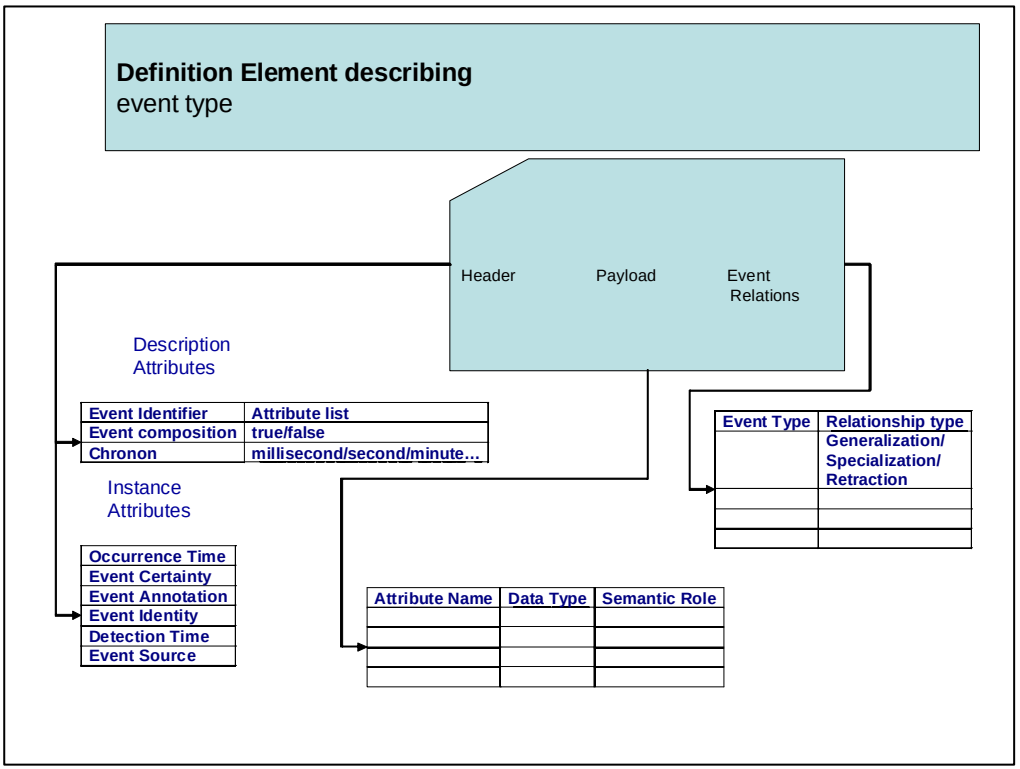


Figure 3.1: The event type definition element consists of attributes that describe the contents of the event header and payload, and a list of relationships between this event type and other event types

In the next three sections we discuss the attributes and relation entries shown in Figure 3.1, along with the corresponding header and payload attributes that appear in the event objects themselves.

3.2 Header attributes

Header attributes contain meta-information about the event that is useful when processing the event. The name and meaning of these header attributes is not specific to a particular event type.

An event type definition element can indicate which of these header attributes must, may or must not appear in an event instance. If an attribute is required to appear in every event instance it is shown as "mandatory", if it is permitted but not required it is shown as "optional", and if its presence is forbidden we show it as "not-applicable". If a definition element does not specify whether an attribute should appear or not, then the requirement for that particular attribute is specified by the application itself via the *application defaults* definition element.

3.2.1 Event type description attributes

In this section we define the "description attributes" shown in the left hand box of Figure 3.1. These attributes describe the event type itself.

Definition

The *Event type identifier* attribute identifies the event type definition that describes the event instance.

As we said earlier, we are assuming that we are dealing with event processing systems that support typed events, and so every event instance has an *event type identifier* attribute, or collection of attributes. This attribute is generally carried in the event object[1]. The Event type identifier also appears in the definition element where it identifies each Event Type definition element instance.

Definition

The *Event composition* attribute is a Boolean attribute that denotes whether the specific event instance consists of composition of several events or not.

Many systems support *composite events* (a composite event is one whose payload is made up of several different event instances, possibly themselves of different types). Composite events have a special kind of event type (a composite event type) that constrains the content of the event by defining the types of event that it is allowed to contain. This attribute forms part of the event type definition element, but can be included as a header attribute in an event object as a convenience to any processor of that object. The default value of this attribute is false, so an event type is considered as non-composite if this attribute is not explicitly specified in the definition element.

---

[1] In some systems the type can be inferred from the context, or "compiled in" to the producer, consumer or agent code and so does not physically need to be included in the event object, however this can be viewed as an implementation-provided optimization.

Definition

The *Temporal Granularity (or Chronon)*[2] attribute denotes the "atom of time" from a particular application's point of view. It stands for the unit in which time-stamps in the application are being measured, examples: second, minute, hour, or day.

Different applications have different requirements when it comes to the precision at which time measurements are taken, and this is normally something specified by the application rather than something that that is set as part of an event type definition. However, it can be overridden for a specific event type, for example you might have an application where most of the events have a granularity of a minute but where there is a specific event type in the application that requires a granularity of a second. The chronon value applies to timestamps in the event such as the *Occurrence Time* and *Detection Time* attributes, defined below, and affects the temporal related processing of events of this type. It also lets an event producer know how accurate the time-related attributes in the event have to be when it is constructing an event instance. It can appear in the event type definition and also, as a convenience, in an event object. The order of precedence is: the value from type definition, if there is none then the value specified as an application default, and if none is provided there either then an implementation default can be used.

In the Fast Flower Delivery example we define the temporal granularity default for all time-stamps to be one minute, but we override it for the event type `Delivery Bid` to have a granularity of a second.

The following code snippet shows the notation we are using to show these values in a definition element. This example is the definition element for the `Delivery Bid` event type from our Fast Flower Delivery application. Later in this chapter we show some concrete examples of event definitions in various languages

```
Definition Element of Event Type Definition element
Event Type Identifier:= Delivery Bid
Event Composition Indicator := false
Temporal granularity := Second
```

In this example the Event Type Identifier identifies the event type, the event is not composite (which is actually the default and does not need to be specified), and the temporal granularity is *second*, which overrides the application's default of *minute*. The rationale here is that for most events in this application a minute's granularity is sufficient, however we

[2] The term Chronon is taken from the glossary of temporal databases, additional material about temporal databases can be found in the additional reading section at the end of this chapter.

need to have a finer granularity for bids since we may want to order event instances using the time at which they were sent.

3.2.2. Event instance attributes

These attributes carry additional meta-information about an event instance. If present in the header of a particular event object they carry information about that object. This includes:

- Instance attributes whose values are set by the application: Occurrence time, event certainty and event annotation
- Instance attributes that are system generated: Event identity, detection time, and event source.
- As we mentioned earlier, the event type definition element can contain indicator entries, these indicators show which attributes must, may or must not appear in the header of an event of that particular type.

Definition

The *Occurrence Time* attribute is a time stamp with a precision given by the event type's temporal granularity (Chronon)[3]. It records the time at which the event occurred in the external system.

The occurrence time of an event is provided by the *event producer* that detects the event, along with the other attributes of that event. For example an in-car GPS system can insert an Occurrence Time time stamp derived from the GPS satellites. Although we define occurrence time as a time stamp, there are cases in which the occurrence time is better represented by a time interval and not a single time point. We shall discuss interval-based occurrence time, and interval-based event patterns later in this book.

In some cases the producer might not be able to determine the time when the event actually occurred and so cannot provide a true occurrence time, for example if the producer works by examining the state of some external entity only at periodic intervals. In this case the best we can do is to record the time at which the producer actually generated the event instance. This *detection time* is defined in the next section.

These time stamps are used in event processing in which the order of events is significant or in applications where the fact that an event happened within a certain time interval is important. There may be problems of inaccuracy if the time stamp is provided by the event producer, especially when the using a short time granularity. We discuss this further in the advanced temporal issues section in Chapter 11. In our use case we assume that occurrence

[3] The actual representation may be presented in a finer granularity, but for all processing purposes, it should be rounded to the chronon granularity.

times generated by the various event producers are accurate enough for use with a minute chronon.

There are two more event instance attributes, *event certainty* and *event annotation*:

Definition

The *Event certainty* attribute denotes an estimate of the certainty of this particular event.

The event certainty has a value of 1.0 if it is certain that this event occurred in reality in the way described by the event payload. The discussion about uncertainty handling of events is outside the scope of this book, and in all examples we assume that this attribute is not applicable for all the events[4]

Definition

The *Event annotation* attribute provides a free-text explanation of what happened in this particular event.

Event annotation is a mechanism that the event producer can use to annotate an event instance with a human-readable explanation. This annotation may be used for documentation purposes. The text can also be processed by an event processing agent, or copied as part of a derived event. Again, this attribute may be mandatory, optional or not-applicable for a certain event type, or a complete application.

System-generated attributes are attributes which, if they appear in the event instance at all, are generated automatically by the event processing system rather than being inserted by an Event Producer or Event Processing agents. We shall discuss three such attributes: Event Identity, Detection Time and Event Source

Definition

The *Event Identity* attribute is a system generated unique id for each individual event instance

Values for event identity may be again mandatory, optional or not applicable, since some applications care about tracing individual events and some do not[5].

---

[4] Those interested to read about uncertain events, are referred to this article: Segev Wasserkrug, Avigdor Gal, Opher Etzion, Yulia Turchin: Complex event processing over uncertain data. DEBS 2008: 253-264 http://portal.acm.org/citation.cfm?doid=1385989.1386022

[5] Note that if the system is implemented in an Object Oriented language, each event object will usually have an object identity in any case

Definition

The *Detection Time* attribute is a time stamp (in the event type's temporal granularity) that records the time in which the event became known to the event processing system

Note that in practice different systems generate this time stamp in different ways. Some record the time when the event producer calls some system API to submit the event, some record the time when the event is placed into a buffer for processing, others use the time when the event is retrieved from the buffer and the actual processing starts. The way that an implementation sets the detection time is often tunable by the user, and some systems make it a mandatory attribute of all event instances.

Some implementations allow the use of buffering as a way of re-ordering out-of-order events so that they can be processed in order of occurrence time regardless of the order in which they were actually detected. The choice of whether the occurrence time or detection time ordering is to be used is something that is made by the application or the Event Processing Agent and does not form part of the event type definition.

Definition

The *Event Source* attribute is the name of the entity that originated this event. This can be either an event producer or an event processing agent

This is again an attribute that is generated by the system, and the event type definition states whether its inclusion in an event instance is mandatory, optional or not applicable. It is sometimes useful for an event processor to know where an event instance came from; in our use case an `Assignment` event can be emitted either by a `store` event producer or by an `automatic assignment` EPA.

The following code snippet shows how the attributes that we have just discussed appear in an event type definition element; once again we are using, as an example, the `Delivery Bid` event type.

```
Definition Element of Event Type Definition element
Event Type Identifier := Delivery Bid
Occurrence Time is mandatory
Event Annotation is optional
Event Certainty is not applicable
Event Identity is mandatory
Detection Time is mandatory
Event Source is mandatory
```

Occurrence Time is denoted as a mandatory attribute for each event of this type, Event Annotation as optional, and Event Certainty is shown as not applicable. All the system generated attributes are mandatory.

In summary, the major questions answered by the header are:

- **What kind of occurrence is this?** This is denoted by the `Event type`

`identifier`.

- **When did this event occur?** This is denoted by the `Occurrence Time` and `Detection Time`
- **Who emitted this event?** This is denoted by the `Event Source`, which can be an event producer for raw events or an EPA in the case of derived events.

Having looked at the header we now move to describe the event's payload.

3.3. Payload attributes

An event's header attributes carry *meta-information* about the event, such as what type of event it is, and when and where it occurred. This is generic metadata in that its syntax and interpretation is independent of the actual event type. In contrast, the attributes that make up the event payload are used to carry the data that describes the actual occurrence. You can liken this to a file in a computer file system; the payload corresponds to the contents of a file, whereas the header corresponds to file metadata such as its name, time of last access and so on. We'll start this discussion by talking first about data types that can be supported, and then talk about attributes with specific semantic roles like reference.

3.3.1. Data types

Every payload attribute has a type[6]. There is a set of basic data types that are well known from programming languages, and a couple of more advance ones. In addition attributes can have complex data types, these are structures composed of other data types.

Basic simple data types:

- String
- Integer
- Floating point number
- Fixed precision decimal number
- Binary data
- Boolean

More advanced simple data types:

- Time stamp
- Location
- Reference to another event

---

[6] Each header attribute also has a type, but their types are pre-defined, while in the payload attributes they need to be defined explicitly

Definition

A *time stamp* is a data type that denotes a certain point in time, its granularity is based on the chronon that applies to the event type.

The location data type is somewhat more complicated;

Definition

A *location* data type[7] is used to designate the location in which an event occurred in the "real world"; it can refer to domain-specific geo-spatial terms, e.g. lines and areas that are defined in this domain.

Here is a short explanation of these representation alternatives:

- Point: the event is considered as occurring in a specific geometric point in the space, using some coordinate systems (2D or 3D). Example: the GPS coordinates of a specific vehicle.
- Line: the event is considered as occurring on a line, or a polyline. Example: indicating that a vehicle is somewhere in the road that is represented by the polyline.
- Area: the event is considered as occurring within a certain geographical area (2D or 3D). Example: indicating that a vehicle is somewhere within a local authority's jurisdiction.

Figure 3.2 shows examples of these different representations being used by applications that monitor traffic on a certain highway.

[7] For complete discussion about location data types refer to: Martin Erwig, Ralf Hartmut Güting, Markus Schneider, Michalis Vazirgiannis.: Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. GeoInformatica 3(3): 269-296 (1999) http://www.springerlink.com/content/k7g1475863151870/?p=1311be22b443477a86a723bf2288efbcn=3

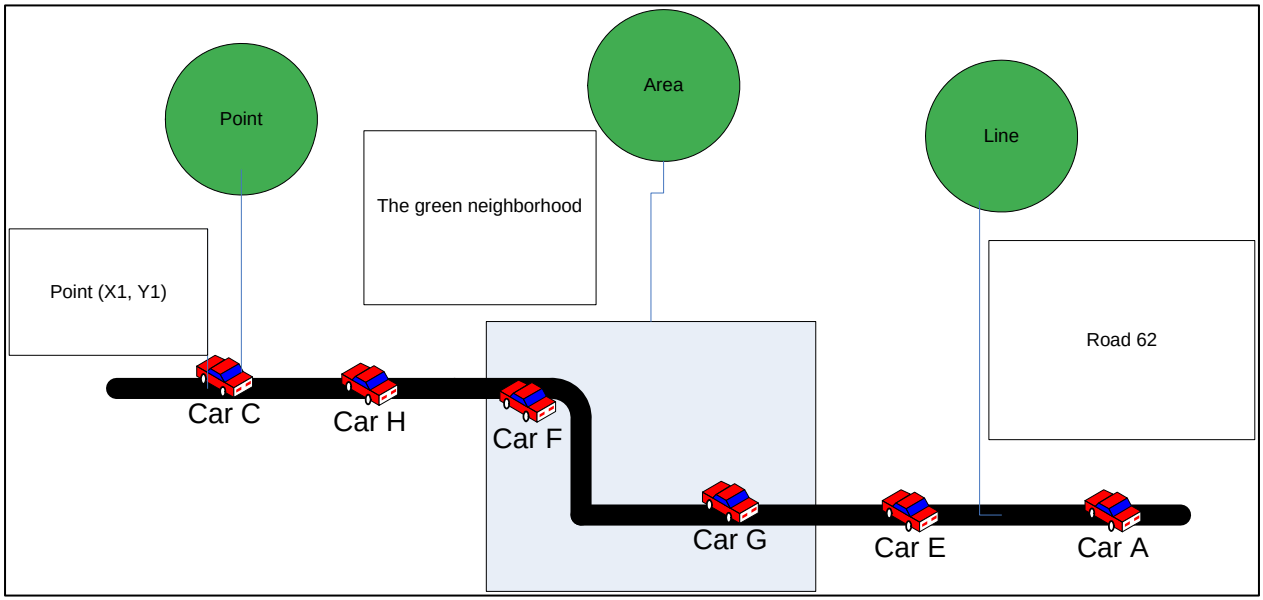


Figure 3.2 A road monitoring example showing three different location data types (point, line and area) used by different intelligent transport applications.

- Application 1 is interested in the exact location of each car; the event that designates the location of Car C is a point location determined by the car's GPS
- Application 2 is interested in granularity of neighborhoods; the events that designate the locations of Cars F and G are located in the same neighborhood, whose name returned as the value of their location.
- Application 3 is interested in cars being on a certain road; all the cars in this picture are located on Road 62, thus its metric is a line (poly-line in this case).
-

The meaning of a payload attribute can be entirely local to the event type in which it appears, or it can have can have a meaning that extends beyond that type definition. We will now look at two such semantic roles that a payload attribute can play.

3.3.2. Attributes with semantic roles

Attributes may also have semantic roles, we shall discuss two semantic roles: entity references and common attributes. We start with entity references:

Definition

An *Event Entity Reference* is an event attribute whose value is a reference to a particular entity external to the event.

In the context of this definition, an *entity* is something modeled by the application and given some kind of unique identifier by that application. It frequently models something that exists in the application's business domain, for example a customer or an order. In the Fast Flower Delivery example "order", "store" and "driver" are all entities. An EPA or Consumer processing an event can use the reference to look up information about the entity, for example finding the current ranking of a driver; the reference data type includes a scheme that tells the processor how to do this look up (for example the attribute value might be a key used to retrieve the data from a particular database table). As well as providing a way of finding out additional information, an entity reference attribute can be used to partition the context that is used to process the event. In our example all events such as `Pick up Confirmation` and `Delivery Confirmation` and their associated alerts and derived events can be partitioned using the value of the *driver* attribute.

In some cases we have attributes in two or more event types that, although they don't reference an external entity, nevertheless have the same meaning. We refer to these as *common attributes*.

Definition

A *common attribute* is an event attribute whose semantics are defined by the attribute name, so within the application domain all attributes with the same name are considered to be semantically equivalent,

In our example Request Id is a common attribute that is used across various event types and can be assumed to be semantically equivalent among these event types.

Listing 3.1 shows the domain specific attributes for the `Delivery Bid` event type

Listing 3.1 Payload of the Delivery Bid definition elements

```
Definition Element of Event Type Definition element
Event type identifier:= Delivery Bid
Payload attributes
Attribute Name      Data Type     Semantic Role
Request Id          Integer       Common attribute
Driver              String        Reference to Driver Table
Store               String        Reference to Store Table
Driver Location     Area
```

This event type has in its payload several attributes, each of them has a type: integer, string or area, and a semantic role. Driver and Store are references to entities that are

stored as reference tables in the application's global state; Request Id is a common attribute matched with other event types by name, Driver Location is of type Area

Next we discuss relationships between event types.

3.4. Event to event relations

An event type may contain references to other event types when there is semantic relationship between them. The event type definition element mentions all the event types that an event of that type might refer to. We discuss three types of event reference: retraction, generalization, and specialization.

Definition

A *Retraction* event relationship is a property of an event type referencing a second event type that is the logical retraction of the referencing event type.

We shall discuss retraction events in Chapters 10 and 11 that deal with context and pattern matching. In our example `Delivery Request Cancellation` is a retraction of the `Delivery Request` event type, since a `Delivery Request Cancellation` terminates any processing related to the delivery.

Definition

The *Event Generalization and Specialization* relationships indicate that an event type is a generalization or specialization of another event type, possibly conditioned by a predicate.

The notion of specialization (and its converse, generalization) originated in semantic data models[8], and has become popular through object oriented programming. In some cases it is more convenient to define the generalization relationship and in other cases it is more convenient to define the specialization. The essence is that if an event type E2 is a specialization of event type E1 then event type E2 inherits the definition of event type E1 with possible additions and modifications, such that each instance of event type E2 can also be considered as an instance of event type E1. This means that a consumer or Event Processing Agent that is able to handle an event of type E1 can also handle an event of type E2. A consequence of this is that additions and modifications in E2 must not substantially

[8] These terms were first introduced in: John Miles Smith, Diane C. P. Smith: Database Abstractions: Aggregation and Generalization. ACM Trans. Database Syst. 2(2): 105-133(1977)

http://www.informatik.uni-trier.de/~ley/db/journals/tods/SmithS77.html

alter the meaning of the event. In the "Fast Flower Delivery" example the `Manual Assignment` event type is a specialization of the `Assignment` event type.

There are cases in which you might want to restrict the conditions under which events of type E2 can also be considered as instance of events of type E1, by making the generalization relationship contingent on a predicate. This is best explained by taking an example (not from our use case). Let's say that we have a helpdesk system that handles problems with office equipment; there are event types related to problems with printers, screens, phones, fax machines and scanners. These problem events are all specializations of a single `hardware problem` event type, since the helpdesk has a single service covering all these types of equipment. However suppose that some people have laser printers in their offices, and that service for these printers is provided by local technicians rather that by the helpdesk service. In this case you might only want a printer problem event to be considered a subtype of the `hardware problem` event type in cases when the printer in question is not laser printer.

The following sippet shows event relations for the `Delivery Request` event type.

```
Definition Element of Event Type Definition element
Event Type Identifier := Delivery Request
Retraction Event :=  Delivery Request Cancellation
Event Generalization :=  Store Request condition = always
```

Bid Request is explicitly caused by Delivery Request, Note that multiple Bid Request events can be caused by a single Delivery Request event. The retraction event is Delivery Request Cancellation, when this occurs, the handling of the Delivery Request should be cancelled (if possible). For the sake of this example, we assume that there is a wider system in which the Delivery Request is just one type of possible Store Request, thus any processing of Delivery Request is also considered to be Store Request and participate in all processing related to Store Requests.

Now that we have described the event type definition element, we can take a look at the event type definitions in the Fast Flower Delivery use case.

3.5. Event types in the Fast Flower Delivery example

In this section we include the complete definitions of all the raw event types used in the Fast Flower Delivery example. This application also uses some derived event types, but we are deferring those until our discussion of event derivation in chapter 8.

We start by defining application defaults for all events below.

```
Application Defaults
Temporal granularity := Minute
Event Composition := false
Occurrence Time is mandatory
Event Annotation is optionalEvent Certainty is not applicable
Event Identity is mandatory
```

```
Detection Time is mandatory
Event Source is mandatory
```

These definitions belong to a definition element that defines application defaults that is discussed fully in Chapter 13; it provides default settings for values not specified in other definition elements.

The values shown in this listing apply to all event types used in the application, so in the event type definition elements that follow we shall not repeat any of these entries unless their values differ from those above.

Figures 3.3 – 3.9 present the definition elements that correspond to the raw event types used in the application.

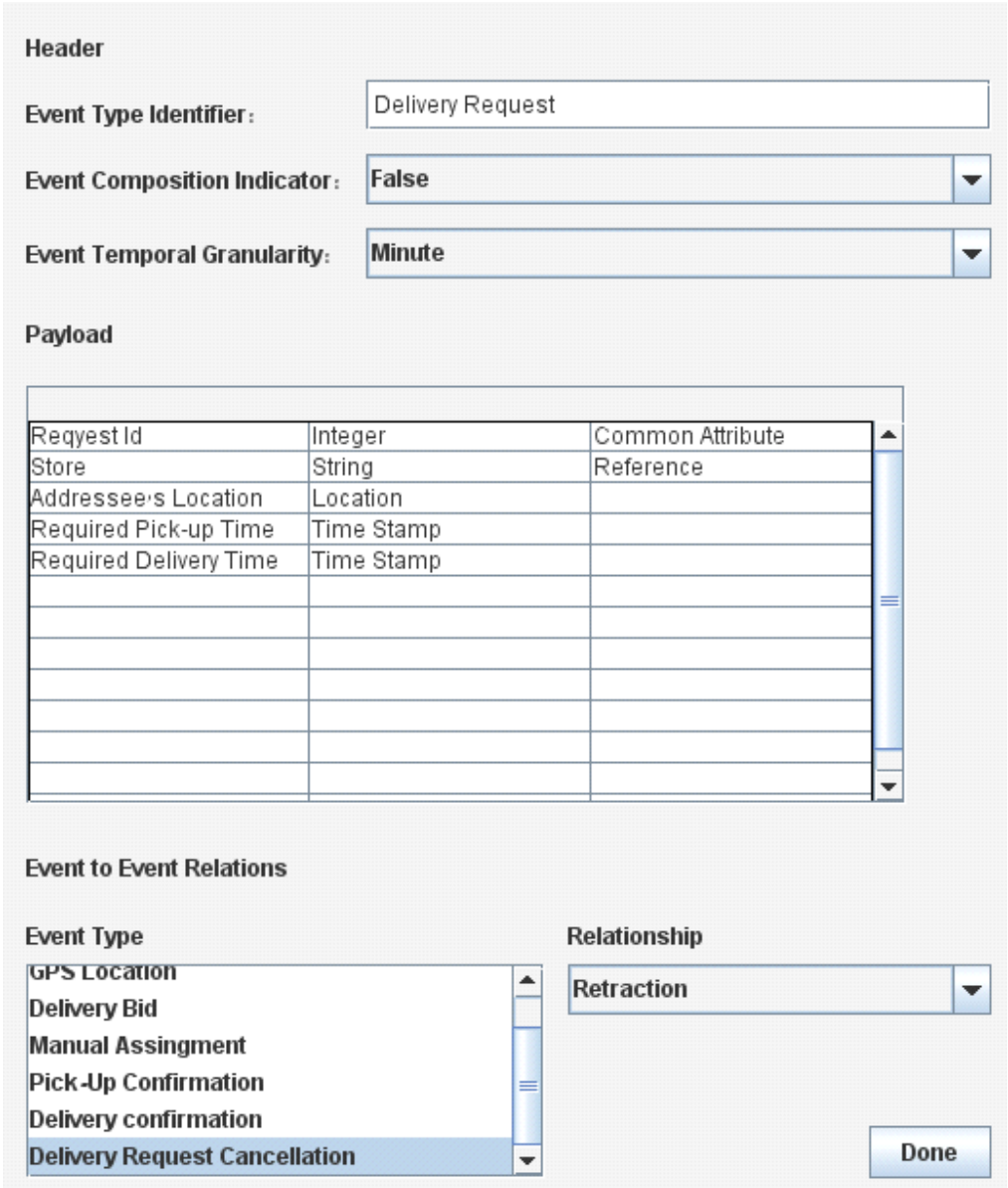


Figure 3.3 The definition element for the Delivery Request event type. The payload attributes contain details of the delivery that has been requested.

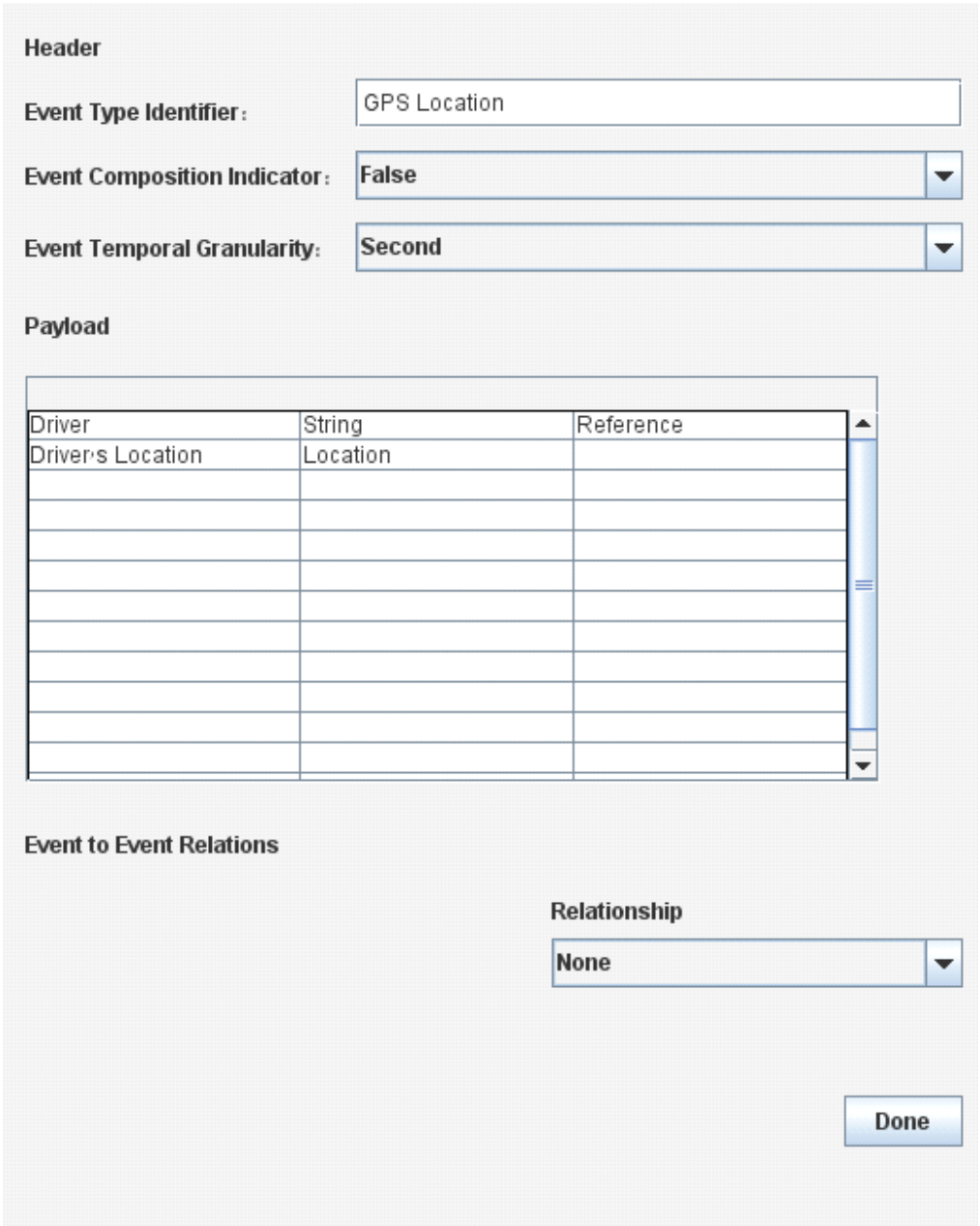


Figure 3.4. The definition element for the GPS Location event type

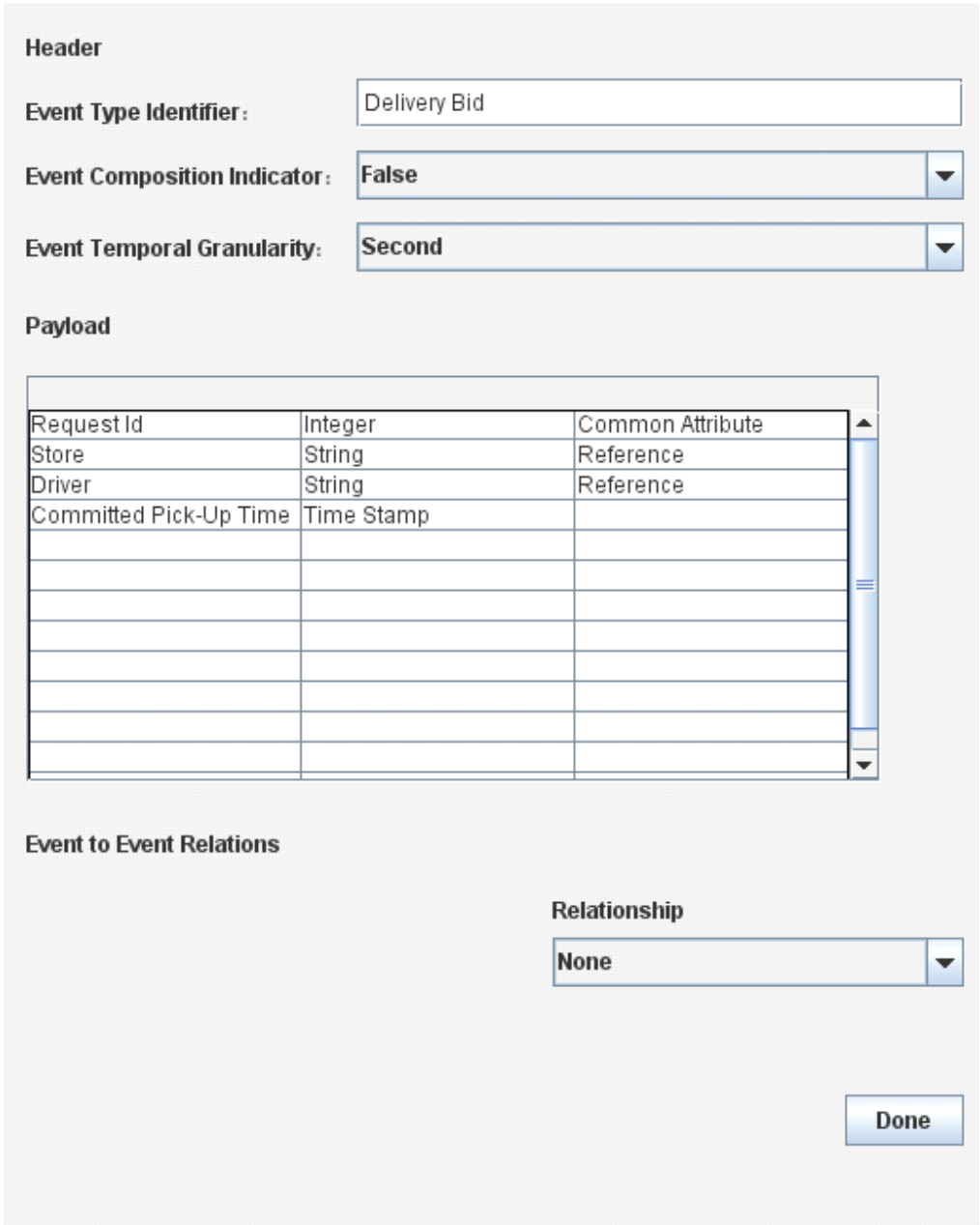


Figure 3.5 The definition element for the Delivery Bid event type

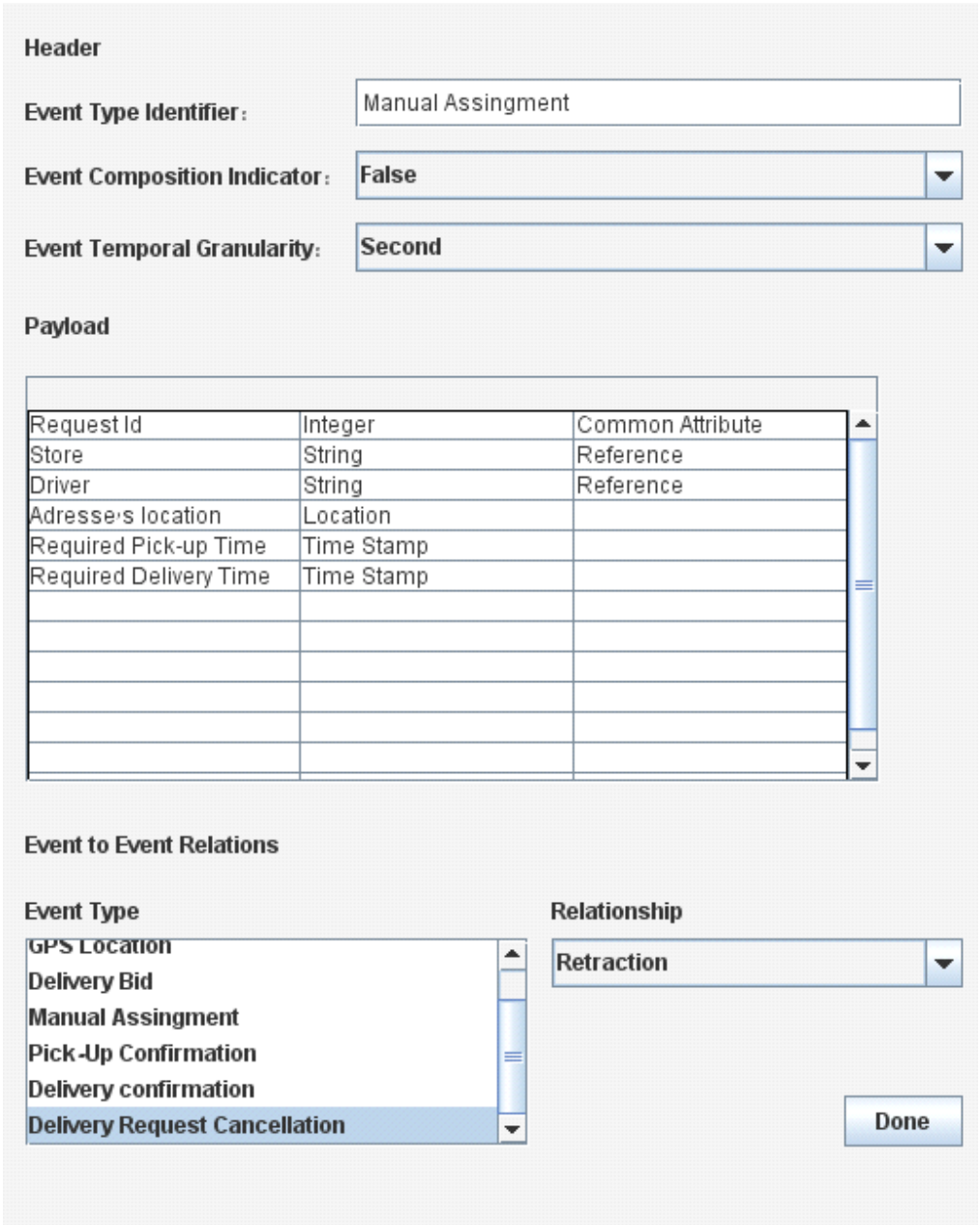


Figure 3.6 The definition elements for the Manual Assignment event type

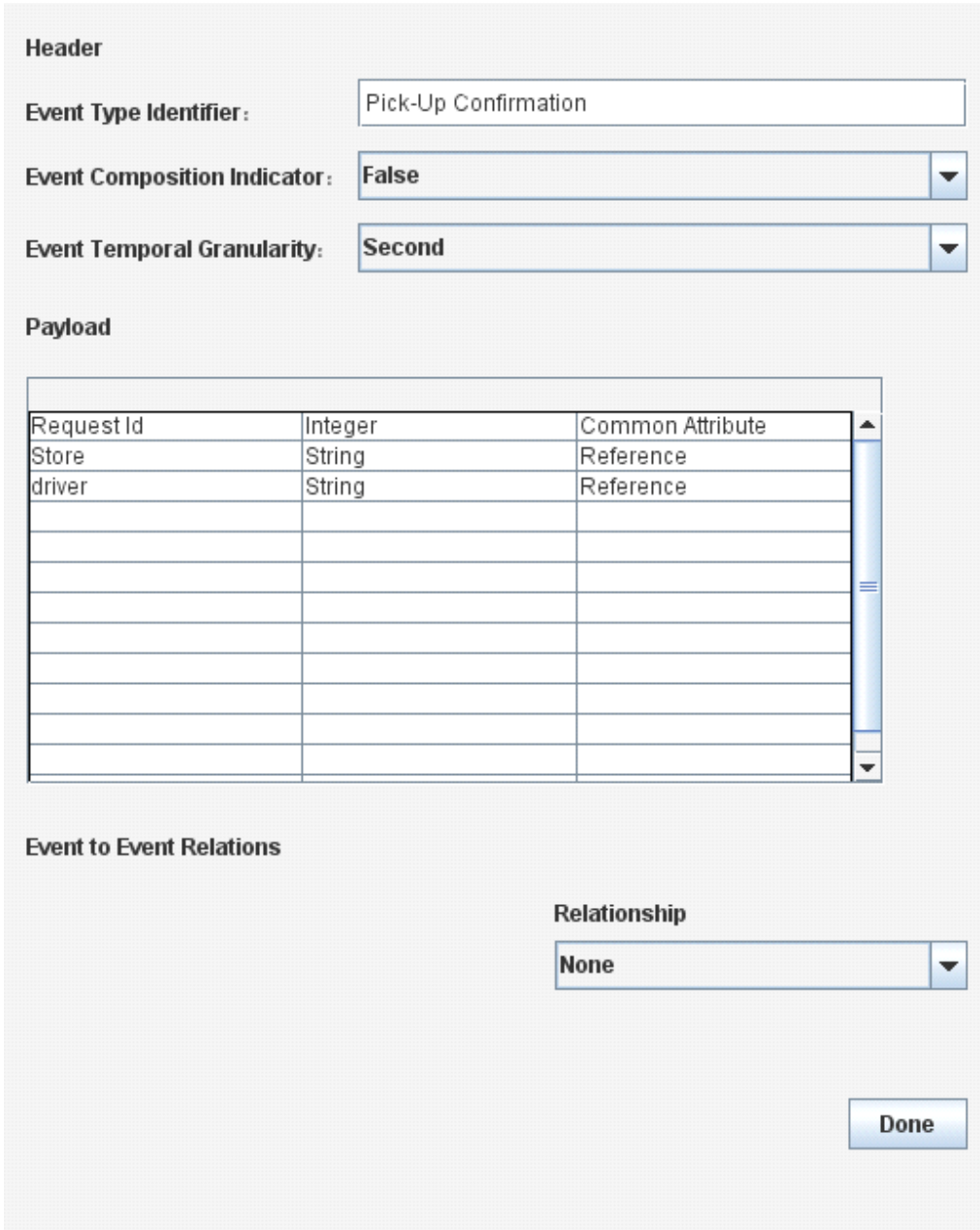


Figure 3.7 The definition elements for the Pick-up Confirmation event type

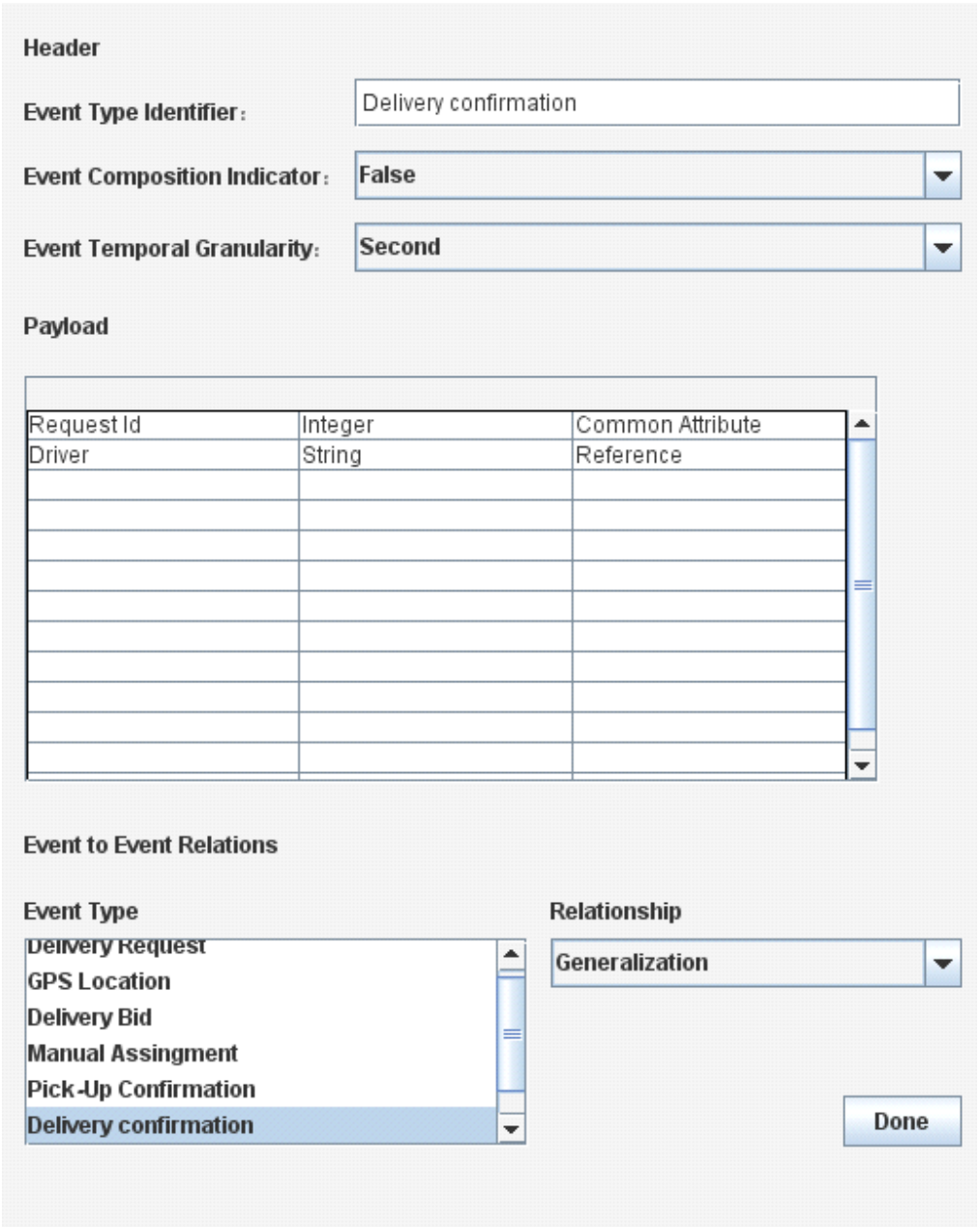


Figure 3.8 The definition element for the Delivery Confirmation event type

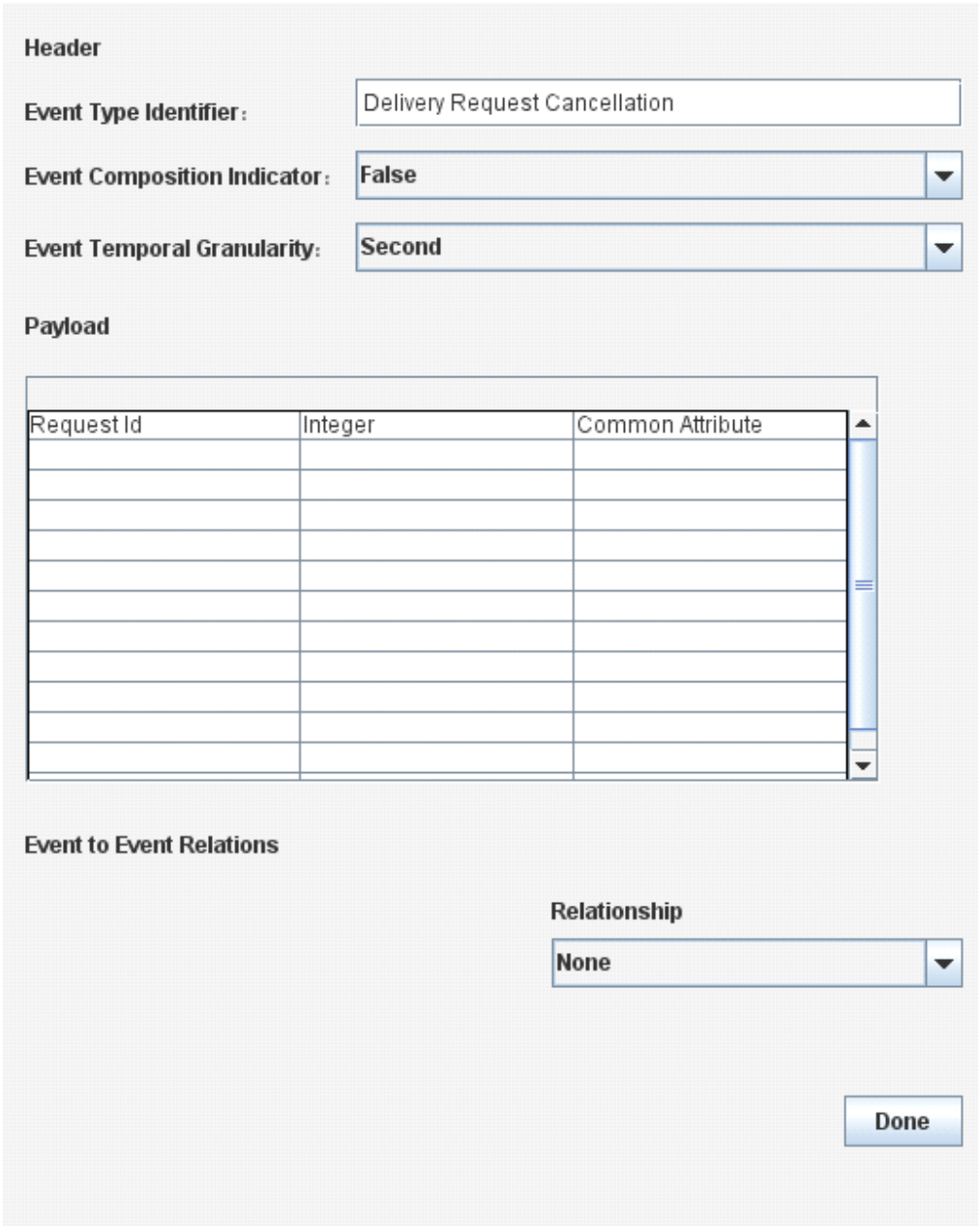


Figure 3.9 The definition element for the Delivery Request Cancellation event type

Some further comments about the various definitions:

- An event of the `Delivery Request` type is created when a customer's request is fed into the system. A `Bid Request` is a derived event that is considered as a direct cause of this event.
- Events of the GPS `Location` type are sent periodically to the system by the vehicle's GPS sensor.
- Events of the `Delivery Bid` type are emitted by the driver and contain in their payload information about the request id, store and the driver concerned.
- Events of the `Manual Assignment` type are emitted by the store, in cases where the store performs manual assignment (recall that some stores may prefer automatic assignment and this is done by an EPA creating a derived event).
- Events of the `Pick-Up Confirmation` type are emitted by the stores to confirm pick up by the driver, note that the header attribute Occurrence Time that exists by default in any of the events in this application records the time stamp in which the pick up has occurred. The Occurrence Time value is entered by the store.
- Events of type `Delivery Confirmation` are emitted by the addressee by signing on the driver's handheld device. This device also provides the value for the Occurrence Time time stamp.
- Events of the type `Delivery Request Cancellation` are used as retraction events to undo processing of a certain request

Listing 3.2 shows the information that might be carried in a run-time event instance. The run-time instance follows the definition of the attributes and other properties of the Delivery Request event type.

Listing 3.2 A particular event object of a Delivery Request event type

```
Event Type Identifier      Delivery Request
Event Identity             455244535
Occurrence Time            20 March 2009,15:15

Detection Time             20 March 2009,15:16
Event Source               Exotic Flowers store
Request-Id                 R429531
Addressee location         5 Main Street
Required Pick-up Time      15:30
Required Delivery Time     16:30
Store                      Exotic Flowers store
```

This listing shows an example `Delivery Request` event instance. The first four entries are header attributes, and show that this is an event type produced by the Exotic Flowers store at the indicated date and time. The Identity 45524435 is a unique value generated by the event processing system, to distinguish this event object from other event object instances.

The remaining attributes form the payload of the event. You can see the application-generated Request-Id attribute. This is the *common attribute* used to correlate this event with other events that relate to this same order.

In this example we have shown the information content of the event object in a simple tabular format. We conclude this chapter with a brief discussion of the various ways in which events are physically represented in practice.

3.6. Event representation in practice

There is, at present, no standard for the way that events or event types are represented. There are various structural representations that can be found in various products:

- Flat representation: similar to a normalized tuple in a database.
- Structure representation: similar to a record that may include arrays and tuples within the structure.
- XML representation: using XML schema to define the structure.
- Object: The event is represented as an object; methods have to be applied in order to retrieve its content.

Applications that receive events from multiple sources may have to be able to support multiple structures of various event types (it may even be the case that events of the same logical type are obtained from different sources in different physical representations). Some current products are able to accept multiple formats.

Next, we present some examples of how event types are defined in various existing languages, starting with figure 3.9, which is an example from the Apama language.

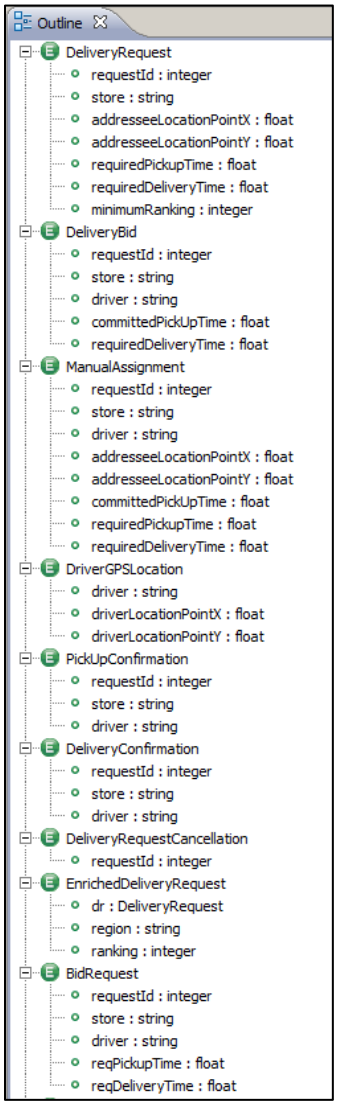


Figure 3.9 This example, done in the Apama language, shows part of the FFD event definitions, define as a schema, and represented by Apama IDE[9]. It shows the event types and the payload's attributes and their data types.

---

[9] Chapter 10 discusses the various programming styles and user interfaces.

Next we'll look at an example from the Rulecore language in Listing 3.3.

Listing 3.3 Event type definition in XML

```
Event: BidRequest
<EventDef eventType="BidRequest">
  <Description>Event: BidRequest</Description>
  <Properties>
    <Property name="RequestId">
      <base:XPath>string(base:EventBody/user:RequestId)</base:XPath>
    </Property>
    <Property name="Store">
      <base:XPath>string(base:EventBody/user:Store)</base:XPath>
    </Property>
    <Property name="Driver">
      <base:XPath>string(base:EventBody/user:Driver)</base:XPath>
    </Property>
    <Property name="AddresseeLocation">

<base:XPath>string(base:EventBody/user:AddresseeLocation)</base:XPath>
   </Property>
    <Property name="RequiredPickupTime">

<base:XPath>string(base:EventBody/user:RequiredPickupTime)</base:XPath>
    </Property>
    <Property name="RequiredDeliveryTime">

<base:XPath>string(base:EventBody/user:RequiredDeliveryTime)</base:XPath>
    </Property>
  </Properties>
</EventDef>
```

This example, taken from the Rulecore language shows an example of event type definition in XML, the payload attributes are defined as properties.

Finally, figure 3.10 is an example from the Streambase language.

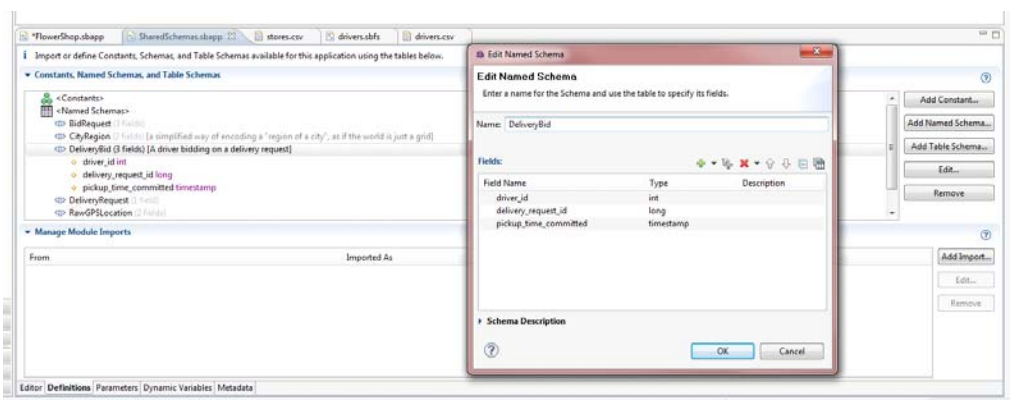


Figure 3.10 This example, written in Streambase is another example of event type definition, this is done by filling a form that is part of graphical user interface.

These three example languages can be accessed through the book's website.

3.7. Summary

In this chapter, the first of our"deep dives", we discussed what is meant by an event type, and described our platform-independent definition of an event type. There is, at present, no agreed-upon standard for how events or event types are to be represented, so in this chapter we focused on the information that needs to be carried by events. This is partitioned into the payload that carries the event's own information, and a header that provides additional meta-information about the event, which is not part of its content. We also listed the raw event types used in the Fast Flower Delivery application

In chapter 5 and the chapters that immediately follow it we will move on to look at event producers and the other entities that make up an event processing network,

Additional Reading

Antony Galton, Juan Carlos Augusto: Two Approaches to Event Definition. DEXA 2002: 547-556 http://www.springerlink.com/content/46mbb6ajt6t20qvd/

This article discusses various approaches for event definition

Christian S. Jensen, Curtis E. Dyreson, Michael H. Böhlen, James Clifford, Ramez Elmasri, Shashi K. Gadia, Fabio Grandi, Patrick J. Hayes,Sushil Jajodia, Wolfgang Käfer, Nick Kline, Nikos A. Lorentzos, Yannis G. Mitsopoulos, Angelo Montanari, Daniel A. Nonen, Elisa Peressi,Barbara Pernici, John F. Roddick, Nandlal L. Sarda, Maria Rita Scalas, Arie Segev, Richard T. Snodgrass, Michael D. Soo, Abdullah Uz Tansel, Paolo Tiberio, Gio Wiederhold: The Consensus Glossary of Temporal Database Concepts - February 1998 Version. In: Opher Etzion, Sushil Jajodia and Suryanarayana Sripada: Temporal Databases Research and Practice, Springer, 1998: 367-405

http://www.springerlink.com/content/03981447077588rj/

Temporal concepts such as: chronon, and other temporal concepts are taken from this temporal databases concepts' glossary. Note that the entire book contains various article on temporal databases

Exercises

3.1. Add three more event types to the Fast Flower Delivery example and write down their definition elements

3.2. What are the cases in which the use of occurrence time is important? Provide an example.

3.3 What are the cases in which the use of spatial properties is important? Provide an example.

3.4. Provide scenarios (not from the Fast Flower Delivery example) that use event generalization and retraction events.

3.5. Create two event instance examples, in the manner of listing 3.6, for each of the raw event types defined in figures 3.3 – 3.9.

3.6. Find documentation of two different event processing commercial products, study the way event types are defined, and translate the event types defined in figures 3.3 - 3.9 to these representations; then create the instance examples you defined in exercise 3.5.

4

Producing the events

"We've heard that a million monkeys at a million keyboards could produce the complete works of Shakespeare; now, thanks to the Internet, we know that is not true."

- Robert Wilensky

We continue our deep dive into constructing event driven applications by discussing the first point in the event flow, the event producer. In particular we look at:

- The notion of an event producer, its role and the definition element describing it
- The different types of event producer
- The different types of interaction with an event producer
- We also include a specification of the event producers used in the Fast Flower Delivery use case.
- We'll start by defining the event producer.

4.1 Event producer: concept and definition element

In Chapter 2 we discussed the decoupling principle which results in the separation between entities that emit events (event producers), the software artifacts that process the events (event processing agents, or EPAs) and the entities that consume the events created by these EPAs. An event producer introduces event objects into the event processing network from the world outside. From a structural point of view, it is represented in the event processing network as a node that has only output terminals. This is one of the differences between an event producer and an EPA, the other being that an EPA's logic is explicitly specified as part of the EPN definition, whereas the logic of an event producer lies outside the scope of the EPN. We make this second distinction in order to set clear borders between

the event processing that takes place in the event processing network and the pre- or post-processing that takes place outside it.

When talking about an event producer, it is important to recognize that there are three aspects to the concept. These are:

- An abstract type of event producer, for example `GPS Sensor`.
- A collection of producer instances, all of the same type, which feature in a given application. We refer to such a collection as a *class;* as an example we have the class of all GPS Sensors deployed in our Fast Flowers Delivery application.
- An actual instance of a producer, for example the GPS sensor inside driver John Galt's vehicle.

It is often helpful to be able to refer to a class of instances rather than to have to refer to each producer instance individually. Some applications can involve hundreds or even thousands of producers. Moreover, in many applications the number of producers is not fixed but varies over time; for example the number of drivers involved in our Fast Flowers application can vary day by day.

So an event producer node in the EPN can represent either a single producer instance connected to the EPN, or a class of producer instances all connected to the EPN. The producer node is a kind of proxy for all these producer(s) and their connection, and as such it describes the type of the connected producer(s), by specifying the event types that they emit through their output terminals.

4.1.1. The event producer definition elements

In the previous section we observed that there are three aspects to the event producer concept, and we use the event producer definition element to model all three. We use it to define abstract event producer types, event producer classes, and event producer instances.

The definition element consists of three parts: producer details, output terminals and relationships to other producers. A producer definition element that is being used to represent a class or instance will have one or more of its output terminals connected to other entities in the EPN (we will talk more about these connections in chapter 6). A definition element used to represent an abstract type of event producer cannot have its output terminals connected to anywhere.

Figure 4.1 illustrates the event producer definition element.

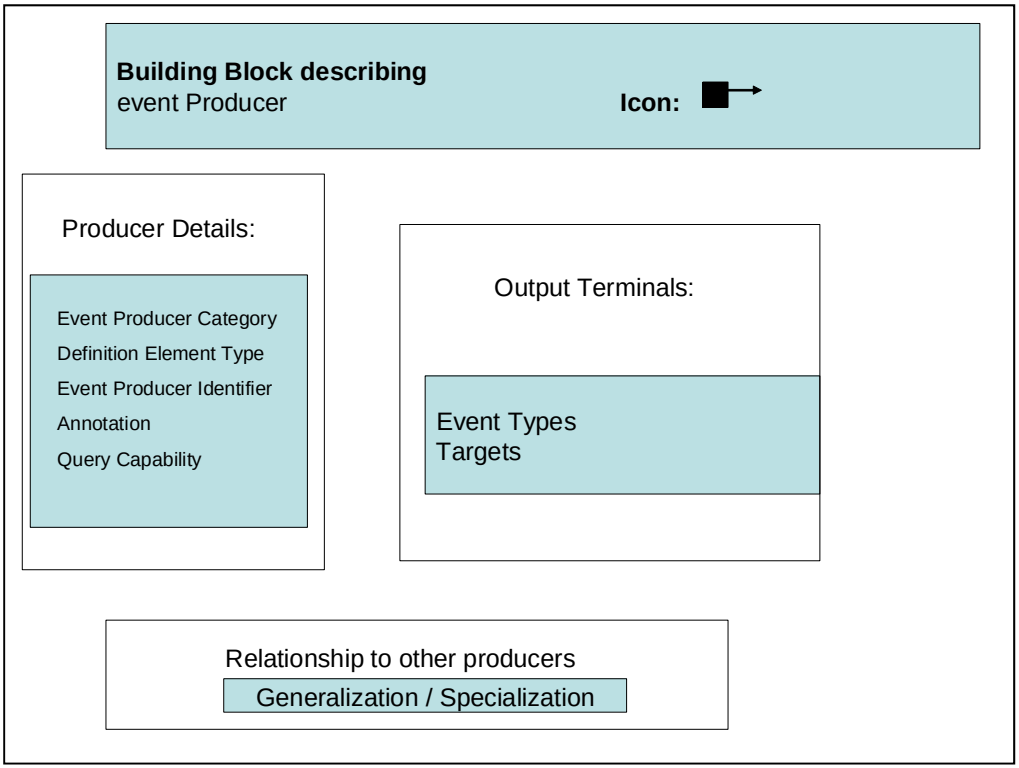


Figure 4.1 The event producer definition element contains details about the definition element itself, a list of output terminals, and a list of relationships to other event producer definitions

We now describe the three parts of the definition element shown in figure 4.1: producer details, output terminals and relationships to other producers.

4.1.2 Event producer details

As we remarked earlier, the event producer definition element is concerned only with the interaction between the producers that it represents and the rest of the EPN, and does not describe the internal logic of the event producers.

The details part contains some attributes that describe the producers represented by the definition element, along with an attribute that indicates whether the definition element represents an instance, class or abstract type.

- *Event producer category* indicates the kind of event producer that is being described by the definition element (for example it could be a software trace point, or an RFID reader); see section 4.2 for discussion of the different kinds of event producer.
- *Event producer identifier* gives the name of the producer type, class or instance described

by the definition element. It can be used when referring to this definition element from elsewhere.

- *Definition Element type* is an attribute which indicates whether the definition element represents an abstract type, class of producer instances or a single instance, its possible values are: abstract type, producer class, producer instance. When the definition element is being used as a node in an EPN this attribute will have one of the two values `consumer class` or `consumer instance`.
- *Annotation* is an optional annotation that provides more information about the event producer instance, class or abstract type.
- *Query Capability* is a Boolean attribute which indicates whether the producer can be queried.

The second part of the definition element describes the event producer's output terminals. Each output terminal has one or more event types associated with it and it also has a number of *targets* - references to entities that receive events that are emitted through the terminal. There are no targets on the output terminals in an abstract type definition element; one or more targets are added when a definition element is used to describe a class or instance, in other words when it represents a node in an EPN.

4.1.3. Output terminal details

An event producer emits events using one of its output terminals. There can be one or more output terminals and each output terminal has the following attributes:

- *Event types*: A collection of event type identifiers showing the types of events that can be emitted through this output terminal. An output terminal can have one or more event types associated with it. This association is not exclusive (an event producer can have the same event type associated with multiple output terminals)
- *Targets*: The identifier of the channel or other EPN which serves as a sink of the output terminal. Each output terminal can have zero or more targets. If the definition element represents an abstract producer type then none of its output terminals have targets assigned to them. We will have a full discussion of output terminals and targets in the next chapter; in the meantime Figure 4.2 shows an example of an output terminal with a single channel as its target.

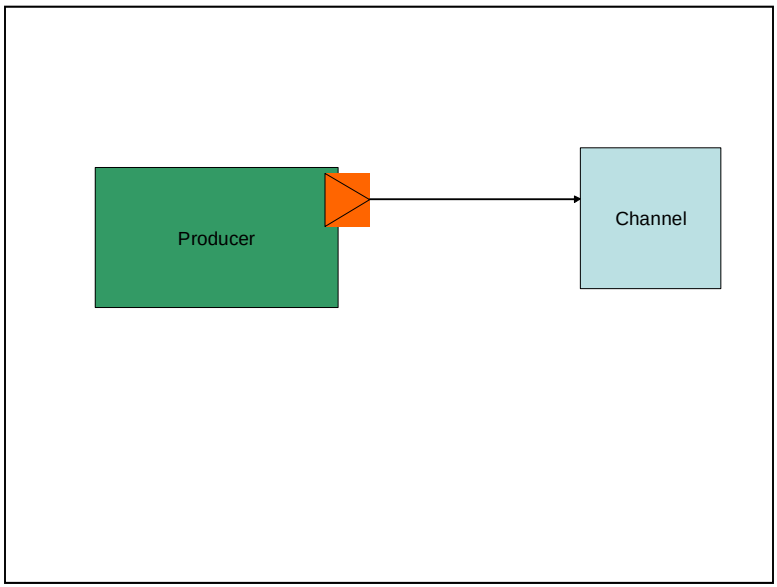


Figure 4.2 Illustration of the EPN edge created by setting a Channel as the target of a producer's output terminal

4.1.4. Producer relationships

A definition element that represents a class, instance or type can be a specialization of another definition element that represents a type, and conversely a type definition element can be a generalization of another type, class or instance.

In order to qualify as a specialization the definition element must include all the output terminals of the definition element that it specializes, although it may contain additional output terminals.

For example, a `GPS sensor` producer type might be a specialization of a `sensor` producer type, or the `store` type might be a generalization of the `flower store` type.

The specialization relationship can be used as a way of indicating the abstract type definition element that corresponds to a particular instance or class. This is useful if the application involves several classes of producer that all have the same abstract type (if it only has one class per type, then you might well choose not to model the type as a separate definition element).

Next we discuss the different kinds of producer.

4.2. The various different kinds of event producer

As we have already said, the event producer definition elements that appear in an EPN description only attempt to capture the EPN-facing interfaces of the event producers that they represent. The nature of an event producer, and its inner workings, are opaque as far as the EPN definition is concerned. All you see in the EPN model are proxies that represent the real producers. However these real producers do make up part an important part of the event processing application, and so it is worth us taking some time to discuss them.

We will do this by classifying them into categories and giving some examples of each. Figure 4.3 shows our three categories: hardware, software and human interaction.

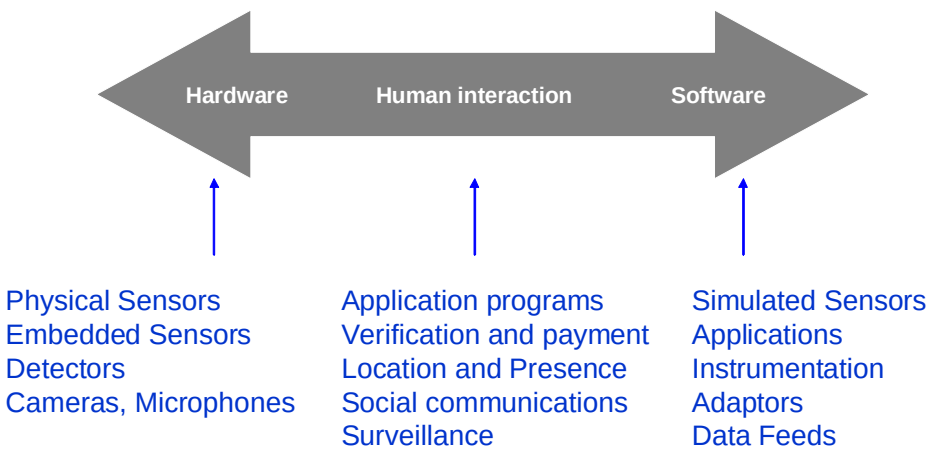


Figure 4.3 Some examples of the different kinds of event producer encountered in event processing applications

Before we go any further, it's worth saying that any classification is bound to be rather subjective – in reality a producer may well involve a mixture of hardware, software and human elements. Also there's sometimes a choice about where you set the boundary of an event producer. When talking in the abstract it can be difficult to be precise about where the event producer stops and the rest of the event processing network begins, but in practice this is rarely a problem when building an application.

4.2.1 Hardware event producers

Hardware producers are used extensively in a number of application areas, in particular:

- Medical equipment and personal body sensors (for example heart-rate monitors)
- Device management (computer systems or industrial equipment)
- Defense and military applications
- Security applications
- Traffic management systems
- Logistics and supply chain management
- Weather reporting and forecasting

The archetypal hardware producer is a sensor that generates events that report on one or more aspects of the physical environment in which it is situated, for example a smoke detector.

A sensor can be packaged as a discrete piece of hardware, such as the smoke detector example we just gave, or it can be built into another piece of equipment, for example a sensor that detects the fan speed on a computer motherboard. The simplest kind of sensor reports on just a single aspect of its environment. The kinds of condition that such a sensor can detect include:

- motion of the sensor itself including vibration
- tilt or angle of orientation of the sensor
- rotation of a rod attached to the sensor
- temperature
- humidity
- light (intensity or color)
- infra-red or radio waves
- sound (intensity of frequency)
- air (or other gas) pressure
- physical pressure applied to the sensor itself
- magnetic field
- level or pressure of a liquid
- airflow
- electric current or potential
- electrical conductivity
- chemical environment (for example pH level, or presence of a particular chemical)

- mechanical strain
- ionizing radiation

There are more sophisticated detectors that use one or more of the physical detection mechanisms listed above to make specific observations or look out for particular kinds of occurrence. Here is a small list of examples:

- A sensor that detects motion external to itself, such as a Passive Infra Red (PIR) sensor. This can be used to detect the presence of people in the vicinity of the sensor, for example when reporting on room occupancy or when looking for intruders.
- A sensor that detects whether the door on the casing of some equipment is open or closed.
- An RFID reader used to detect the presence of an RFID tag. This can be used in supply chain or many other application areas.
- A seismometer used to detect and report on earthquakes or nuclear tests
- A Traffic speed detector. Apart from their obvious use in penalizing speeding drivers, speed detectors can also be used in intelligent traffic management systems
- GPS Location detectors. These are used in a wide variety of tracking and location-aware services.

Cameras (both still and video), microphones, telephones and radio receivers can also be viewed as event producers since the data that they produce can be processed by event processing applications. For example a security application could process frames coming from a video camera looking for the presence of unauthorized personnel in a secure area.

4.2.2 Software event producers

While there is often some software associated with a hardware producer, there are some event producers that exist entirely of software.

Our first category consists of *Simulated Sensors*. These are software simulations of the kinds of hardware producer that we talked about in the previous section. Simulated sensors are used when the entire external system is itself a simulation, for example a flight training simulator or virtual reality game, and they can also be used to stand in for a real piece of hardware when testing an event processing application.

An Event producer could be a first-class part of a software *application*. By this we mean that it is a piece of application logic that explicitly generates an event object and submits it to the event processing network. This may happen as a result of a human interaction (see the next section) but there are cases where events can be generated less directly. For example in a financial trading system there might be a settlement application that automatically generates payment events. The application uses some kind of programming interface to submit the event, and we will discuss that in section 4.3.

Events can be produced indirectly by a technique known as *instrumentation*. Here the events are not generated by application code itself, but instead are produced by software that is monitoring the application, looking for noteworthy activity[1]. There's a wide range of things that could be viewed as noteworthy, ranging from tracing program start-up and shut-down or tracing function calls within the application, through detecting updates that the application makes to its data, reporting computer performance statistics, or spotting and reporting on hardware or software errors if they occur. Instrumentation can be provided by the operating system or container that runs the application, or by database or messaging middleware that the application uses. Examples include workflow engines, which can generate events when a particular workflow goes through a state transition, security subsystems which can generate alerting events when they detect an attempted security violation and message queuing systems which can generate events when queues go above a certain depth.

We use the term *adaptor* to refer to a producer which doesn't directly detect events itself, but instead collects information available from elsewhere and uses that to generate events. Adapters can be used to provide the instrumentation we mentioned in the previous paragraph, for example adapters that run against application or database log files, but they are also used to connect hardware sensors to an event processing network. A single adaptor can be used to connect multiple sensors to a network, and so act as concentrating gateways, as well as translating from the protocol used by the sensor into the protocol used by the event processing network. It is sometimes convenient to view the adaptor as the event producer, rather than having each sensor appear as a separate producer.

Our final category of software producer is a News or Data *Feed*. This is a mechanism that brings data in from outside the organization that owns the event processing network. For many years there have been financial trading applications that use Stock Feeds provided by News Agencies or individual exchanges. These feeds contain price information about recent trades of stocks or other financial instruments, as well as variety of other pieces of financial information. More recently we have seen the emergence of web feeds. There are now literally tens of thousands of these available on the Internet, using ATOM or RSS feed formats.

4.2.3 Human interaction

Some events are generated directly by human interaction, albeit with a bit of software and hardware assistance. Two of the three producers in our Fast Flowers example are of this kind: there's the Store producer, where an employee of the store enters or cancels requests, picks drivers and confirms pick-up, and the driver's handheld delivery confirmation device.

Human interaction can be facilitated by an *application program* with a user interface that in effect allows the user to enter the event. In our example we could imagine that each store

[1] This kind of producer is sometimes called a *monitor* or *probe*

has a web application, and store personnel use a form-style interface to place a delivery order request.

Events can also be generated using a *verification* or *payment* device, for example the delivery confirmation produced by the driver's handheld device in our application, or a purchase event being generated by a till in a retail store.

There are also producers that detect our *presence*, for example when swiping an identity card (or having a personal RFID tag scanned) when entering a secure area, using an NFC (near field communications) tag to go through a ticket barrier on public transport, or passing through immigration control. Instant messaging applications (and increasingly telephony applications) can produce presence events that indicate when a user has turned a computer or handheld device on or off, and these presence events sometimes also include information about the user's location or describe what they are doing.

This brings us to our next area, *Social Communications*. As well as providing its regular web browser interface, the popular Twitter internet service offers RSS feeds that can be used to communicate presence (and other) events. Events can also be produced from other social networking applications (recall the final example we gave in section 1.1.2)

Our final category is *surveillance*. We are all familiar with CCTV cameras, but a more controversial area is web activity monitoring, where user's website interactions are captured and tracked.

4.3. Interfacing with an event producer

Having looked at the wide range of possible producers, we now turn our attention to how they interface with the rest of the event processing network. We start by looking at the interaction patterns that they use, and then we will look at the mechanics of how they connect.

4.3.1 Interaction patterns

A quick examination of the producers reviewed in section 4.2 shows that they fall into two further categories.

- Producers that sense environment or state, for example temperature sensors
- Producers that detect specific occurrences and report on them via events. These occurrences can be rare (for example an intruder alarm) or frequent (a human heartbeat, or a customer walking through a turnstile), but they nevertheless identifiable as separate occurrences

In the second case, it's clear that the event objects generated by the producer describe the occurrences that it detects, but what about the first category? What kind of event objects do they produce, and more important when should they produce them? A purist might claim that there are no real occurrences and that therefore they should not be counted as event producers at all. That would however be to deny the fact that they are used in event

processing applications, and that the event objects that they emit can be treated in the same was as the event objects produced by the detector style of producer.

The event objects generated by a sensor-type producer usually record the current value of whatever it is that the sensor is measuring. There are three strategies that a sensor-type producer can adopt as to when it emits the event. The first of these is to emit an event only when it detects an appreciable change in the value of what it is measuring, for example a temperature sensor might generate an event if the temperature has moved by more than 0.5 °C from the value that it last reported. This approach is closest to that of an event detector, since it is in effect reporting the "temperature has changed" occurrence. The second strategy is simply to report the current value, whatever it is, on a periodic basis, and the third approach is for it never to emit any event spontaneously, but just to record the current value and return it to anyone who asks. This third approach is sometimes referred to as "polling", and is similar to the pull style of event distribution that we discussed in chapter 2.

The same choices are available to the detector-style of event producer. It can implement a push-style interface, which means that it emits an event object each time it detects the corresponding occurrence, it can hold onto the event and on a periodic basis emit an object corresponding to the last event detected (this approach is rarely used in practice) or it can wait until polled before returning an event object describing the last occurrence (or in some cases occurrences) that it detected.

The choice of approach is usually made by the designer of the producer, and is influenced by considerations such as the nature of the events themselves, as well as the capabilities of the producer implementation (for example it would be difficult to implement the polling approach if the producer has no mechanism for receiving incoming requests). In general a push approach is used if low latency is important, or if every event is important. A polling approach can be used if only the most up to date value is important (in other words if the latest event generated by the producer makes earlier events obsolete). We discussed other reasons for using a pull approach in section 2.3.

4.3.2 Queriable event producers

Many of the event producers we discussed in section 4.2 simply forget about an event occurrence once they have emitted the corresponding event object[2]. However there are some producers that retain a history of events that have occurred. This could be for purposes of non-repudiation or other legal or regulatory reasons or to assist with subsequent problem determination. Also there are some software producers (for example log or database adapters or feed producers) which continue to have access to the raw data that they use to produce events.

[2] They might of course have to hold on to an event object for a while before emitting it, particularly if they support a pull interface, but in this section we are concerned with more open-ended longer term retention of events

Producers that do retain historical event data in this way can provide an interface to allow this data to be queried by an event processing application, and we use the definition element's *Query Capability* to attribute to indicate if this is the case or not. One area in which the ability to query past events is useful is the area of *Active Diagnostics* that we mentioned in chapter 1, since events which seem insignificant at the time when they occur can prove important when diagnosing a problem that is detected later. If the producer of such events is queriable then it means that an active diagnostics application does not have to maintain its own store of historical events.

4.3.3 Interfacing mechanisms

An event producer interacts with the event processing network using an event distribution mechanism (we introduced event distribution in chapter 2). An event processing network can support one or more event distribution mechanisms, and these mechanisms can either be protocol based or API based.

As it name suggests, in a protocol based mechanism the producer uses a transport protocol supported by the EPN implementation. This could be a proprietary protocol or a standardized one, and describes the way that event objects are serialized as well as the transport protocol used to transmit them. It can be support either push-based or pull-based distribution (as discussed in the previous section) or both. The protocol might also include a way for the event distribution network to provide a filter to exclude certain events. To see why this might be useful, consider an event processing application that is monitoring a piece of equipment and needs to know when its temperature exceeds 40 °C. It could certainly do this by taking every temperature change event from the sensor and then filtering out all temperature change events unless they go above this temperature, but it can reduce the amount of events being produced, distributed and processed, if it is able to delegate the task of doing the filtering to the producer itself[3].

In the API approach, the EPN provides some kind of programming interface for the producer to use, and this approach is frequently used by software producers. The programming interface shields the producer from the underlying protocol that is used by the middleware. There are two main styles of API. One is the kind of programming interface provided by Message Oriented Middleware, such as the Java Message Service API, where the producer explicitly constructs an event object and then uses the API to transmit it. In the other style of API (sometimes found in producers used in management applications) the API provides access to one or more resources (objects recording aspects of the current state of the producer). Rather than having to construct an event object explicitly the producer simply

[3] There is another option, which is to program the temperature threshold value (40 °C) into the design of the sensor, but the dynamic approach of having this value supplied by the EPN itself allows the value to be changed easily if required

makes updates to the state held in these resources, and these updates in turn cause resource state change event objects to be created and distributed.

Now we have discussed the theory, let's look to see how these ideas are put into practice in our Fast Flower Delivery application.

4.4. Producers in the Fast Flower Delivery example

In the Fast Flower Delivery example there are three different producer types: Store, Driver and Vehicle. Listing 4.1 shows the definition of the different types:

Listing 4.1 Producer type definitions

Definition element of Producer type Store

Producer Category	Definition Element Type	Producer Identifier	Output Terminal	Event Type	Targets
Human	Producer Class	Store	Send Delivery Request	Delivery Request	Delivery Request Channel
			Report Manual Assignment	Manual Assignment	Assignment Channel
			Confirm Pick-up	Pick-up Confirmation	Pick-up confirmation channel
			Request Cancellation	Delivery Request cancellation	Delivery Cancellation channel

Note that in our example each of the stores' output terminals is wired to a separate channel. We have chosen to use separate channels for each event type, with the exception of the Assignment channel which is used for both Manual and Automatic assignments,

Definition element of Producer type GPS Sensor

Producer Category	Definition Element Type	Producer Identifier	Output Terminal	Event Type	Targets
Sensor	Abstract Type	GPS Sensor			

Definition element of Producer type Vehicle

Producer Category	Definition Element Type	Producer Identifier	Output Terminal	Event Type	Targets
GPS Sensor	Producer Class	Vehicle	Report Location	GPS Location	GPS Channel

This producer belongs to the category of GPS sensor, which has been defined as an abstract producer type in the previous definition element.

Definition element of Producer type Driver

Producer Category	Definition Element Type	Producer Identifier	Output Terminal	Event Type	Targets
Human (via handheld device)	Producer Class	Driver	Bid for Delivery	Delivery Bid	Delivery Bid Channel
			Confirm Delivery	Delivery Confirmation	Delivery Confirmation Channel

Delivery confirmation is produced by the Driver's handheld device, but requires signature of the delivery recipient

We have three concrete types of producer in this application: the participating stores, the vehicles and the drivers themselves (represented by their hand-held devices). We represent each of these using class-style definition elements. We can do this because, for each of the three types, all the producers of that type are treated similarly – for example each store has two output terminals and these terminals are connected to the same targets, regardless of which store it is. There is one abstract type definition (GPS Sensor) and the Vehicle producer class is a specialization of that type, reflecting the fact that there is a physical GPS sensor in each vehicle, and the event types are actually being produced by that sensor.

4.5 Summary

In this chapter we have introduced the first link in the event processing flow, the event producer. We have discussed the event producer definition element, and noted that it can be used to represent an abstract consumer type, a concrete event producer or a class of multiple concrete event producers. We also remarked that the event producer internal logic is outside the scope of the event processing network and that an event producer node in an EPN is a proxy that represents the connection from one or more real event producers to the rest of the event processing network. We have also discussed a number of different kinds of producer and looked at interactions between producers and the rest of the network.

The Event producer is the first EPN node that we have discussed, and from here we move on to the rest of the EPN picture, by looking at the event processing network itself, and the way that it is used to connect event processing entities to each other.

Additional reading

The Workflow Management Coalition Specification, Workflow Management Coalition, White Paper – Events

http://www.huihoo.org/jfox/jfoxflow/specification/06.WfMC_White_Paper_Events.pdf

This white paper describes workflow as producer and consumer of events

IBM Redbooks, Implementing event processing with Cics

http://www.amazon.com/Implementing-Event-Processing-Cics-Redbooks/dp/0738433365/ref=sr_1_1?ie=UTF8&s=books&qid=1258899442&sr=1-1

This book describes an example of software instrumentation to serve as producer for event processing.

Carlos De Morais Cordeiro, Dharma P. Agrawal: Ad Hoc & Sensor Networks: Theory And Applications, World Scientific Publishing company, 2006

http://www.amazon.com/Ad-Hoc-Sensor-Networks-Applications/dp/9812566821/ref=sr_1_3?ie=UTF8&s=books&qid=1258899679&sr=1-3

This book provides introduction to sensor networks

Exercises

4.1. Explain, with examples, why it is useful to have the concept of a class of event producers.

4.2. Provide an example of event producer class that contains multiple instances

4.3. We have stated that all the members of an event producer class must have the same type. Explain why we impose this requirement

4.4. Provide an example of an application using multiple event producer classes in which it is helpful to have an abstract producer type.

4.5. Provide three examples of useful instrumentation of programs, explain what events will be created in each case, and what purpose they may serve.

.

5

Consuming the events

"What information consumes is rather obvious: it consumes the attention of its recipients. Hence, a wealth of information creates a poverty of attention and a need to allocate that attention efficiently among the overabundance of information sources that might consume it."

- Herbert Simon

We now move on to look at the event consumer, which in many ways is the mirror image of the event producer that we discussed in the previous chapter. We therefore recommend that you read that chapter before reading this one.

In this chapter we look at:

- The notion of an event consumer, its role and the definition element that describes it
- The different types of event consumer
- The different types of interaction with an event consumer
- The distinction between a consumer and a subscriber
- We also include a specification of the event consumers used in the Fast Flower Delivery use case.
- We'll start by defining the event consumer, the core concept of this chapter.

5.1 Event consumer: concept and definition element

The event consumer is the logical complement of the event producer that we looked at in chapter 4. An event consumer accepts event objects from the event processing network and

processes them in some way (how it actually processes the event objects lies outside the scope of the EPN definition).

An event consumer is represented in an EPN as a node that only has input terminals. Just as is the case with the event producer an event consumer node in the EPN is a proxy, either for a single instance of a consumer connected to the EPN or for a class of consumer instances all connected to the EPN.

5.1.1. Event consumer definition elements

The event consumer definition element can be used in one of three ways:

- To define an abstract event consumer type
- To represent a class of concrete event consumer instances
- To represent a single concrete event consumer instance

The definition element consists of three parts: consumer details, input terminal specifications and relationships to other consumers. A consumer definition element that is being used to represent a class or instance will have one or more of its input terminals connected to other entities in the EPN (we will talk more about these connections in the next chapter); a definition element used to represent an abstract type of event consumer cannot have its input terminals connected to anything.

Figure 5.1 illustrates the event consumer definition element.

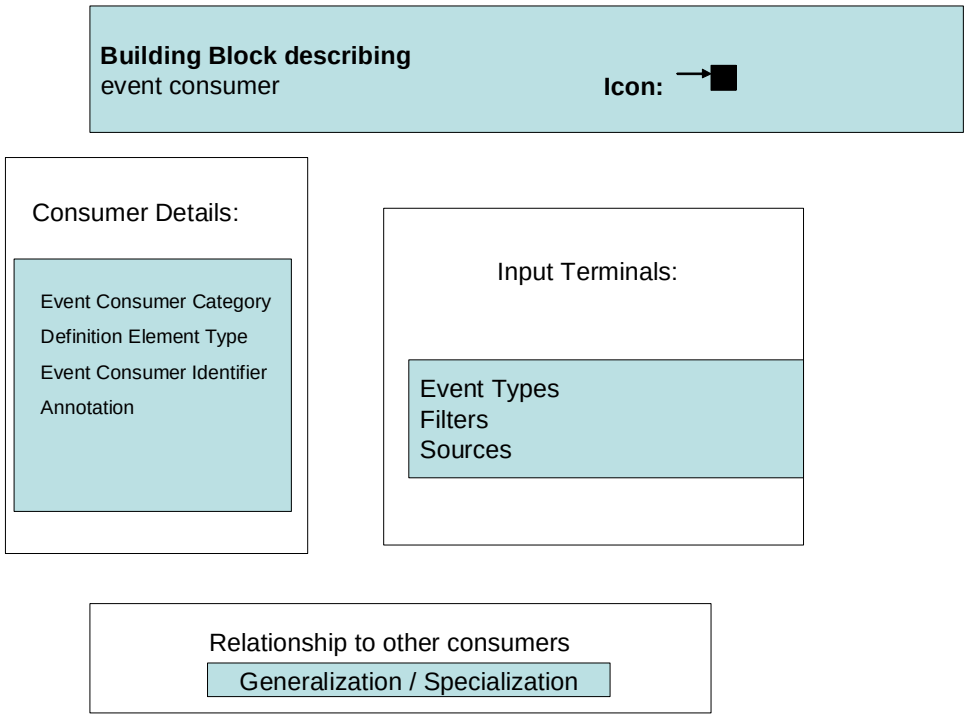


Figure 5.1 The event consumer definition element contains details about the definition element itself, a list of input terminals, and a list of relationships to other event consumer definitions

We now describe the three parts of the definition element shown in figure 5.1: consumer details, input terminals and relationships to other consumers.

5.1.2 Event consumer details

The event consumer definition element is concerned only with the interaction between the consumer or consumers that it represents and the rest of the EPN; it does not describe the internal logic of the event consumers.

The *consumer details* part of the definition element contains some attributes that describe the consumers represented by the definition element, along with an attribute that indicates whether the definition element represents an instance, class or abstract type.

- *Event consumer identifier* gives the name of the consumer type, class or instance described by the definition element. It can be used to refer to this definition element from elsewhere.
- *Definition Element type* is an attribute which indicates whether the definition element

represents an abstract type, class of consumer instances or a single instance; its possible values are: abstract type, consumer class, consumer instance. When the definition element is being used as a node in an EPN, this attribute will have one of the two values consumer class or consumer instance.

- *Event consumer category* indicates the kind of event consumer that is being described by the definition element, for example it could be a dashboard or a hardware actuator. This attribute is used for descriptive purposes and has no effect on the way that events are handled by the EPN itself. We give some suggestions for categories in section 5.2.
- *Annotation* is an optional annotation that provides more information about the event consumer instance, class or abstract type.

The second part of the definition element describes the event consumer's input terminals. Each input terminal has one or more event types associated with it and it also has a number of *sources* - references to entities from which the terminal is to receive events. There are no sources on the input terminals in an abstract type definition element; one or more sources can be added when a definition element is used as a node in an EPN, as then it is being used to describe a class or instance.

5.1.3. Input terminal details

An event producer can receive events through any of its input terminals. There can be one or more input terminals and each input terminal has the following attributes:

- *Event types*: A collection of event type identifiers showing the types of events that can be accepted by this input terminal. An input terminal can have one or more event types associated with it.
- *Sources*: Identifiers of channels or other EPN entities which can send events to the input terminal. Each input terminal can have zero or more sources. If the event consumer definition element represents an abstract consumer type then none of its input terminals have sources assigned to them. We will have a full discussion of input terminals and sources in the next chapter; in the meantime Figure 5.2 shows an example of an input terminal with a single channel as its source.

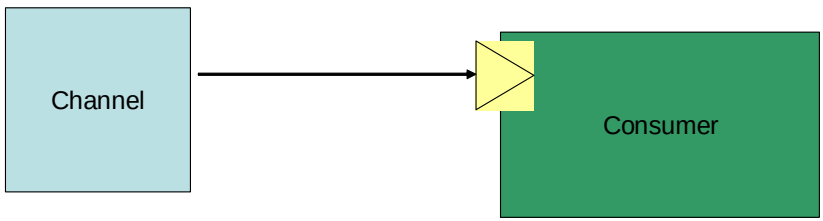


Figure 5.2 Illustration of the EPN edge created by setting a Channel as the source for a consumer's input terminal

As we will see in the next chapter, a Consumer can be associated with a channel (or other source of events) without specifying an explicit source on an input terminal. This is because the input terminal can itself be specified as a target on the output terminal of the source object.

5.1.4. Consumer relationships

A definition element that represents a class, instance or type can be a specialization of another definition element that represents a type, and conversely a type definition element can be a generalization of another type, class or instance.

In order to qualify as a specialization the definition element must include all the input terminals of the definition element that it specializes, although it may contain additional input terminals. As with event producers, the specialization relationship can be used as a way of indicating the abstract type definition element that corresponds to a particular instance or class.

Next we discuss the different kinds of consumers.

5.2. The various different kinds of event consumer

In the previous section we described the *Event Consumer Category* attribute. This is a free format text attribute that gives an indication of what kind of event consumer is being described. In this section we give some examples which could be used as categories.

As with Event Producers, we observe that there are three broad families of consumers: hardware, software and human interaction. These are shown in Figure 5.3.

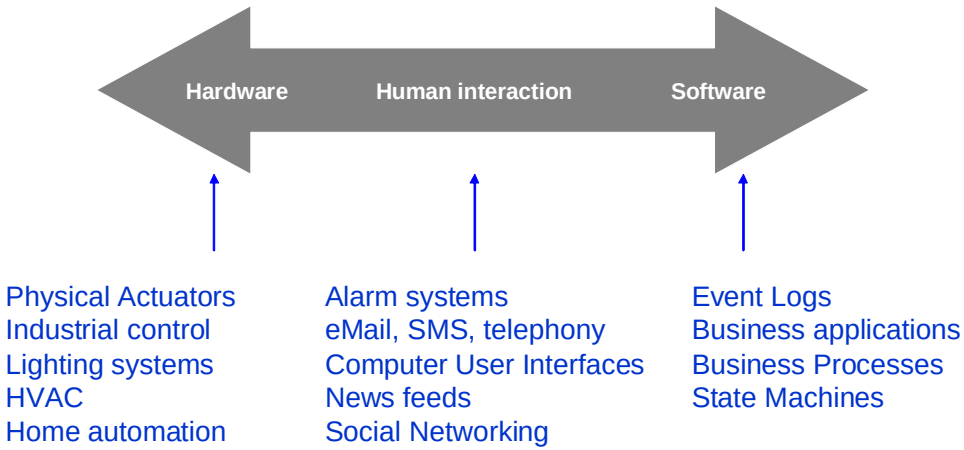


Figure 5.3 Some of the different kinds of event consumer encountered in event processing applications

As is the case with Event Producers there isn't a clear distinction between these classifications; you may well encounter consumers that involve a mixture of hardware, software and human interaction elements.

5.2.1 Hardware event consumers

A piece of hardware that consumes events is often referred to as an *actuator*, and is the counterpart of the hardware *sensor* that produces events. An actuator takes an incoming event and reacts to it by performing a physical action. This action, which is often used to control something in the physical world, might involve physical motion (if the actuator includes some kind of motor), changing a magnetic field, or producing an electrical or radio signal.

Here are some examples of actions that could be performed by an actuator:

- Locking or unlocking a door
- Raising or lowering a barrier

- Applying the brakes on a vehicle
- Opening or closing a valve
- Controlling a railroad switch
- Turning a piece of equipment on or off

An actuator could be physically packaged alongside a sensor in the same piece of hardware, but this is fairly unusual, and when this happens we still model the sensor and actuator pieces in an EPN as a separate producer and consumer.

Figure 5.3 lists some specific areas where actuators can be applied. For example in industrial control applications actuators are used to power equipment on and off, to control the operation of machinery and to control the flow of liquids. In intelligent building applications actuators can be used to control lighting, heating and air conditioning systems. An emerging area is home automation where, as well as controlling lighting and heating, actuators can be used to perform tasks such as opening or closing garage doors or window blinds.

5.2.2 Human interaction

The actuators described in the previous section react to events by directly controlling something in the physical world. There is another style of event consumer whose job is to interface with human beings. Depending on the nature of the event, this might involve alerting someone about a serious occurrence that needs immediate attention, or reporting something less urgent that nevertheless might be of interest, or updating the information displayed by a visual display of some sort.

People can be alerted to important and urgent events by alarm systems (for example bells or flashing lights) or by system-generated telephone calls. For less urgent alerts, you could use a consumer that generates an email or an SMS text message.

Consumers frequently convey information about events through the medium of a computer user interface. This interface is implemented either by a specially-written client application or, increasingly frequently, via a web application allowing the event data to be viewed on a web browser without the requiring the installation of specialist software.

The name *dashboard* is given to a particular kind of user interface that gathers together information on past and present events, often from multiple sources. Dashboards are used by applications that provide monitoring capability, for example performance monitoring or business activity monitoring tools. Figure 5.4 shows an example panel from a network performance tool. This uses a portal-like dashboard to show routine performance-related information.

Figure 5.4. A Performance monitoring dashboard. This event consumer takes a variety of performance measurement events and displays them as graphs.

As well as gathering raw data, monitoring systems also calculate *metrics* or *key performance indicators*. These are values computed from the raw data that have meaning to the business concerned. Dashboards can use graphical techniques to let people know when these metrics or indicators show that something is not behaving as expected. Figure 5.5 shows a number of network performance metrics displayed in a dashboard panel.

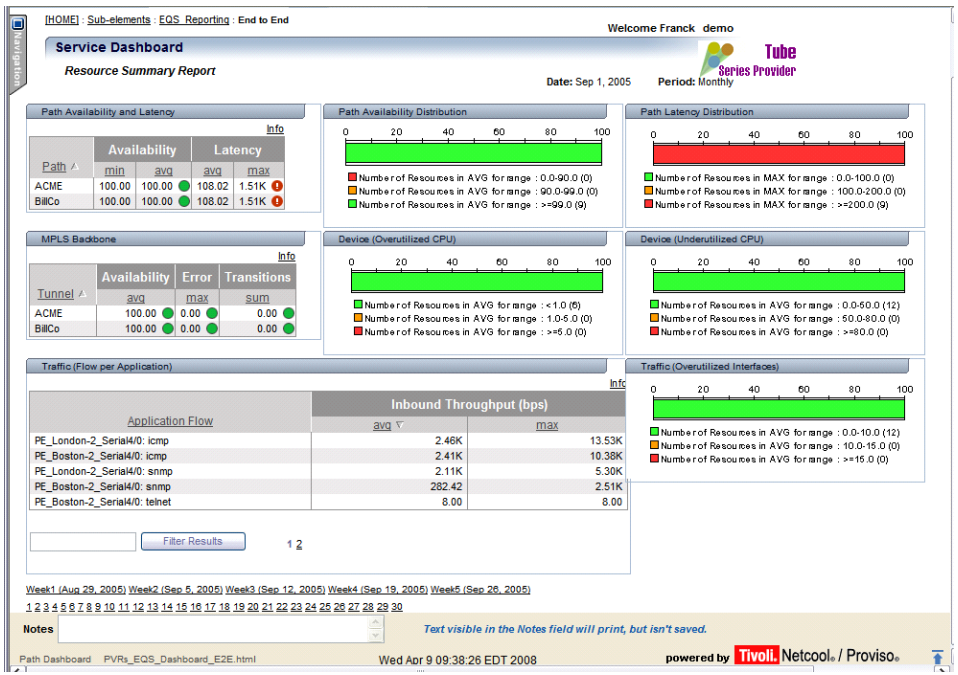


Figure 5.5 A dashboard screen showing the use of color-coding to indicate out-of-line situations. The window at the top right indicates that all of the resources being monitored exceeded the maximum acceptable value for path latency

In this example raw performance events from a variety of network resources have been analyzed and a number of metrics have been computed and averaged over the previous month (for example CPU utilization, traffic, and path latency). These averaged metrics have then been compared against pre-defined value ranges, and are displayed as red, orange or green bars.

A busy executive can get a quick summary of the dashboard using a device such as the Ambient Orb from the Ambient Devices company[1], shown in figure 5.6. This can be programmed to change color in response to data from the Event Processing application, and so can be used to show the status of a particular metric or collection of metrics.

[1] http://www.ambientdevices.com/cat/orb/orborder.html

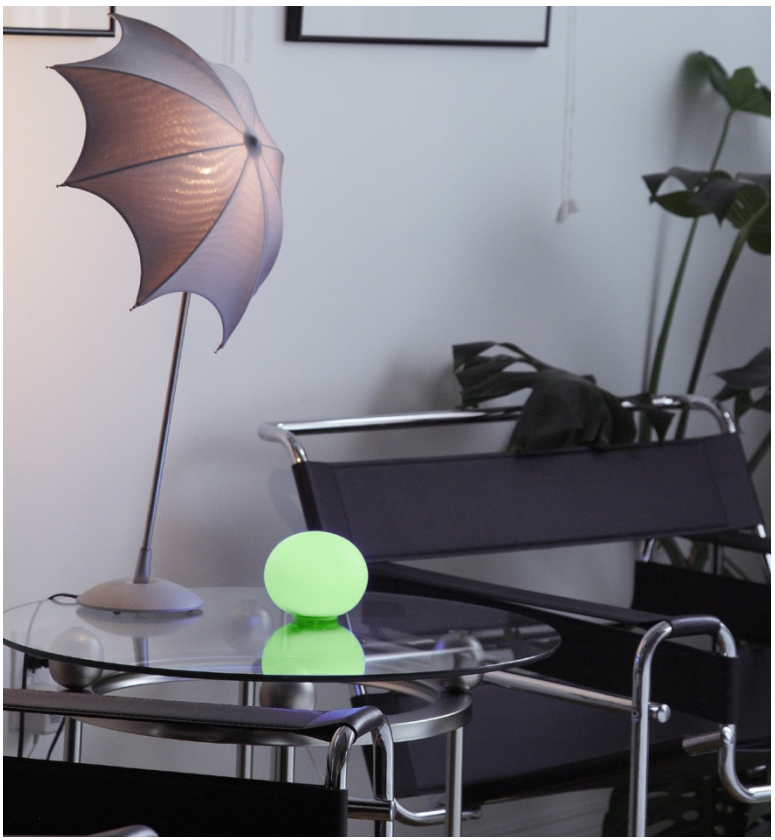


Figure 5.6. The Ambient Orb. The orb can be programmed to change color to reflect the general health of an organization, or to show the status of a specific key indicator.

Event processing applications can use consumers specifically designed to display event data in a way that is appropriate to the type of event in question (this is sometimes referred to as “event visualization”). In figure 5.7 we can see a consumer that provides the visual display part of a vehicle tracking application.

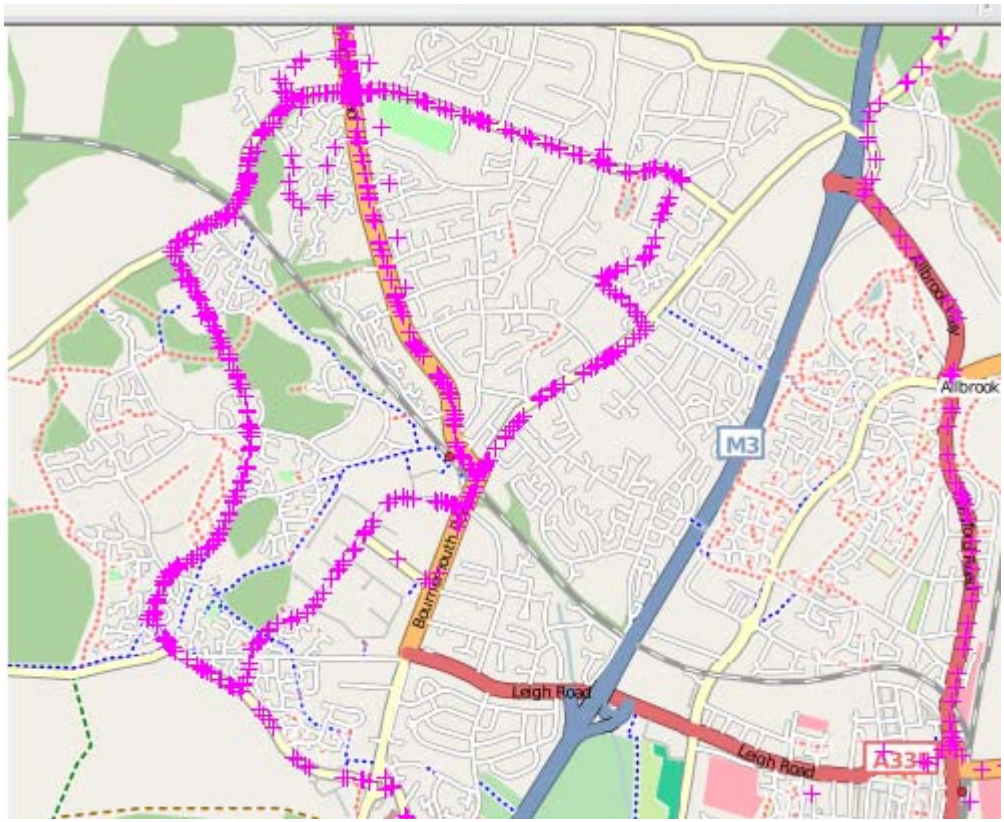


Figure 5.7 A visual display from a location tracking application. The individual markers show the positions that a particular vehicle (in this case a bus) has visited in the previous 24 hours

This application uses a map with overlays to indicate the locations (past and present) of the item being tracked. Map-like interfaces like this are often used for displaying events relating to specific locations.

Other styles of user interface include travel departure and arrival boards and sports scoreboards. These can be simulated by computer applications or implemented as physical displays. The difference between these two is blurring as more and more display boards are being implemented in software; the physical departure board is rapidly becoming a thing of the past.

Event Processing applications can also make people aware of events by distributing them via web feeds (using ATOM or RSS protocols). Users can view such events using readily-

available feed reader software. In similar vein, event notifications can be posted to a wide community using Twitter or social networking sites. Figure 5.8 shows Twitter[2] being used.

Figure 5.8. This screenshot shows how Twitter can be used to report the position of a vehicle.

The Twitter panel shown in figure 5.8 is part of the same bus tracking application that we saw in figure 5.7.

5.2.3 Software event consumers

Our third family consists of event consumers that are comprised only of software and which do not themselves offer a user interface.

[2] http://twitter.com/hursleyminibus

One thing a software consumer can do is to keep a record of the events it receives, either in a flat file or a database. We refer to such consumers as *Event logs*. An event log can be useful in *active diagnostic* applications; when analyzing a problem it is frequently useful to be able to go back in time and look at events that took place in the run-up to the occurrence of the problem itself, but which appeared unremarkable at the time. Another use of an event log is to provide an audit trail, should that be needed. When designing an event log you need to consider what bits of event data need to be recorded, and what kinds of searches are going to be performed against that data. If the log is only going to be read from occasionally, then you should consider implementing the log in a way which optimizes writing of the data rather than reading it.

Software consumers also include what might be called "line of business" applications, in other words application logic not modeled in the EPN itself. The event consumer is in effect the gateway between the EPN and this application code. This category covers a wide range of core business applications, for example asset and inventory management, or enterprise resource planning and personnel systems.

There are various ways in which the consumer can integrate with the line of business application. If the application has been constructed using a Service Oriented Architecture approach, or if it has been adapted to provide SOA interfaces, then the consumer can make use of these interfaces. If this is not the case, then the consumer might have to be implemented in the form of an adapter to the application.

Another approach is for the consumer not to interface directly to a single line of business application, but rather to have it interface to a Business Process Management (BPM) system; the BPM system then orchestrates the interaction with one or more applications. There are two commonly used BPM programming models. In a workflow model (embodied by the BPMN and BPEL standards) a Business Process is defined as a graph of activities. Figure 5.9 (which comes from the BPMN 1.2 specification) shows a simple business process.

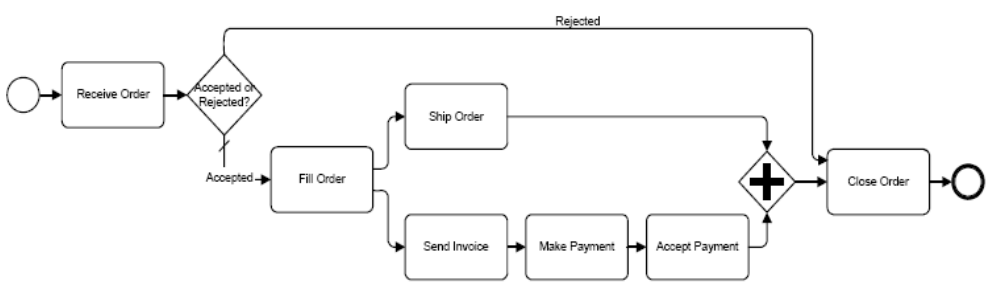


Figure 5.9. A simple business process modeled using BPMN. A process instance is created when a new order is received, and proceeds through a sequence of activities.

The receipt of an event by the consumer can either be used to trigger the start of a new instance of the process (in this example the arrival of an order would be such an event), or to terminate a process that is already running, or it could be used to cause an already-running process to transition from one activity to the next. BPM systems also make use of *state machine* programming models, and in a similar fashion the arrival of an event at the consumer can be used to instantiate a new process, or cause a state change within an existing state machine.

5.3. Interfacing with event consumers

In this section we will look, in an abstract way, at the way that event consumers attach to the rest of an event processing network. We start by looking at the interaction patterns that they use, and then we will look at the mechanics of how they connect.

5.3.1 Interaction patterns

In the previous chapter we noted that the designer of an event producer has to choose a strategy about when it produces an event. We don't have to worry about such complexities when it comes to event consumers. An event consumer interacts in pretty much the same way as any other node within an EPN. Its input terminals identify, by means of filters, the set of event types and instances that the consumer is prepared to accept, and the implementation has a choice between using a push-style or a pull-style interface. We discussed the differences between and push and pull, and reasons for choosing one over the other, in chapter 2.

The complications that occur with event consumers arise from the fact that an event consumer definition element can (and often does) represent a class of multiple concrete consumers. The first complication is that the members of the class may need to vary over time (in other the words the class could be "dynamic"). This is particularly likely to happen if the consumer provides some kind of visualization interface, as then there could be a new concrete consumer each time a new user opens the visualization application. Consumers that support this do so by providing some kind of dynamic registration interface, allowing concrete consumer instances to be registered or de-registered. We refer to the act of regsistering a consumer instance as *subscribing*, and the act of de-registering as *unsubscribing*. The entity that submits the subscribe request is called a *subscriber*. In many cases it is the would-be consumer instance that submits the subscribe request (so the subscriber and consumer are one and the same), but there are cases where one entity can request a subscription on behalf of another.

The second complication occurs because an input terminal, and its associated filters, applies to all members of the class, and so you would expect that if an event meets the

requirement of those filters then it would be delivered to every member of the class[3]. However there are cases when the Event Processing logic might want to control which members actually get to see the event. A case in point occurs in our Fast Flowers application, where we want `Bid Request` events to be delivered only to drivers who happen to be within a particular geographical location and who meet certain other criteria to do with reputation and reliability.

In some cases the best thing to do is to use smaller classes, or even to avoid using class type consumers and model each consumer instance as its own separate definition element, but if there's more than a small number of concrete instances this leads to unnecessarily complex EPN definitions, along with the need to revise the EPN definition each time a consumer joins or leaves. So to allow us to continue to benefit from the advantages of class type consumer definition elements, we allow event objects to contain *routing metadata*. This metadata is inserted into the header portion of the event object by a routing node upstream from the consumer in the EPN, and identifies one or more members of the class which are to receive the event. When an event object is received by a class type event consumer it is then tested against any filters present on the input terminal and, if it passes them, the consumer then distributes it to the instances identified by the routing metadata.

5.3.2 Interfacing mechanisms

The mechanisms that can be used to interface an event consumer with the rest of the event processing network are the same as those used with event consumers. An event processing network can support one or more event distribution mechanisms, and these mechanisms can either be protocol based or API based.

In a protocol based mechanism the consumer uses a transport protocol supported by the EPN implementation. This could be a proprietary protocol or a standardized one, and describes the way that event objects are serialized as well as the transport protocol used to transmit them. It can be support either push-based or pull-based distribution (as discussed in the previous section) or both. The protocol might also include a way for the consumer to provide a filter to exclude certain events; this allows for more efficient implementations than if the filtering is left to the consumers themselves, since it is possible for the network to combine filters from multiple consumers and thus do filtering earlier on in the distribution process.

In the API approach, the EPN provides some kind of programming interface for the consumer to use. As is the case with producers, there are two main styles of API. One is the kind of programming interface provided by Message Oriented Middleware, such as the Java Message Service API, where the consumer receives an explicit event object across the API.

---

[3] Of course an individual concrete instance is still free to ignore the event, and since the behavior of consumer instances is not modeled by the definition element there's nothing to stop some consumers from processing it and some from ignoring it.

In the other style the API provides access to one or more resources (objects recording aspects of the current state of a producer). The EPN communicates with the consumer by making updates to the state held in these resources.

Now let's turn our attention to the Fast Flower Delivery application.

5.4. Consumers in the Fast Flower Delivery example

Our Fast Flowers application contains three Consumer definition elements: Driver, Store and Manager. The Driver definition element represents the class of all drivers and is illustrated in listing 5.1. As you can see from this listing, drivers respond to `Bid Request` events (these are issued during the bid phase) and the driver selected during the assignment phase then receives an `Assignment` event.

Listing 5.1 Definition element for the Driver class

Consumer Category	Definition Element Type	Consumer Identifier	Input Terminal	Event Type
Human	Consumer Class	Driver	Bids	Bid Request
			Assignments	Assignment

The Store definition element, shown in listing 5.2, represents the florist stores that participate in the program. Stores that have elected to do manual assignment will receive Delivery Bid requests from the drivers bidding for work. There is also a collection of alert events that a store can receive.

Listing 5.2 Definition element for the Store class

Consumer Category	Definition Element Type	Consumer Identifier	Input Terminal	Event Type
Human	Consumer Class	Store	Bids	Delivery Bid
			Alerts	No Bidder
			Alerts	No Assignment
			Alerts	Delivery alert

There is a single manager component. Its definition element is shown in listing 5.3.

Listing 5.3 Definition element for the Monitoring system Manager

Consumer Category	Definition Element Type	Consumer Identifier	Input Terminal	Event Type
Human	Consumer Instance	Manager	Alerts	No Bidder
			Alerts	No Assignment

```
Alerts        Pick-up alert
Alerts        Delivery alert
```

This component listens for the various alerts generated while the application is running.

5.5 Summary

In this chapter we have moved to the other end of the event processing flow and looked at the event consumer. We have discussed the event consumer definition element, and noted that it can be used to represent an abstract consumer type, a concrete event consumer or a class of multiple concrete event consumers. Similar to event producers, we notice that the event consumer's internal logic is outside the scope of the event processing network and that an event consumer node in an EPN is a proxy that represents the connection from one or more real event consumers to the rest of the event processing network. In cases where the consumer definition element does represent more than one real consumer, we may need to use routing metadata in the event object to target an event at a subset of these real consumers. We have also discussed a number of different kinds of consumer and looked at interactions between consumers and the rest of the network.

Now we have looked at producers and consumers, it is time to turn to the event processing network itself, and the way that it is used to connect event processing entities to each other.

Additional reading

Nils H. Rasmussen, Manish Bansal, Claire Y. Chen: Business Dashboards: A Visual Catalog for Design and Deployment, Wiley, 2009

http://www.amazon.com/Business-Dashboards-Visual-Catalog-Deployment/dp/0470413476/ref=sr_1_2?ie=UTF8&s=books&qid=1258900014&sr=1-2

This book discusses the concept of business dashboards and their implementation

Exercises

5.1 Give an example where it is useful to have an event consumer definition element representing an abstract type.

5.2 Explain, with examples, how Generalization and Specialization relationships apply to event consumer definition elements.

5.3 Give two ways in which you could avoid using routing metadata in the Fast Flowers Delivery application. What the advantages or disadvantages of these approaches?

5.4 Give an example of an application of the Ambient Orb, and describe the coloring scheme

5.5 Give an example in which the consumer is a workflow, and events are used to start a new process, stop an existing process, and modify the state of an existing process

5.6 Give two examples of applications that use dynamic consumer subscriptions, and one where all the consumer instances are statically defined.

5.7 Give an example which involves third-party subscribers, that is an example in which a consumer instance is registered by a separate entity

5.8 Describe the possible security implications of third-party subscriptions. Suggest ways in which they can be mitigated.

6

The Event Processing Network

"If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them."

- Henry David Thoreau

The event processing network is the central concept in this book. It links two of our fundamental building blocks, the event producers that we looked at in chapter 4 to the event consumers of chapter 5 and it provides a way of modeling the processing that takes place between these producers and consumers. This intermediate processing is represented as a graph that uses a further three building blocks: event processing agents, global states and channels.

In this chapter we will look at the event processing network and these three building blocks. In particular we discuss:

- The event processing network itself. We start with a summary of the concept and the notation that we introduced in chapter 2 and then look at some further details
- The concept and anatomy of event processing agents, and a discussion of the different types of EPA. We explore these types in more depth in chapters 8 and 9.
- The concept and functions of an event channel
- The way that we represent global state in an event processing network

We illustrate these points by describing the EPAs, channels and state definition elements from the Fast Flower Delivery example that accompanies us throughout this part of the book.

6.1 Event processing networks

When we introduced event processing networks in chapter 2 we showed how an Event Processing Network can be viewed as a collection of event processing agents, producers, consumers linked by channels.

We will start with a brief recap of this idea, showing the graphical notation that was introduced in that chapter, and extend it to add elements that represent global state. We will then look at how you can use recursion in the EPN model to simplify complex networks and to reuse network sub-elements. We conclude this section with a discussion of how the conceptual event processing network that we have been talking about relates to the actual artifacts that make up an EPN implementation, and a summary of the rationale for having a conceptual network representation in the first place.

6.1.1 The event processing network and its notation

Figure 6.1 shows our graphical notation, and illustrates a number of features of an event processing network. We represent a network as a kind of graph and the various processing elements that make up the network (producer, agent, consumer, channel) are shown as shapes, and make up the nodes of this graph. These nodes have input or output terminals, shown as triangles (a triangle pointing into a shape is an input terminal; a triangle pointing outwards is an output terminal).

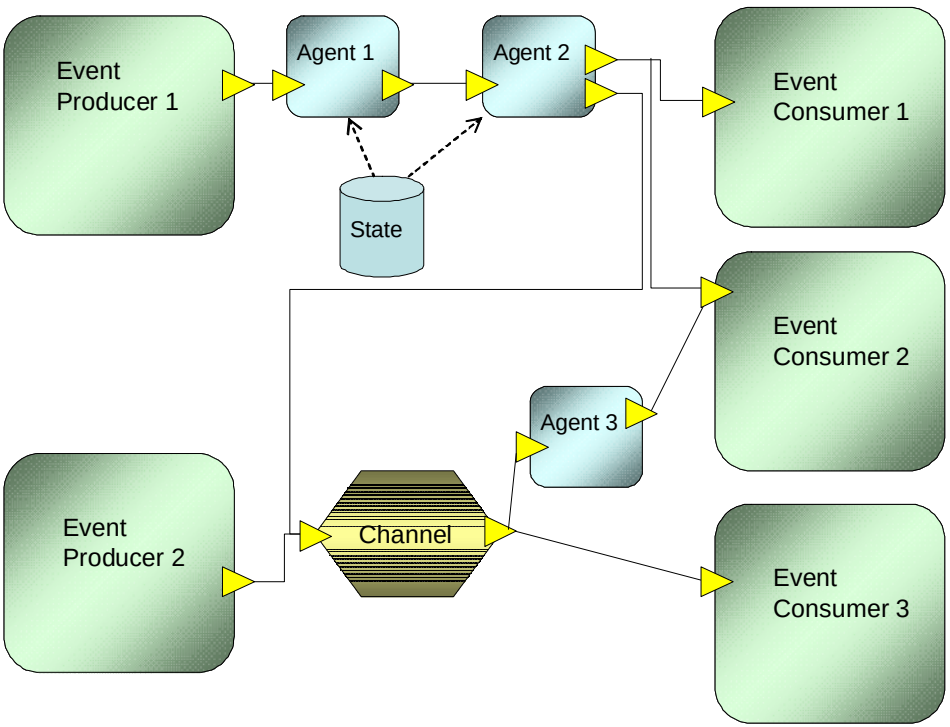


Figure 6.1. An example Event Processing Network, showing the graphical notation we are using, and illustrating some of the features of an event processing network.

An output terminal can be connected to an input terminal via a solid line. We refer to these lines as *links* or *edges,* and you can see there are several of them in the figure. The presence of a link indicates that any event instance emitted by the output terminal is to be distributed to the corresponding input terminal. If you prefer to think about event distribution in terms of streams, then you can picture a stream of events that emerging from the output terminal and flowing along the link.

In the figure you will see that there are two links emerging from one of the output terminals of Agent 2. This means that when Agent 2 emits an event through this terminal, two copies of the same event instance are distributed. In this case one copy goes to Consumer 1 and the other to Consumer 2. The notation does not dictate the order in which these two copies are distributed. In some implementations the two copies might be distributed concurrently, in others it could be one after the other. In stream terms, the stream that emerges from the output terminal splits into two equal copies, one copy of the stream flowing along each edge.

It's also possible to have two links connected to the same input terminal, as you can see by looking at the input terminal for Consumer 2. This means that the input terminal can receive events from either link, and the streams of events coming in along these links are interleaved. The manner in which this interleaving occurs is implementation dependent. To impose a specific order on the interleaved stream you use an explicit compose EPA with two separate input terminals, there is more about specific EPA's like this later in this chapter.

While you can link producers, consumers and EPA's together using edges, you can also link them using an explicit channel node. You can see an example of a channel being used at the bottom of figure 6.1. The advantage of representing a channel with a processing element (and thus have it appear as a node in the graph) is that it lets you specify how you would like the channel to behave, and have this specification included as part of the overall event processing network definition. There is more about channel specifications later in this chapter.

An event processing network graph can contain one further type of node called a global state element. These elements, which we discuss further in section 6.4, represent stateful data that can be read or updated by event processing agents when they do they work. The EPN notation includes additional edges to record the relationship between global state elements and the event processing agents that use them. These edges differ from the edges we have discussed so far, in that they represent the transfer of data to or from a global state element, rather than the transfer of events between elements, so we show them using dashed rather than solid lines. You can see examples at the top of figure 6.1 where Agent 1 and Agent 2 both use the element called State.

6.1.2 Recursive Event Processing Networks.

The graphical representation of a large event processing network can be somewhat unwieldy, but this can be simplified by nesting one event processing network inside another. This also means you can use a hierarchical approach[1] to designing and maintaining an event processing network.

The nesting process works like this: an event processing agent can represent a nested EPN instead of representing a single agent. The terminals of the EPA correspond to producers and consumers of the EPN that it contains. This process could continue again, some of the event processing agents in the nested EPN could themselves contain further sub-nested EPNs, so we refer to the nesting of EPNs as *recursive* EPN composition. We show an example of this in Figure 6.2.

---

[1] Hierarchical approach for design of software is quite common, and was originated from the HIPO (Hierarchical Input Process Output methodology); William S. Davis, David C. Yen : The Information System Consultant's Handbook: CRC Press, 1998. Chapter on HIPO
Hhttp://www.hit.ac.il/staff/leonidM/information-systems/ch64.html

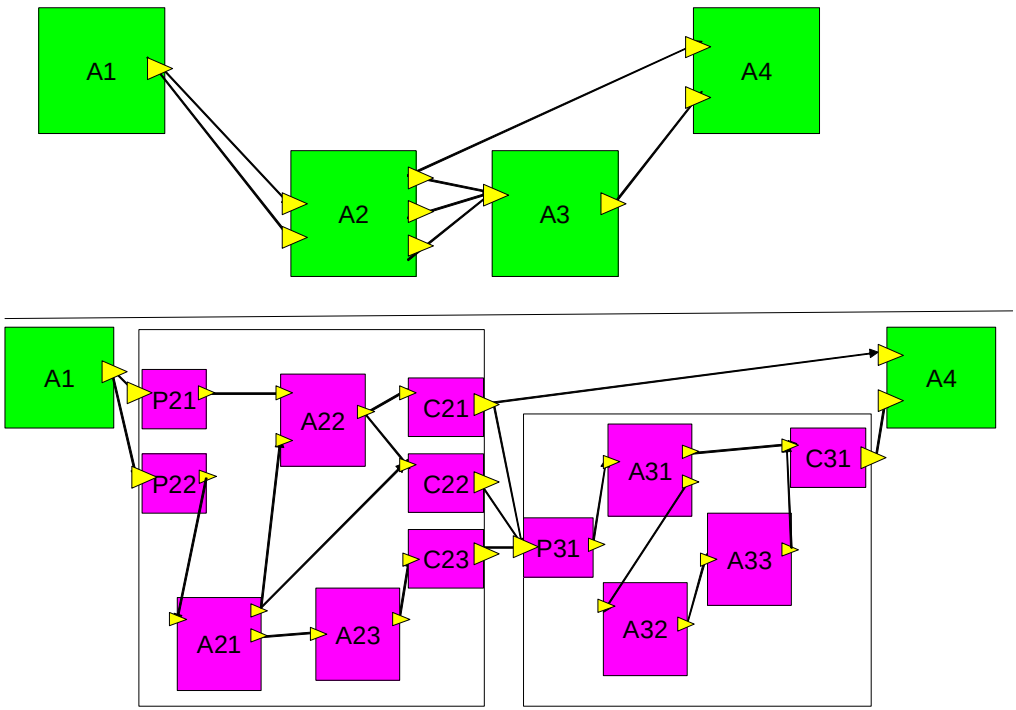


Figure 6.2. An example showing recursive EPNs. The upper part of the figure shows an EPN graph with four nodes. Two of these (A2 and A3) are agents that contain nested EPN definitions. The lower half of the figure shows the full graph with these nodes expanded to reveal the agents inside them.

The graph at the top of the figure shows part of the unexpanded network consisting of four nodes: A1 through A4 and their interconnections. At the bottom of the figure we see the expanded graph; agent A2 is expanded to an EPN that contains three EPAs – A21, A22, A23, likewise A3 is expanded to an EPN that contains the EPAs A31, A32 and A33. You will see that the input terminals of A2 correspond to producers P21 and P22 in the leftmost nested EPN, and the output terminals of A2 correspond to consumers C21, C22, C23.

6.1.3 Implementation perspective

In chapter 2 we discussed the difference between a platform-independent definition element, which describes a piece of event processing functionality in an abstract manner, and the concrete software or hardware artifact[2] that implements it.

The event processing network, as we have described it, is constructed out of platform-independent elements. It is therefore an abstraction that describes the functional behavior of the event processing application, without necessarily representing the physical realization of this functionality. This means that the event processing graph can be set out in a way that expresses the behavior as clearly as possible.

When implementing or managing an event processing application you are dealing with the concrete artifacts that deliver the functionality specified by the abstract EPN definition. There are many different ways in which you can realize the elements in an abstract EPN as concrete artifacts, and we show some of these approaches in Figure 6.3.

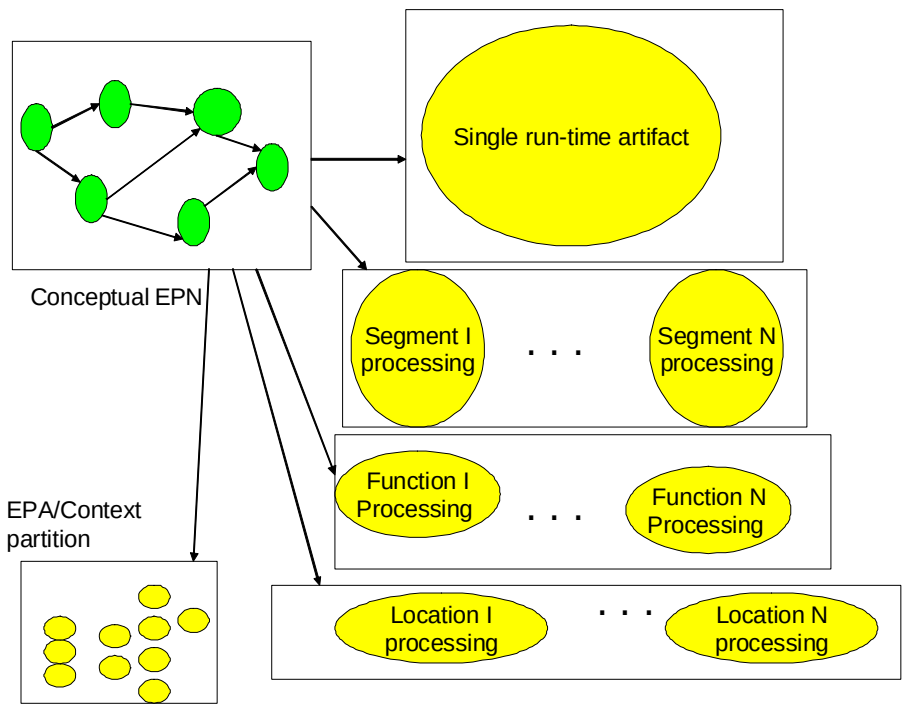


Figure 6.3. Some possible mappings of a conceptual EPN to run-time artifacts.

[2] We use the term "run-time artifact" to mean a software module which can be an event processing engine, an instance of such an engine that deals with a specific function, or a special piece of code that implements a specific function

There are various ways in which an EPN can be implemented:

- The entire EPN is run by a single centralized run-time artifact; in this case this single run-time artifact contains the functionality of all the EPA instances and the routing of derived events to EPAs that use them as input, is also part of this logic. Many of the existing event processing products either started in this manner, or still practicing a centralized solution
- The other extreme is that there is a separate run-time artifact for each atomic unit of processing, which is a combination of EPA and context partition[3]. This provides the maximal modularity and can serve as a basis for parallel processing.
- In between these two extremes there are several options partition the EPA space to a collection of run-time artifacts, each may include multiple EPAs. This can be a basis for a distributed implementation[4]
- As shown in Figure 6.3 this can be based on segments, for example: all processing of platinum customers are done by one run-time artifact, all processing of gold customers are done by another run-time artifact and so on;
- the partition can also be done on functional basis, for example, all the processing related to bid requests and assignments in the FFD example, are done by one run-time artifact, while all time-outs are handled by another run-time artifact, and all rankings are done by a third run-time artifact;
- Additional partition can be geographical partition, for example in the FFD example, each store has a dedicated run-time artifact, while drivers' ranking is done on a separate run-time artifact

6.1.4 Benefits of an explicit EPN representation

Some event processing systems do not contain an explicit EPN concept. A designer using one of these systems defines various functions, and the relationships between them are inferred by the system and are hidden from the designer. However, experience shows that there are some benefits of making event processing networks explicit and visible to the designers, developers and even end-users. Here are some of these benefits:

1. People tend to place more trust in systems in which the flow is explicit. Even if it does not have to be defined explicitly, it should be visible, and updatable. This is based on experience and user feedback.
2. If you have an explicit representation of an event processing network then you can validate the network using static and dynamic analysis techniques to detect possible

[3] Contexts are explained in detail in Chapter 7
[4] Chapter 10 deals with distribution and parallel implementations

problems such as termination, inaccessible nodes, non-deterministic behavior, contradictions, and make other observations. We will return to the subject of validation in chapter 10.

3. An explicit representation of an event processing network can be used to perform performance optimization such as mapping of EPAs to software artifacts, or distribution of those artifacts among threads and servers.

Now we move from talking about the Event Processing Network in general to talk about the various components of the network, starting with event processing agents.

6.2 Event processing agents

The Event Processing Agent (EPA) is one of our seven building blocks. It plays a major part in the conceptual Event Processing Network and, as we explained in the previous section, it can be mapped in different ways to runtime artifacts. There are several types of event processing agent, but before we discuss them we will look at the logical structure of an EPA and the kinds of function that an EPA can include.

6.2.1 The functions of an EPA

Figure 6.4 shows the anatomy of an EPA with its three logical functions:

- Filtering: selecting which of the input events participate in the processing. Filtering can be performed in multiple places, and is discussed further in chapter 8.
- Matching: finding patterns among events and creating sets of events that satisfy the same pattern, discussed in chapter 9.
- Derivation: using the output from the matching step to derive new events and setting their content, discussed in chapter 8.

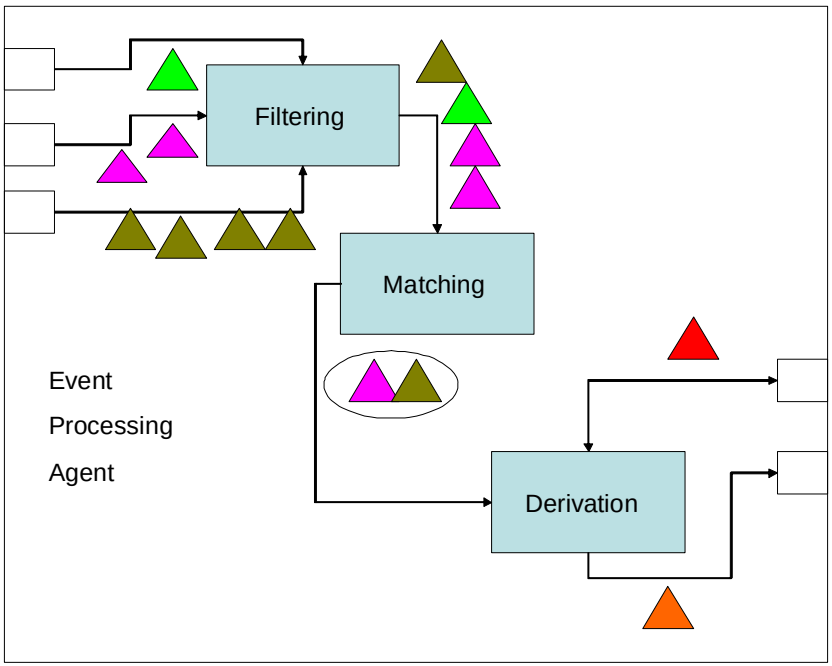


Figure 6.4 An example of an Event Processing Agents showing its three logical functions.

Figure 6.4 illustrates the logic and flow of an EPA. In this example there are three input terminals, which are the entry points to the EPA. Event instances flow into the EPA via these terminals. As we saw in chapter 2, each terminal can have a filter condition associated with it selecting event instances based on their type and/or based on the values of the various attributes of the event. An EPA can also be associated with a *context* and, if this is the case, any filtering associated with this context is also performed at this stage (we will discuss contexts further in chapter 7). The filtering step takes each incoming event as an input, and applies the filter conditions. In general it eliminates any event instance that does not meet these conditions[5]. The matching step takes all events that have been left by the filter step, and looks for matches between them, using an event processing pattern or some other kind of matching criterion. It creates matching sets, each of which contains a collection of event instances that satisfy the criterion. In the example in figure 6.4 there is a single matching set

[5] In the next section we will see that sometimes we are also interested in processing events that are filtered out.

that contains two events of different types. The derivation step takes the matching set as an input and derives new events, applying a derivation function on the events in the matching sets. It should be noted that some EPAs can omit one or more of these functions, for example an EPA might contain only filtering and derivation without matching, in which case the events inform the filter step are input directly to the derivation step. We will now look at some specific types of EPA.

6.2.2. Type of EPA

As we pointed out earlier, there are several types of EPA. Figure 6.5 repeats figure 2.10 in showing the most important of these types.

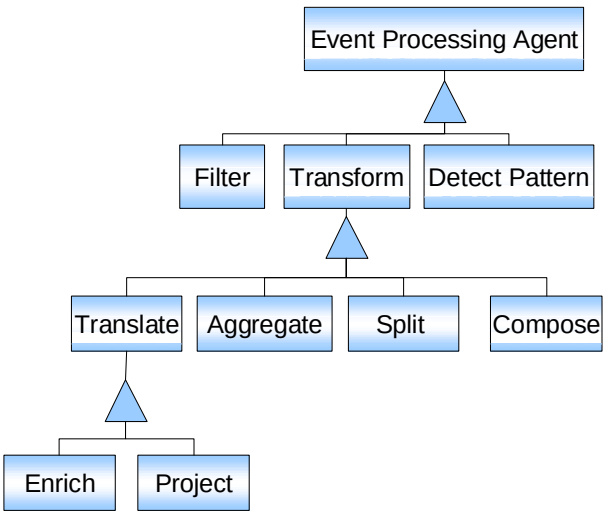


Figure 6.5 Some specific types of EPA. This inheritance diagram illustrates relationships between these types, for example Enrich and Project are special cases of Translate, which in turn is a special case of Transform.

In the sections that follow we include some formal definitions of these EPA types, to supplement the informal descriptions from Chapter 2.

6.2.3 The Filter EPA

Event Processing applications sometimes involve event producers which generate a large volume of events, not all of which are of interest to the application. A `filter` EPA can be used to reduce this volume by excluding unwanted event instances. Although any EPA can

perform filtering, because each input terminal can have an associated filter expression, it's helpful to have an EPA that focuses solely on filtering.

Definition

A filter EPA is an EPA that performs filtering only and has no matching or derivation steps, so it does not transform the input event.

A filter EPA, shown in figure 6.6, has one input terminal and three output terminals, which we will discuss shortly. It also has a filter expression which determines which event instances are to be *filtered in* (selected) by the agent and which are to be filtered out. The exact definition of the filtering expression is discussed in Chapter 8.

Filtering is always performed on a single event at a time; no state is carried forward from one event instance to the next, so when making the filtering decision the agent does not take any earlier events into account. A filter agent may be intended to handle multiple different event types, in which case the filter expression should be well-defined for each of these event types, for example an expression that filters events depending solely on the value of an attribute called `Driver` will work as intended against any event type that contains a `Driver` attribute.

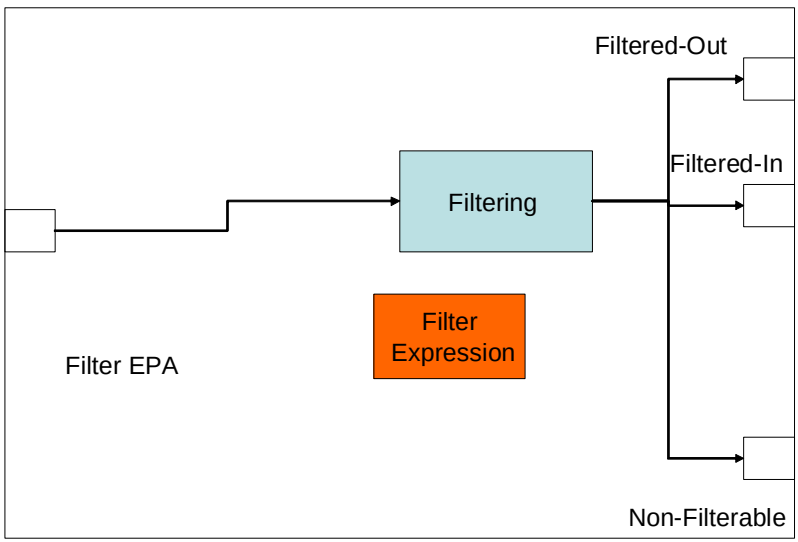


Figure 6.6 The filter EPA showing the input terminal and three output terminals.

There are three possible output terminals, although of course when constructing an EPN you do not need to have edges connected to all three. These output terminals are:

- Filtered-in: Any input event that satisfies the filtering expression flows out through this terminal.
- Filtered-out: Any input event for which the expression can be evaluated, but which does not satisfy the filtering expression flows out through this terminal.
- Non-filterable: Any input event for which the expression cannot be evaluated flows out through this terminal. Note that the definition of a "non-filterable" event depends on the language used for the filter expression. Some languages, for example XPATH, are reasonably tolerant and will generally attempt to filter events either in or out.

The Filtered-out terminal is a special feature of this agent. You can connect up the Filtered-in and Filtered-out terminals to different EPAs, so as to arrange for an event instance that doesn't satisfy the filter expression to be processed differently from one that does.

As seen from the definitions, a filtering agent transfers an event from its input terminal to one of the output terminals and does not make any transformation of its content. We now look at EPAs that specifically focus on transforming events into new derived events.

6.2.4 The Transform EPA

Transform EPAs take input events and create output events that are functions of these input events, as shown in Figure 6.7. Transform EPAs can be stateless, processing each event instance individually, or stateful in which case the way a particular event instance is processed can depend on other instances that have been processed by the EPA.

Definition

A transform EPA is an EPA that performs the derivation function, and optionally also the filtering function.

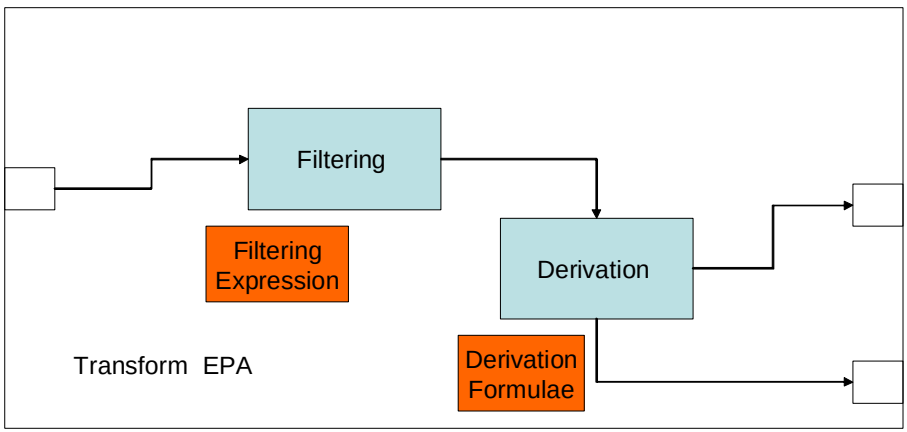


Figure 6.7. An illustration of the transform EPA.

There are several different types of transform EPA, as shown in Figure 6.8. They differ by the kind of transformation they perform, for example some are stateful and some stateless. They also differ depending on whether they take a single input stream or multiple input streams, and whether they emit a single output stream or multiple output streams. We will discuss each type in turn starting with the simplest, the `translate` EPA.

Definition

A translate EPA is a stateless transform EPA that takes as an input a single event, and generates a single derived event which is a function of the input event, using a translation formula.

The `translate` EPA is useful in order to convert events from one type to another, or to add, remove or modify attributes of the event. An example of a type change is an XSLT program that translates XML event formats; derived attributes are defined and discussed in chapter 8. `Enrich` and `project` are special cases of the `translate` EPA type.

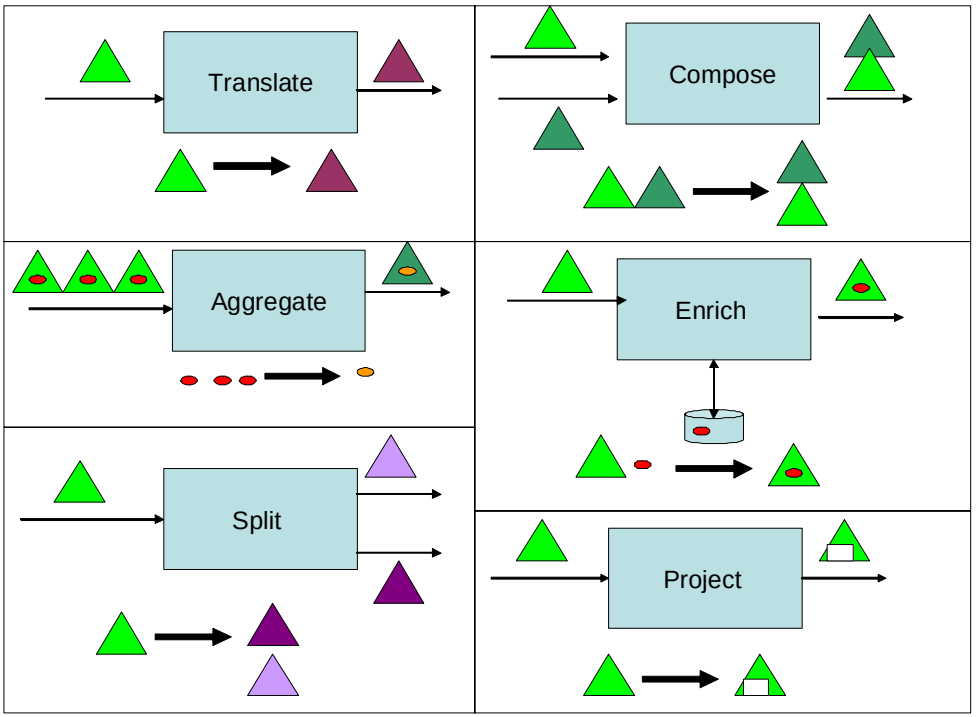


Figure 6.8 Illustrations of the six subtypes of the transform EPA, showing the input, output and transformation type.

Definition

An `enrich` EPA is a translate EPA that takes a single input event, matches it against a global state element, and creates a derived event which includes the original event, with possible modified attributes, and an additional collection of attributes {A1,…,An} copied or calculated as a result of using the global state.

Global state elements are discussed later in this chapter; an example of a global state is a table containing reference data.

Definition

A `project` EPA is a subtype of the translate EPA that takes an input event, and creates a single derived event that contains a subset of the attributes of the input event

A `project` EPA is similar to the project operator in relational algebra; it selects a subset of the attributes of a single event.

Now we will look at stateful transform EPAs.

Definition

An aggregate EPA is a transform EPA that takes as input a collection of events and creates a single derived event by applying an aggregation function over the input events.

The input of the `aggregate` EPA is a collection of events and the output is a single derived event. In figure 6.8 the illustration shows an aggregation over a single attribute, but the aggregate operation may involve multiple attributes. Note that the input events may be processed together as a set once they have all arrived, or one by one so that the aggregation is computed incrementally. Examples of aggregation functions are: sum, average, maximum, minimum; there are, of course, many more aggregation functions.

Next, we define the `split` EPA.

Definition

A `split` EPA is a transform EPA that takes as an input a single event and creates a collection of events, each of them can be a clone of the original event, or a projection of that event containing a subset of its attributes.

The `split` EPA creates multiple derived events as a result of processing a single input event instance. Split can be used to send different portions of the input event to different agents.

Definition

A `compose` EPA is a transform EPA that takes groups of events from different input terminals, looks for matches using some matching criterion and then creates derived events based on these matched events.

The `compose` EPA creates a set of derived events; each of them is a function of a composite event that includes events from each input terminal. This is similar to the join operator in relational algebra.

Note that it's possible to concatenate a number of these transform EPAs to produce a more complex transformation. These transform EPAs could be combined together to form a composite EPA using the recursive composition approach described in section 6.1.2. Some examples are:

- An event e1 is an input to an `enrich` function, which derives the event e2, which in turn is an input to a split function that creates the event e3, e3, 35.
- The events e1, e2, e3 are aggregated through an aggregate function to an event e4,

which is enriched by an enrich function to create the derived event e5.

6.2.5 The pattern detection EPA

A pattern detection EPA uses all the different steps described in section 6.2.1. It examines a stream of incoming event instances looking for the occurrence of a specific pattern in that stream.

Definition

A pattern detection EPA is an EPA that performs a pattern matching function on one or more input streams. It emits one or more derived events if it detects an occurrence of the specified pattern in the input events.

The notion of pattern is defined and discussed in length in Chapter 9. A pattern detection EPA works on multiple events possibly from multiple input terminals, and possibly of multiple event types. As with all EPAs the input terminal may specify a filter condition. The result of a pattern matching is a matching set that contains all the events that meet the pattern. This matching set can serve as an input to a derivation function that does various kinds of transform functions. If no derivation is specified, the matching set participants flow out from the output terminal. The Fast Flower Delivery application uses a simple pattern detection agent to detect when a driver has not met the committed pick-up time.

We conclude this section by showing the EPA definition element.

6.2.6 The EPA Definition Element

Now we have discussed a number of different types of EPA, it's time to see how to represent them in an Even Processing Network definition. The Event Processing Agent is one of our seven fundamental building blocks, so we represent each EPA instance using an event processing definition element. All EPAs, regardless of their type, have definition elements that take the shape shown in figure 6.9.

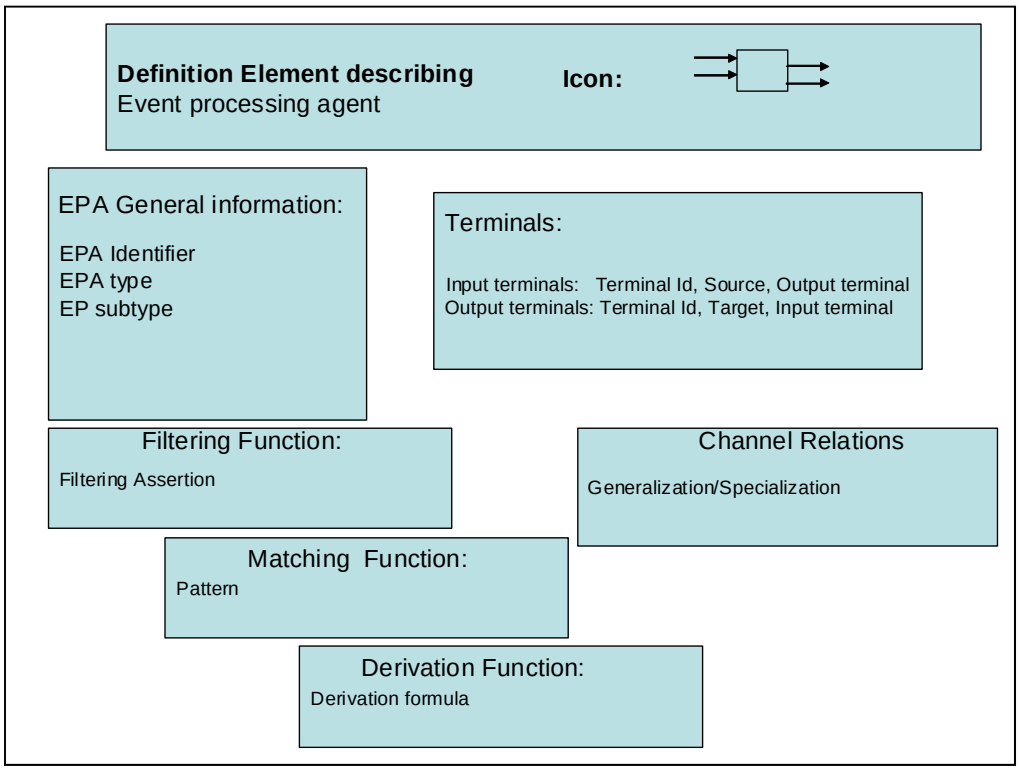


Figure 6.9 The EPA Definition Element

Figure 6.9 describes the EPA building block, it consist of:

- General information that includes identifier, EPA type, and subtype (in case of the transform type). Note that besides the regular types of EPA, an EPA can be of type EPN, and thus designate a recursive EPN.
- Input and output terminals, and their connections to the source/target.
- The three functions – filtering, matching and derivation. The exact definitions of filtering assertion and derivation formulae are introduced in Chapter 8. The exact definition of pattern is introduced in Chapter 9.

6.2.7 Event processing agents in the Fast Flower Delivery example

Listing 6.1 shows all the EPAs defined for the Fast Flower Delivery. It does not include their full specification; this will be completed in chapters 8 and 9.

Listing 6.1: Event Processing Agents in the Fast Flower Delivery Example

EPA Identifier	EPA type	Input Terminals	Output Terminals
Bid Request Creator	Enrich	Bid Request Channel, Store Reference Global State	Delivery Request channel
Location Service	Translate	GPS channel	Delivery Request Channel
No Bidders	Pattern	Delivery Bid Channel	No Bidder Channel
Automatic or manual Matching	Filter	Delivery Bid Channel	Store, Automatic matching EPA
Automatic Matching	Pattern	Automatic matching EPA	Assignment Channel
Assignment not done	Pattern	Assignment channel	Assignment Timeout channel
Pick Up Alert	Pattern	Pick-up confirmation channel	Alerts channel
Delivery alert	Pattern	Delivery Confirmation Channel	Alerts Channel
Ranking Increase	Pattern	Ranking Input Channel	Ranking Output Channel
Ranking Decrease	Pattern	Ranking Input Channel	Ranking Output Channel
Improving Note	Pattern	Ranking Output Channel	Improvement Note Channel
Daily Assignments calculator	Aggregate	Assignment Channel	Daily Assignment Channel, Input Evaluation Input Channel
Daily	Aggregate	Daily	Daily

Statistics Creator		Assignment Channel	Channel
Permanent Weak Driver	Pattern	Daily Channel	Evaluation Input Channel
Idle Driver	Pattern	Evaluation Input Channel	Evaluation Output Channel
Consistent Strong Driver	Pattern	Evaluation Input Channel	Evaluation Output Channel
Consistent Weak Driver	Pattern	Evaluation Input Channel	Evaluation Output Channel
Improving Driver	Pattern	Evaluation Input Channel	Evaluation Output Channel

Event Processing Agents play a critical role in an Event Processing Network, as it is the EPAs that perform the intermediary processing that takes place in the application. However we can't finish our discussion of Event Processing Networks without mentioning the two other building blocks that they can contain. These are event channels, which route events between EPAs, and global state elements which support EPAs by providing shared state data for them to use.

6.3 Event Channels

In Chapter 2 we explained the rationale for using event channels as intermediaries between EPAs. We noted that there are advantages in representing channels as explicit nodes in the event processing network, both because this adds clarity in cases where there are many processing elements to be connected, and also because a channel provides a way to specify routing behavior explicitly. We start the discussion by defining the concept of the event channel itself and then discuss routing schemes.

6.3.1 The event channel notion

An event channel routes event instances between other processing elements[6] in the event processing network. As with our other building blocks, there are various ways in which a channel can be implemented as a run-time artifact. For example it could be implemented as a queue, a function call, or a publish-subscribe topic. The best choice for an implementation depends on the function required of the channel, the environment in which it runs and the nature of the processing elements to which it is connected.

Definition

[6] We refer to EPA, producer and consumer as processing elements.

> An event channel is a processing element that receives events from one or more source processing elements, makes routing decisions, and sends the input events unchanged to one or more target processing elements in accordance with these routing decisions.

The simplest sort of channel accepts events from a single source and routes them all to a single target. We don't require simple channels like this to have an explicit definition element or to appear as explicit nodes in the graph of the conceptual event processing network. If all you want to do is to route events from a source element to a target element you can just define an edge in the conceptual EPN graph that links the two elements (there will still of course be a run-time channel artifact, it's just that it doesn't appear as an explicit node in the conceptual EPN graph). This makes the EPN presentation simpler and less crowded – in Chapter 2 you can see some example event processing network graphs that do not use explicit channels.

More complex channels, for example channels that make routing decisions, do feature as nodes in the conceptual event processing network graph, and so we have a building block that is used to define event channel definitions elements. An event channel definition element has one or more input terminals and these receive events via a link (edge in the event processing network graph) from a source element (a producer or EPA node in the graph). A link is usually created by setting the target attribute of the source's output terminal to point at the channel's input terminal.

An explicit event channel can have multiple output terminals and, depending on the routing scheme used, it is possible that an event instance received on an input terminal is forwarded through multiple output terminals. If this happens, each event instance that is forwarded is a logical copy of the original event, each of them is independent of each other. The conceptual model makes no assertions about the order in which these events are emitted by a run-time channel artifact.

The channel building block illustrated in Figure 6.10 provides the means to define explicit channels. The different parts of this definition element are as follows:

- Terminals: the input and output terminals connecting with sources and sinks.
- Routing schemes: The way that the routing decision is done, this is further discussed in section 6.3.2
- Quality of service assertions: These affect the actual implementation of routing. We defer the discussion about this to part III of the book.
-

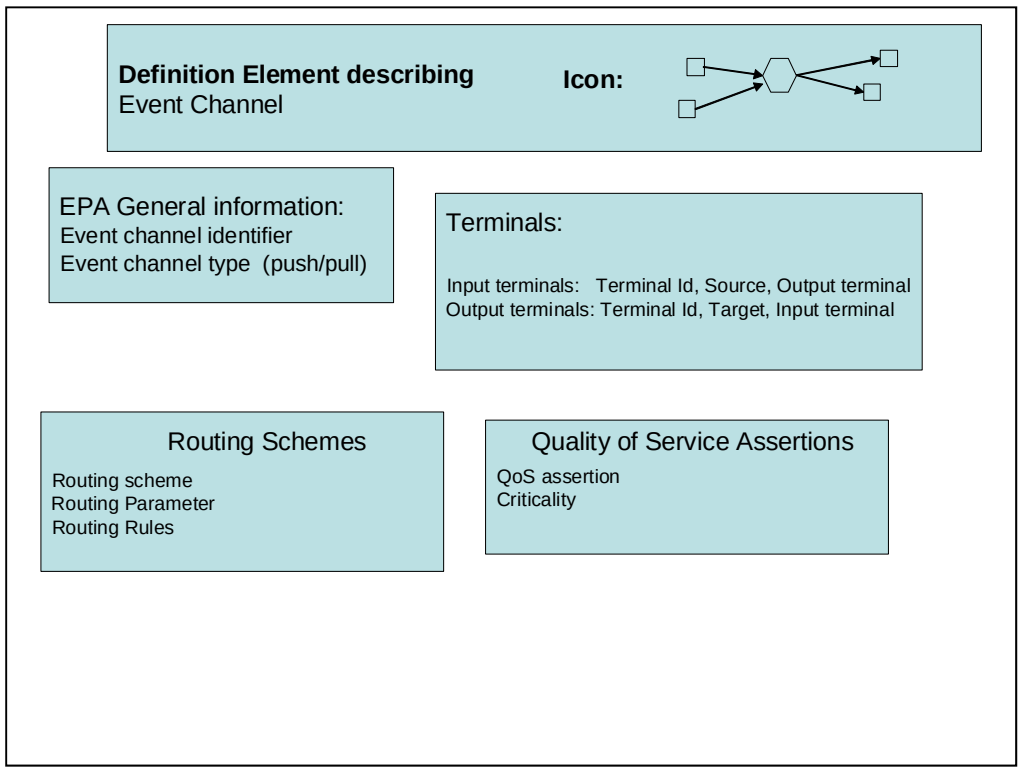


Figure 6.10 The definition element describing an event channel

6.3.2 Routing schemes

As we mentioned earlier, one of the reasons for having an explicitly modeled event channel is that it gives us a place in the model to specify how events are to be distributed. One aspect of this is the routing scheme that determines which processing element or set of processing elements is to receive any particular event instance handled by the channel.

Definition

A routing scheme denotes the type of information used by the channel to make a routing decision. The possible routing schemes are: fixed, subscription-based, itinerary-based, type-based and content-based.

Here is a short description of each of these routing schemes:

- Fixed: The channel routes every event that it receives on any input terminal to every output terminal. In cases where there are multiple output terminals this means that separate copies of each input event are transmitted on each output terminal
- Subscription-based: EPAs or consumers can subscribe to the channel dynamically. The routing decision is determined according to the list of subscribers that is valid at the time that a decision is made. Note that if traceability is desired, then the history of the subscription list should be kept. Subscription based routing enable to support dynamic EPN, we discuss this option among the advanced topics in chapter 12.
- Itinerary-based: The sink's input terminal identifier or identifiers are obtained from some attribute in the event's payload, this is used to send an event to a specific consumer instance, when the EPN node is the consumer class. Example: in the Fast Flower Delivery example, notifying a specific driver about assignment made for this driver.
- Type-based: The channel makes routing decisions based on the event type of the event that is being routed.
- Content-based: The routing decision is based on the event's content, this can be phrased as decision trees or decision tables, and are based on the input event content, as well as context information.

A routing scheme may be composed from multiple routing schemes, which means that an event is only routed to an output terminal if it would have been routed to it by all of these routing schemes. An example is that a routing scheme is composed from a subscription-based scheme and content-based scheme, and has to satisfy the content-based condition to select a subset of the subscribers that are permitted to consume this event.

6.3.2 Channels in the Fast Flower Delivery Example

Listing 6.2 shows all the simple channels used in the Fast Flower Delivery application. These can all be modeled as edges in the EPN graph.

Listing 6.2: Simple channels in the Fast Flower Delivery Example

Channel Identifier	Routing Scheme	Routing Rules/ parameter	Event Type	From (input terminal)	To (output terminal)
Bid Request	Fixed		Bid Request	Store	Bid Request Creator
GPS Channel	Fixed		GPS Location	GPS sensor	Location Service EPA

Automatic matching channel	Fixed	Delivery Bid	Manual or automatic matching EPA	Automatic matching EPA
Manual matching channel	Fixed	Delivery Bid	Manual or automatic matching EPA	Store
Assignment timeout channel	Fixed	Assignment not done	Assignment not done EPA	System Manager
Pick-up Confirmation Channel	Fixed	Pick-up Confirmation	Store	Pick-up Alert EPA
Delivery Confirmation Channel	Fixed	Delivery Confirmation	Driver	Delivery Alert EPA
No Bidder Channel	Fixed	No Bidders Alert	No Bidders EPA	System Manager
Alerts channel	Fixed	All alert events	All alert EPAs	Store, Manager
Ranking Input channel	Fixed	Delivery Alert	Delivery Alert EPA	All Ranking EPAs
Ranking Output channel	Fixed	Ranking Events	All Ranking EPAs	Monitoring System, Driver's Guild
Daily Assignment Channel	Fixed	Assignment	Daily Assignment EPA	Daily Statistics EPA
Evaluation Input Channel	Fixed	Daily assignment EPA, Daily Statistics EPA	Store, Assign EPA	All evaluation EPAs
Evaluation Output Channels	Fixed	All evaluation events	All Evaluation EPAs	Driver's Guild

This listing shows the type of events that flow across each channel, and the “From” and “To” columns show the names of the processing elements attached to the channel. Where there are several similar processing elements, for example EPAs that produce alerts, they have been given a single entry, with the prefix “All”.

Listing 6.3 shows the explicit channels that are used in the application.

Listing 6.3: explicit channels in the Fast Flower Delivery Example

Channel Identifier	Routing Scheme	Routing Rules/ parameter	Event Type	From (input terminal)	To (output terminal)
Delivery Request Channel	Content Based	Driver. Area = Store Location and Driver. Ranking ≥ Store. Minimal-Ranking	Delivery Request Driver Location	Build Request Creator EPA Location Service EPA	Drivers
Delivery Bid Channel	Itinerary based	Store	Delivery Bid	Driver	Manual or automatic matching EPA No Bidders EPA
Assignment Channel	Itinerary Based	Driver	Assignment	Store, Automatic matching EPA	Driver, Assignment not done channel, Daily Statistics EPA, Daily Assignment
Improvement note channel	Itinerary Based	Driver	Improvement Note	Improve EPA	Driver

We conclude this chapter with a discussion of the sixth of our seven building blocks, the global state. This is the last of the building blocks that can appear explicitly in our graphical Event Processing Network notation.

6.4 Global State

There are several cases in which event processing makes use of stateful data. Some EPA types are stateful in nature and maintain their own local states. However there are cases where have stateful data that needs to be accessed by more than one EPA. We refer to this shared stateful data as the *global state* of the event processing network. The global state comprises a number of *global state items*:

- Historical events retained in an event store so that they can be processed at a later phase.
- Reference data that is used by event processing agents for enrichment. This data is not maintained by the event processing system, but can be considered as part of the event processing state, since the event processing results may depend upon values of this data.
- State of external entities that, like reference data, is not maintained by the event processing system but which can be used as part of processing events. Examples are:

current state of the workflow, an airport alert level, weather related states (sun, clouds, rain, snow…). Note that this state can change as a result of a (raw or derived) event.

- Global variables (whether persisted or used on a shared memory) that are used across EPAs, and can be updated by EPAs. Typically these global variables are maintained by the event processing system.

Global state is one of the seven building blocks, and as such is defined using definition elements. Figure 6.11 shows what the definition element for a global state item looks like.

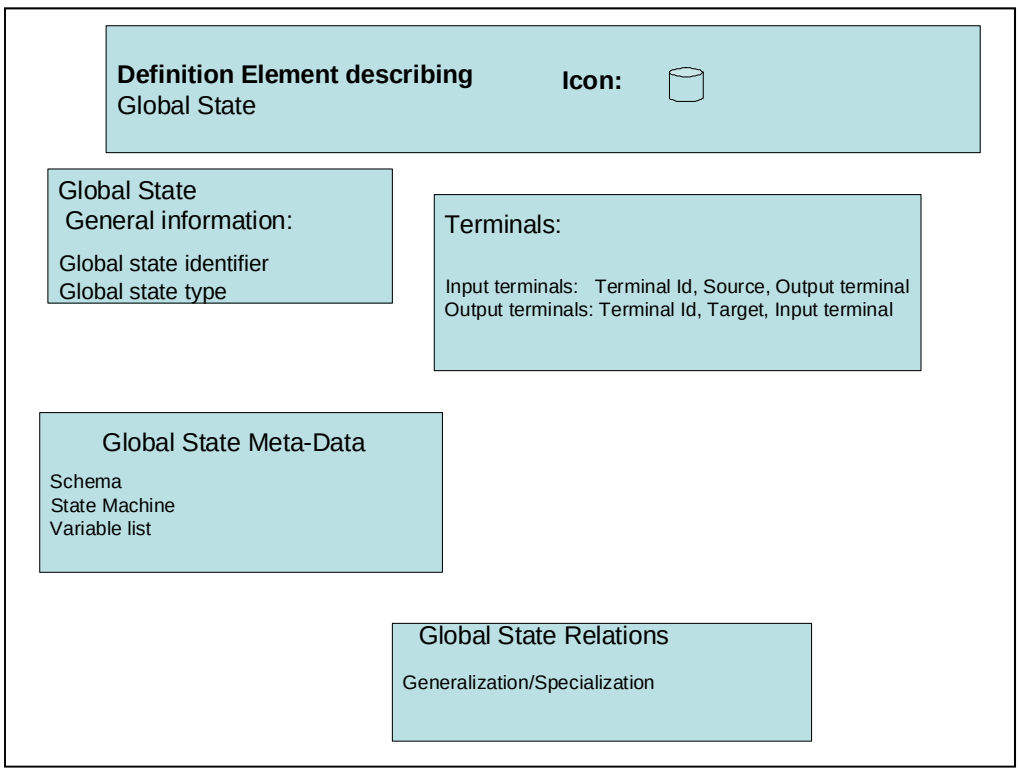


Figure 6.11 Definition elements describing a global state item

The global state type indicates the type of the item: event store, reference data, external state or global variable as explained above. The meta-data relates to the global store type, it is a schema for event store and reference data, a reference to the relevant external entity in the case of the external state type, and a variable list for global variable case. Global state also has terminals, like any other processing element. The input terminals link it to EPAs that can update the global state, while output terminals link it to EPA that retrieve from it.

Listing 6.4 shows the global states that are used in the Fast flower Delivery example

Listing 6.4 Global states in the Fast Flower Delivery Example

Global State Identifier	Global State Type	Meta-Data	Input Terminals	Output Terminals
Location-Reference	Reference Data	Geospatial DB schema		Location Service EPA
Driver-Variable	Global Variable	Driver Ranking	Ranking EPA	Delivery Request Channel
Store-Reference	Reference Data	Store Minimal Ranking, Store location		Delivery Request Channel

There are three global states in the example. Location-Reference is a geospatial reference database that divides the city into areas. Driver-Variable is a global variable that keeps the current driver's ranking, and Store-Reference is reference data relating to a store. This reference data comprises two attributes, Store location and Store Minimal Ranking. Both of these are used as part of a routing decision.

6.5 EPN in practice

In this section we show some examples of the concepts defined in this book as implemented in various languages. For thorough look at various implementations the reader is invited to use the book's website.

Figure 6.12 shows a graphical representation, of part of the EPN that taken from the Streambase solution to the FFD example, for those who are used to code being imperative code this may not look like a code example, but in fact the developer builds the application using such a graphical user interface.

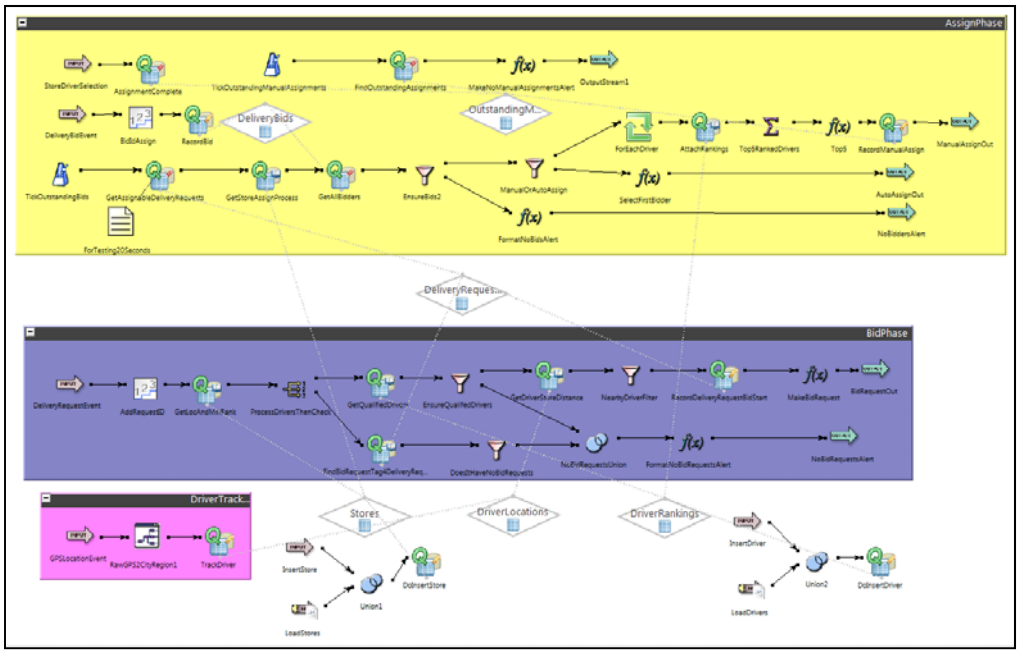


Figure 6.12 A graphical user interface to define EPN from the Streambase language.

This kind of graphical interface allows to develop EPN in a top down and explicit fashion, while in other cases it is being developed in bottom up and implicit fashion, each EPAs are constructed in separation, and the EPN is built by subscription or defining input events to various EPAs.

Figure 6.13 shows another example of top down view of EPN taken from Event Zero[7], which has an explicit EPN model.

[7] Event Zero is not one of the participants on the book website, so additional information about their language can be obtained directly from the company.

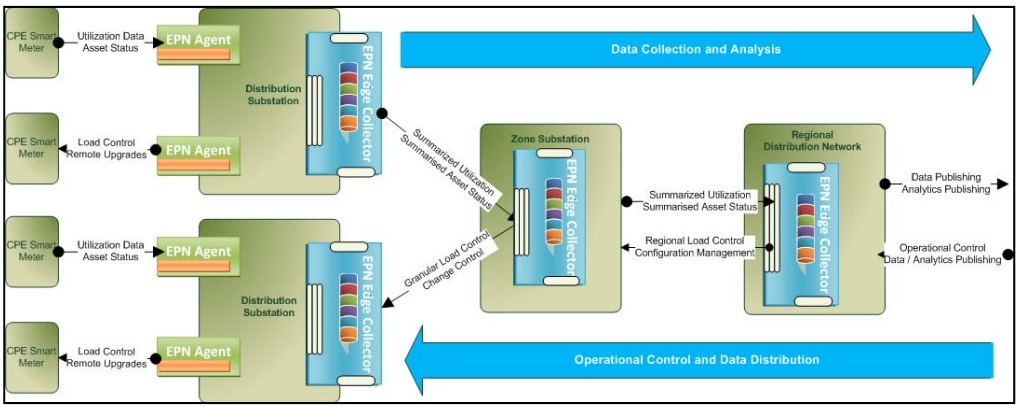


Figure 6.13 An explicit EPN representation from Event Zero. Note that the EPN semantics in this illustration is slightly different than the one we are using in this book, but the principle is similar.

Note that in many of the languages the EPN is not explicitly defined, as the construction of the application is done bottom up.

Routing decisions in event processing products are often mixed with EPA functionality, and there is no explicit routing entity, programmable routing entities typically exist in MOM software, where programmable channels exist.

6.6 Summary

In this chapter we covered the event processing network, the central concept in this book, and the tool we use to model the functional definition of an event-processing application. We gave a brief summary of the Event Processing Network itself, and then looked that three of the building blocks that you might find in an Event Processing Network. These are the Event Processing Agents (EPAs) which do the actual intermediary processing, the channels that link them together, and the global state elements that represent state that is accessed by the EPAs.

We will look at EPAs in greater depth in chapters 8 and 9, but in the next chapter we will encounter Context, our seventh and final event processing building block.

Additional reading

Gilles Kahn: The Semantics of Simple Language for Parallel Programming. IFIP Congress 1974: 471-475

This is a classic article in which the data flow networks were introduced. The EPN idea is a descendant of data flow network.

Guy Sharon, Opher Etzion: Event processing network: Model and implementation. IBM System Journal, 47(2):321–334, 2008. http://researchweb.watson.ibm.com/journal/sj/472/sharon.html

This paper defines event processing network and many of the related concepts, such as: event channel. It can be considered as an ancestor of this chapter (the thinking has evolved since this paper has been written).

Gregor Hohpe, Bobby Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003
http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Deploying/dp/0321200683/ref=sr_1_1?ie=UTF8&s=books&qid=1258829949&sr=1-1

This book has some patterns related to channels and routing schemes

Exercises

6.1 We have described several ways in which a conceptual EPA can be mapped to executable run-time artifacts. Can you think of another variation, we did not mention? What do you think is the benefit of each of these mappings?

6.2 Can you suggest a validation feature that can be enabled by explicit representation of EPN?

6.3 Can you think of examples in which the use of events that are filtered out, and non-filterable events are useful?

6.4 Devise an example which has at least one EPA of each of the transform EPA types.

6.5 Devise an example for each routing scheme channel.

6.6 What role can event processing have in implementing state machines? Give an example.

6.7 What role can state machines have in implementing event processing? Give an example.

7

Putting Events in Context

"The skill of writing is to create a context in which other people can think"
- Edwin Schlossberg

The way we view things in our daily life is affected by their context. Context may relate to the state of the weather, for example I can open my car with the remote control from quite a long distance at night but have to come quite close when it is sunny. Context may also relate to location, in my own city I might feel safe enough to carry money in my wallet, while in some countries which have a reputation of people being mugged, I hide the money. Context may relate to other external conditions such as the state of the traffic, the route I choose to drive to the airport might depend on my knowledge of likely traffic conditions, or on congestion reports that I have picked up from the radio.

In this chapter we discuss context as an explicit building block in our event processing model and provide a deep dive into the idea of context applied to the processing of events. This chapter discusses the notion of context and the way it affects the processing of events. It looks at four context dimensions: state-oriented, temporal, spatial and semantic, and explains the meaning of a context instance in each of these dimensions. It also introduces a mechanism to fine-tune a context definition, which we refer to as a context policy. Like the other chapters in this part of the book it also shows how contexts are used in the Fast Flower Delivery application.

7.1. The Notion of Context and its definition element

Context plays the same role in event processing that it plays in real life; a particular event can be processed differently depending on the context in which it occurs, in fact it may be ignored entirely under some conditions. You could try to achieve this effect using the event

processing constructs that we met in chapter 6, constructing an Event Processing Network that uses filtering and routing to direct events instances through different routes in the EPN, depending on the context associated with them. However this can quickly lead to large and unwieldy networks. Context-dependent processing occurs sufficiently frequently that it is worth treating context as an explicit construct in the EPN model.

There are three main uses of context with event processing applications:

- A stream, as we defined it in Chapter 1, is an open-ended set of event instances. If you want to perform an operation on the stream you cannot wait until all the event instances have been received. Instead you have to divide the stream up into a sequence of "windows" (context instances) each containing a set of consecutive event instances. You can then define the operation in terms of its effect on the events in the window. The rule that determines which event instances are admitted into which window is something we call a *temporal context*.
- A stream of events may contain events that aren't really related to each other, even if they occur close together in a temporal sense. They might refer to occurrences in different locations, or relate to different entities in the real world. *Spatial* or *Segmentation-oriented* context allows us to group these events together into context instances such that events in one context instance are not related to events in another context instance. This allows us to process the events from one context instance in isolation from events in another.
- Context also allows EPA's to be context sensitive, so that an EPA that is active in some contexts may be inactive in others. We refer to this as *state-oriented* context.

The notion of context in computer science has been explored in the discipline known as context aware computing. A number of definitions of context have been proposed, all of which have the same net result: they divide a "cloud" of event instances by classifying them into one or more sets or partitions. An event processing operation can be associated with a context, so that it operates on each of these partitions independently. This means that events in different partitions are kept separate from one another when they are processed. A context instance can be semantic, for example all events that relate to a single customer, or all events that relate to a class of customers like platinum customers, or it can be based on spatial properties, for example all events within 1 km from a given location, or it can be based on time. While context has an important role in optimizing implementations (if you have partitioned the event space you can process the different partitions concurrently) it is also a major semantic abstraction as it can affect how events are processed.

Contexts are realized in different ways in event processing languages. Some languages have the notion of context as a primitive language construct, while others have one construct for temporal grouping (this is usually called a window), and a separate mechanism for content grouping (like the SQL "group by" clause). In our model we view both of them as specializations of context since they have a similar role.

Definition

A context is a named specification of conditions that groups together event instances for the purpose of processing them together. A context may have one or more *context dimensions* and consist of one or more *context instances*.

The context dimension tells us what aspect of the event is used to do the grouping. At the start of this section we introduced four context dimensions, namely temporal context, spatial context, state-oriented context and segment-oriented context. We will discuss these dimensions, and the various types of context associated with them, in the following sections of this chapter.

We refer to the groups of event instances as context instances. The nature of a context instance depends on the type of context:

- Some context types give rise to just one context instance. We will see an example of this later when we look at fixed location contexts
- Some contexts can give rise to a fixed number of context instances, for example the distance location context.
- Some contexts, for example some temporal contexts, don't have a fixed number of instances. Instead new context instances are dynamically opened over time.
- We used the term context interchangeably with context instance, where the meaning depends on the context.

Context is one of our seven fundamental building blocks, so we use Definition Elements to represent context instances. Figure 7.1 shows the shape of the context definition element.

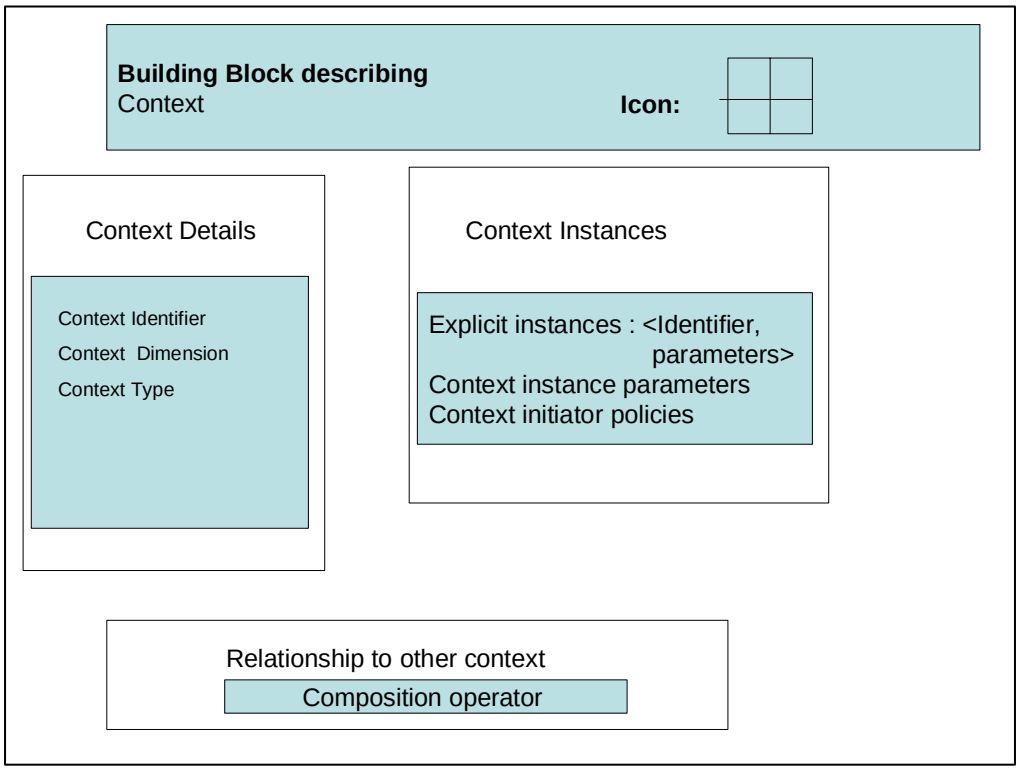


Figure 7.1. The context definition element

Each context has an *identifier*, and a *dimension* which is one of the following: temporal, spatial, state-oriented, segmentation-oriented, or composite. The *type* determines the approach used to assign event instances to context instances, and the set of type values depends on the dimension. If the context gives rise to a finite number of instances these instances can be listed in the definition element and given identifiers. This is to allow an EPA to be assigned to a specific instances or set of instances, so that you can specify different processing for different context instances. Each context has a collection of parameters. These parameters are specific to the context type and are discussed in the following sections. Some context types can have a context initiator policy, a concept that is discussed later in this chapter.

Figure 7.2 shows a number of different context types, organized by dimension, and lists the parameters associated with each type.

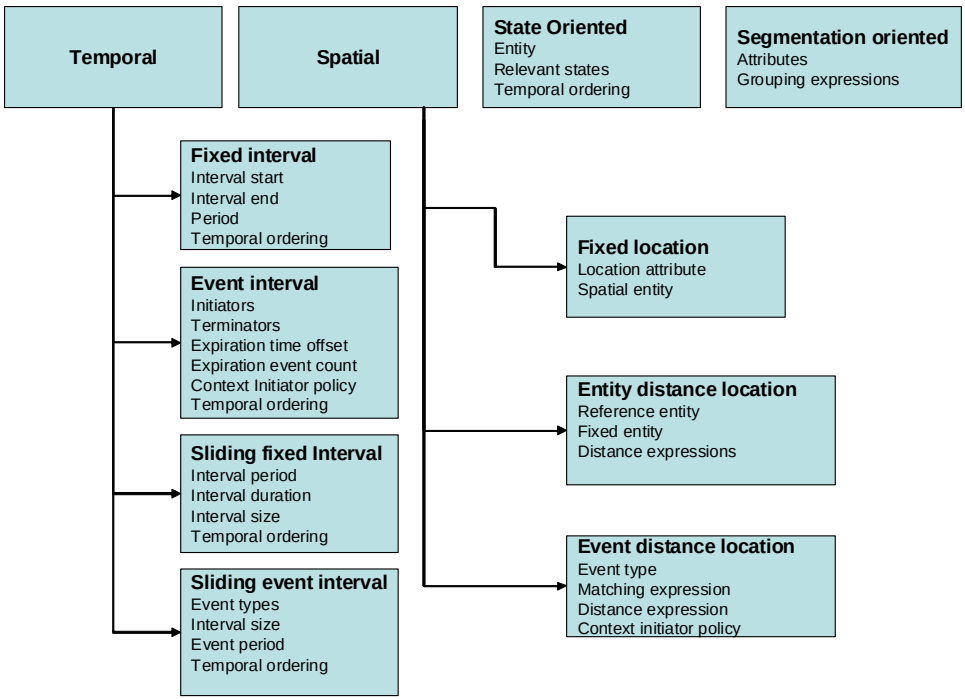


Figure 7.2. The different types of context and their parameters. This diagram also shows the principal dimension associated with a type, for example the Fixed Location type is concerned with the Spatial context dimension

We'll now discuss each of these four dimensions and look at the types of context associated with them.

7.2 Temporal Context

A temporal context divides a stream of event instances and breaks it into one or more instances or groups of consecutive event instances to be processed together. We illustrate this with a couple of examples:

- An EPA that is designed to enforce a regulation that prevents a person from making more than three withdrawals from an ATM machine within a single day. Each day (starting at midnight) is a separate context instance, and the events that occur during that day are associated with that instance.
- If the regulation is modified such that a person cannot withdraw more than three times from an ATM machine within a 24 hour period, then a context instance now

starts whenever a customer withdraws money from an ATM machine and ends 24 hours later.

These instances consist of a set of consecutive events from the stream. The definition of "consecutive" can be based on the value of a timestamp or sequence number in the event instance or it can simply be based on the position of the event instance in the stream. If a timestamp is used it could be either the occurrence time or the detection time of the events.

Figure 7.3 shows some examples of the context instances that arise with different types of temporal context.

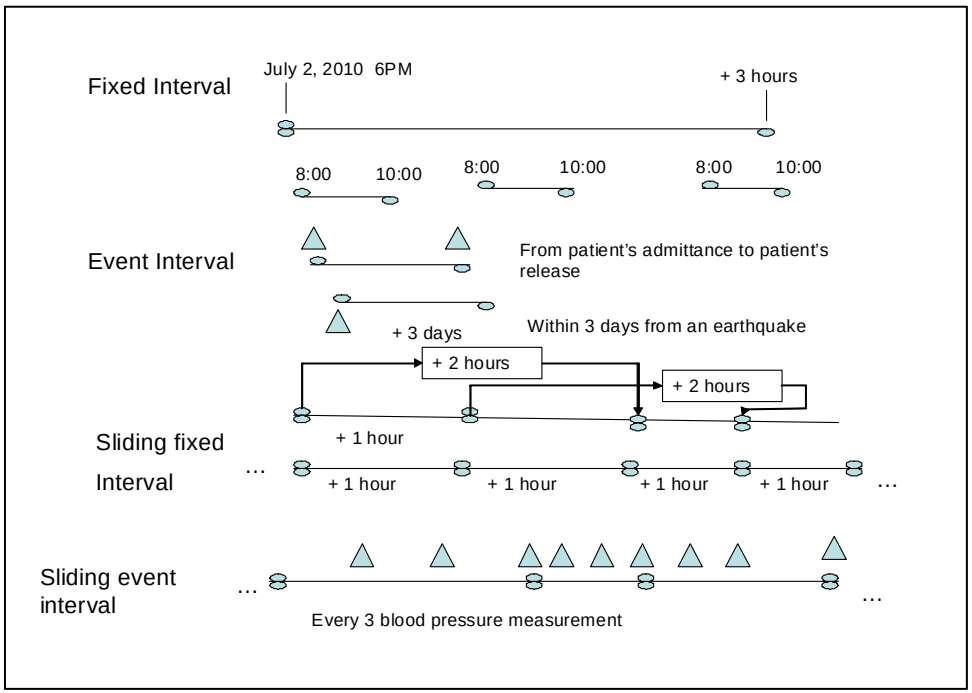


Figure 7.3 Some of the different types of temporal context with their instances

- Looking at figure 7.3 you can see that in some cases the context definition specifies a finite number of instances, sometimes just one. In other cases the definition specifies the conditions under which context instances start and end, and can thus give rise to a potentially unbounded sequence of instances. If there are multiple instances then they can overlap, complement one another, or have gaps between them.

To get a handle on all these options we define a number of temporal context types, whose names are shown in figure 7.3. We will now describe each of these types in turn, but before we do that we need a quick word on terminology. In these sections we will follow we will refer to temporal context instances as windows and use the phrase "opening a window"

or “closing a window” to mean the same as “starting a context instance” or “ending a context instance”.

7.2.1 Fixed interval

The `fixed interval` context is used to represent either a single fixed-length time period, or a fixed-length time period that repeats in a regular fashion, for example the trading hours of a financial market. The intervals do not overlap.

Definition

In a *fixed interval context* each window is an interval that has a fixed time length; there may be just one window or a periodically repeating sequence of windows.

The parameters of this type are:

- Interval start: this may be a fully specified date and time, or a truncated datetime, for instance `Tuesday at 09:30`, or even `10:00`. Truncated datetimes are useful when specifying repeating intervals.
- Interval end: this can be either a date and time, like interval start, or it can be expressed as an offset from the interval start. We use a + sign to indicate an offset.
- Period: this specifies how frequently the interval is to repeat, if at all.
- Temporal ordering: This parameter indicates whether the ordering of elements in the stream is based on their physical position in the stream, or on their detection time or their occurrence time timestamp.

Examples of a single non-repeating interval are:

- *Super Bowl XLIV* (start = 2 July 2010 6pm EDT , end = + 3 hours)
- During the months of July and August 2009. (start = 1 July 2009, end = 31 August 2009).
- Examples of periodically repeating intervals are:
- *Every working day between 8 - 10 AM.* (start = 8:00, end = 10:00, period = day). This is the second example in figure 7.3.
- *Every year during the month of December. (start = December, end = December, period = year).*
- Fixed intervals are used for periodic aggregations, such as aggregation of events to be compared against key performance indicators (KPI). Examples:
- *Calculate total value of purchases from a web store for each hour separately.*
- *Monitor a patient for a fixed time.*
- *Calculate indications about the behavior of the stock market within a single day.*

If an Event Processing Agent is associated with a fixed interval context then it will only receive events that are associated with that context. In cases where the interval repeats, then each window is handled by a different logical instance of the agent; no local agent state is carried forward between these instances.

7.2.2 Event interval

In an `event interval` context, windows are opened or closed by particular events in the event stream. You can see an example in Figure 7.3 where there is a window that is opened by a `patient admittance` event and closed by a `patient release` event.

Definition

In an *event interval context* each window is an interval that starts with the occurrence of an event that satisfies some predicate and terminates with an occurrence of another event that satisfies a predicate, or when a given period has elapsed.

The parameters for an `event interval` context are:

- Initiator event list (event type, [predicate])*; The event interval starts when any of the events specified in the list occurs. An event may be specified just by an event type, in which case any instance of that event type will start the event interval, or it may be specified by the combination of an event type and a predicate expression. If the predicate is present then the event interval will only be started if the event instance also satisfies the predicate (that is to say that the predicate expression must return true when evaluated on the event instance).
- Terminator event list (event type, [predicate])*; The event interval ends when any of the events specified in the terminator list occurs. The terminator list is similar to the initiator list; entries in the list consist of an event type and optionally a predicate expression. If the predicate expression is present, then the event interval will only stop when there is an event instance of the designated type that satisfies the predicate.
- Expiration time offset; The interval will end after this time period has elapsed, even if no terminator event has been encountered. By default this is an offset from the occurrence time of the initiator event, but it can also be specified as an offset from any attribute of the initiator event whose data type is a time stamp.
- Expiration event count; The interval will end after this number of events have been encountered, even if no terminator event has been encountered.
- Context Initiator policy; This parameter specifies what is to happen if a second initiator event is encountered. Context initiator policies are discussed in section 7.6
- Temporal ordering; This parameter indicates whether the ordering of elements in the stream is based on their physical position in the stream, or on their detection time or

their occurrence time timestamp.

Examples:

- From patient admittance to patient release (initiator = patient admittance, terminator = patient release)
- From entrance to the parking lot until exit from the parking lot, but no more than 8 hours (initiator = parking lot entrance, terminator = parking lot exit, expiration offset = + 8 hours)
- Within three days of an earthquake (initiator = earthquake, expiration offset = + 3 days).
- From an `assignment` event to a `delivery` event, with expiration offset of `assignment.Required-delivery-time` + 5 minutes. This example is taken from the Fast Flower Delivery application, see Listing 7.7.
- A book review process, starting with the call for reviews and ending when three reviews have been received. (initiator = Review process start, expiration event count = 4).
-
- The event interval starts with the occurrence of a pre-specified event (called: initiator). This can be a particular event instance, or one of a number of possible event instances. The event interval terminates on the occurrence of a pre-specified event (called a terminator). This can be again a single event instance, or a member of a collection of event instances. The event interval may expire even if a terminator has not arrived, either after some time offset, or when a certain number of events have been detected.

Note that if your application uses special calendar or time-of-day events, then you can use these to initiate or terminate a context.

7.2.3 Sliding fixed interval

In a `sliding fixed interval` context new windows are opened at regular intervals. Unlike the non-sliding fixed interval context these windows are not tied to a particular time of day, instead each window is opened at a specified time after its predecessor. Each window has a fixed size, specified either as a time interval or a count of event instances. In figure 7.3 you can see an example where there is a window opened every hour that lasts for exactly an hour.

Definition

In a *sliding fixed interval* context each window is an interval with fixed temporal size or fixed number of events. New windows are opened at regular intervals relative to one another.

The parameters for a `sliding fixed interval` context are:

- Interval period. The time period that elapses between the start of each window.
- Interval duration. The time period for which each window stays open
- Interval size. The maximum number of event instances to be included in each window
- Temporal ordering: This parameter indicates whether the ordering of elements in the stream is based on their physical position in the stream, or on their detection time or their occurrence time timestamp.

The specification must include an interval period parameter and either an interval duration or interval size (or both).

Sliding intervals may be overlapping or non-overlapping, they are overlapping if and only if the interval period < interval duration. Here are a couple of examples:

- *Start a sliding interval of one hour every 10 minutes (interval period = 10 minutes, interval duration = 1 hour).* In this case there will be six partially overlapping windows at any point in time.
- *Start a sliding window of one hour every hour (Interval period = 1 hour, Interval duration = 1 hour).* In this case there is only one window open at any point in time.

Sliding fixed interval contexts are typically used for aggregation operations, for example counting the number of events of a certain type that occur in the sliding interval.

7.2.4 Sliding event interval

The `sliding event interval` context is similar to the sliding fixed interval context that we just described. The difference is that the criterion for opening a new window is specified as a count of events, rather than as a time period. In figure 7.3 you can see an example where each group of three successive blood pressure measurements is assigned into a new window.

Definition

A *sliding event interval* is an interval of fixed number of event instances that continuously slides on the time axis.

The sliding event interval context gives rise to windows that consist of a fixed number of event instances. Windows are continually started (how frequently this happens depends on the event period parameter) so it's possible to have back to back windows, one after the

other, or to have overlapping windows. A sliding event interval is typically used for aggregation purposes.

The parameters for a `sliding event interval` context are:

- Event types; the set of event types that count towards the interval size and event period. Note that the stream may include event instances whose types are not in this list. These events are included in the window but do not count towards the window size.
- Interval size: this determines the size of each window. It is specified as the number of event instances (of the types listed in the event type's parameter) that are to be included in the window.
- Event period: the number of event instances (of the types listed in the event types parameter) encountered by the current window before a new window is to be opened. If not given it defaults to the interval size, which means that a new window is opened each time the previous window closes.
- Temporal ordering: indicates whether the ordering of elements in the stream is based on their physical position in the stream, or on their detection time or their occurrence time.

Some sliding event interval examples:

- Every three blood pressure measurements (aggregate to see trend for physician report). In this case the physician wishes to see a rolling average of every three readings so, unlike the example in figure 7.3 a new window is opened for each measurement (event period count = 1; Interval size = 3). This means that each blood pressure measurement is aggregated three times with three other measurements.
- Every 100 flights (aggregate for aircraft amortization management). A new window is started every 100 flights, but each flight is included in only one aggregation. (interval size = 100, event period count = 100).

Temporal context is the most widely-used context in event processing due to the fact that many languages support some kind of time window, but we observe a growing number of applications that make use of event location. The idea of location awareness is part of the general spatial context concept, which we will discuss next.

7.3 Spatial context

Spatial Contexts group a stream of events instances according to geospatial characteristics. This type of context assumes that the event payload or header contains an attribute that identifies its location, for example an attribute with a location data type (see chapter 4). An event instance is only classified into a spatial context instance if it contains such an attribute, events without such a location attribute are not classified into spatial contexts.

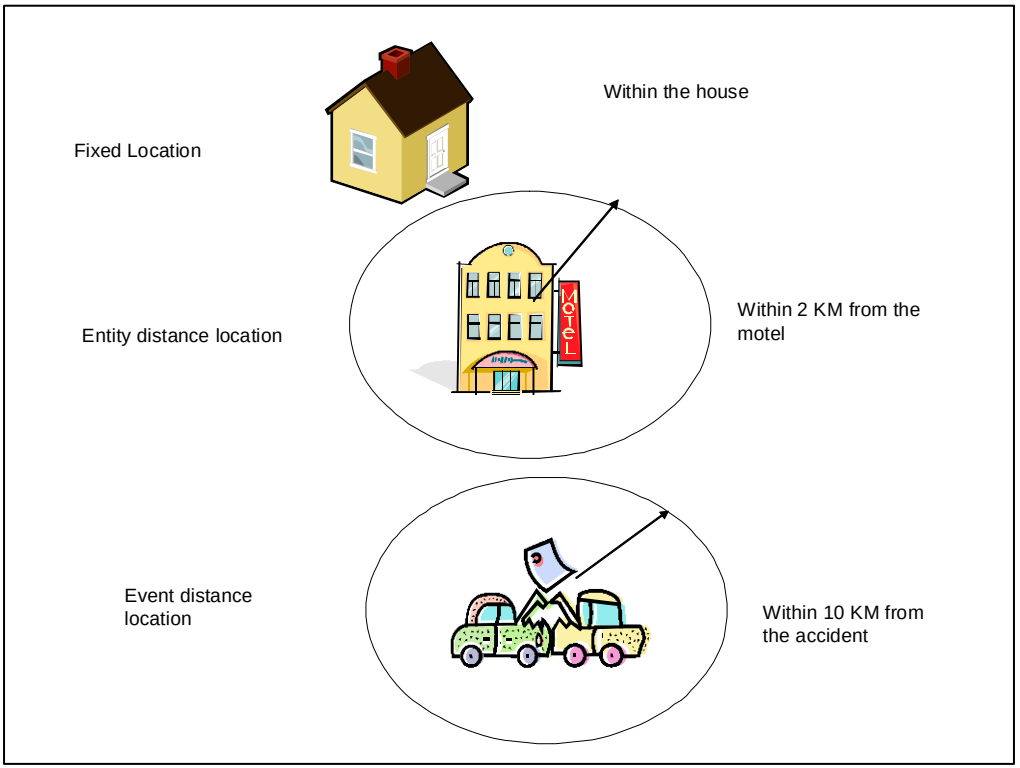


Figure 7.4 Some examples of spatial context

As few can see in figure 7.4, there are several types of spatial context: `fixed location`, `entity distance location`, and `event distance location`.

7.3.1 Fixed Location

A `fixed location` context has a single context instance which is a geometrical object, sometimes referred to as a *geofence*. An event instance is classified into the context instance if its location indicates that it lies within the bounds of this object, or if the event indicates the passage of an entity into or out of the object.

Definition

A *fixed location* context has a single predefined context instance based on event location. The event location is either determined directly by the value of a location attribute or by mapping of a location attribute to another spatial entity.

The parameters for the `fixed location` context are:

- The name of the event attribute that gives the event's location
- Spatial entity (taken from the global state).

Examples of a fixed location context:

- There are various events related to building such as: *entering a building*, *exiting a building*, and parking *in front of a building*. In all these events, building is a location attribute of data type area. All events that have the same value of building belong to the same grouping (location attribute = building)
- The events `entering a building` and `exiting a building` have a point location attribute that corresponds to the door through which the entry or exit occurred. The doors are mapped to the building, and again all events that relate to this building belong to the same grouping (location attribute = entrance, spatial entity = building). In this case we assume that there is some mapping service that maps the entrance to the building.

7.3.2 Entity distance location

An `entity distance location` context gives rise to one or more context instances, based on the distance of an event's location from some other entity. This entity may be either stationary or moving. In the case of a moving entity, the distance relates to the location of the entity at the time that the event occurred (occurrence time). The entity may either be one that is referenced by an event attribute, or a fixed one specified in the context definition

Definition

An *Entity distance location* context assigns events to context instances based on their distance from an entity location that is either specified by an event attribute or is a fixed entity.

The parameters for the `entity distance location` context are:

- The name of the event attribute that gives the event's location

- Attribute that refers to the entity, or
- Fixed entity
- Distance expressions

Examples of entity distance location:

- Vehicle breakdown events are partitioned according to their distance from a particular service center: they are grouped by distance as follows: less than 10 km, between 10 km – 30 km, between 30 km – 60 km and more than 60 km. In this case the service center is a fixed entity specified in the context definition, and there may be a different EPA handling each group (fixed entity = service center, distance expressions = <10 km; ≥ 10 km and < 30 km; ≥ 30 km and < 60 km; ≥60 km). Each of these conditions defines a separate context instance.
- At a big conference, a person uses a location service to generate alerts when based on proximity of other people. Alerts can be set according to the distance of the specified person, e.g. an alert for persons in list A are issued when they enter a distance of less than 100 meters, an alert for persons in list B are issued when they move more than 100 meters away. This relates to moving entities, and the entities are specified as an attribute in the event's payload (Reference entity = List A, Distance expression = <100 m); (Reference entity = List B, Distance expression = > 100 m).

7.3.3 Event distance location

This type of context specifies an event type and a matching expression predicate. If an event occurrence is detected that matches this predicate, then a new instance is created and subsequent events are then included in the context instance if they occurred within a specific distance of the initiating event.

Definition

An *Event distance location* context assigns events to context instances based on their distance from the location of another event.

The parameters for the `event distance location` context are:

- Event type
- Matching expression predicate
- Distance expression
- Context initiator policy – see section 7.6

Examples:

- Delivery trucks arriving to 10 km distance from an accident (Event type = accident,

Matching expression = all, Distance expression = < 10 km).

- A case of an infectious disease is detected within a distance of 100KM from a previous detection of this disease (Event type = Disease, Matching expression = disease type, distance = < 100 km).

Temporal and spatial contexts group events based on data from the events themselves. We now look at another kind of context, state-oriented context.

7.4 State Oriented Context

State-oriented context is the third of our four context dimensions. It differs from the other dimensions in that the context is determined by the state of some entity that is external to the event processing system. This is best illustrated with some examples:

- An airport security system could have a `threat level` status taking values: green, blue, yellow, orange, or red. Some events may need to be monitored only when the `threat level` is orange or above, while other events may be processed differently in different threat levels (Entity = Threat Level, relevant states = all).
- Traffic in a certain highway has several status values: traffic flowing, traffic is slow, traffic jams. Some events are monitored only during traffic jams, in order to reroute vehicles, or alert people to expect late arrivals (Entity = Traffic on highway A3, relevant states = traffic jam).
- There is only one state oriented context type and it is defined as follows:

Definition

In *State oriented context* events are grouped of based on a state of an external entity that is in effect when the event occurs or is detected (according to the temporal order of this context)

The parameters for `state oriented` context are:

- Entity
- Relevant states
- Temporal ordering: Indicates whether an event is classified based on the value of the state at the event instance's occurrence time or at its detection time.

If an EPA is associated with a state oriented context then it only processes incoming events if the given entity is in one of the states specified by the *relevant states* parameter.

The last dimension left for us to discuss is segmentation oriented context.

7.5 Segmentation oriented context

Segmentation oriented context is used to group event instances into context instances based on the value of some attribute or collection of attributes in the instances themselves. As a simple example suppose that each event in the stream contains the same `customer-identifier` attribute. The value of this attribute can be used to group events so that there's a separate context instance for each customer. Each context instance only contains events related to that customer, so that the behavior of each customer can be tracked independently of the other customers. Alternatively, a segment oriented context definition can include one or more predicate expressions relating to one or more of the event attributes. Each predicate corresponds to a context instance; an event is assigned to a context instance if the corresponding predicate evaluates to true.

Definition

A *Segmentation-oriented context* assigns events to context instances based on the values of one or event attributes, either using the value of these attribute(s) or using predicate expressions to define context instance membership.

There is only one type of Segmentation-oriented context. Its parameters are:

- Attributes: List of one or more attributes
- Grouping expressions: List of zero or more grouping predicates

The grouping can be based on one of the following grouping options:

- The context instances are partitioned by the value of a single attribute, or by a combination of values of attributes. Examples: if the attribute is `driver`, then events are grouped according to the value of the driver attribute, so each driver has its own group; if the attributes list consists of driver and store, then events are grouped according to the combination of values of driver and store.
- The context instances are partitioned by having grouping expression that specifies a predicate for each grouping. An example: the partition is based on age; ages are grouped by the following predicates (age is below 21, age between 21-30, age between 30-50, age between 50–67, and age above 67). Again each age group can have separate processing In this examples the predicate expressions would be age < 21; age ≥ 21 and age < 30; age ≥ 30 and age < 50; age ≥ 50 and age < 67; age ≥ 67. In this example all predicates refer to a single attribute, however, predicates may refer to multiple attributes.
- We have now completed our review of the four context dimensions and the context types associated with them. In this discussion we have made of mention of something called a context initiator policy, and we turn to look at that next.

7.6 Context initiator policies

We need to tune up the semantics for a couple of the context types that we have mentioned. These are types in which event instances determine the boundaries of the context instances. If this kind of context is used, there is the possibility that several context initiator events may occur over the time that a context instance is active.

Recall the event interval subtype where a new window is opened when a particular event occurs and consider the example given in section 7.2.2:

> *Within three days of an earthquake (initiator = earthquake, terminator = earthquake + 3 days).*

Assume that there is an earthquake at 10:00AM on *May 5, 2007 10:00AM.* This would establish a window corresponding to the *in*terval [M*ay 5, 2007 10:00, May 8, 2007 10:00].* However suppose that another earthquake event happens at 06:00 on May 7, 6:00. This lies *in the middle of t*his *inter*val. We need to specify how this case should be handled, and to do this we introduce the idea of the context initiator policy.

Definition

A *context initiator policy* is a semantic abstraction that defines the behavior required when a window has been opened and a subsequent initiator event is detected. The possible policies are: open another window, ignore the new initiator event, refresh the window or extend the window.

In the event interval case, the effects of the various context initiator policies are:

- Add: returning to the earthquake example, another window will be added with the interval [May 7, 6:00, May 10, 6:00], while the original window is still open. The new earthquake event will be assigned to both windows.
- Ignore: the original window will be preserved. The new earthquake event will be added to the original window, and no new window will be opened.
- Refresh: The original window will be closed, and a new window will be opened.
- Extend: The new earthquake event will be added to the original window. If this window has an offset time terminator, the offset is reset to take account of the new initiator event.
- Figure 7.5 illustrates these policies:

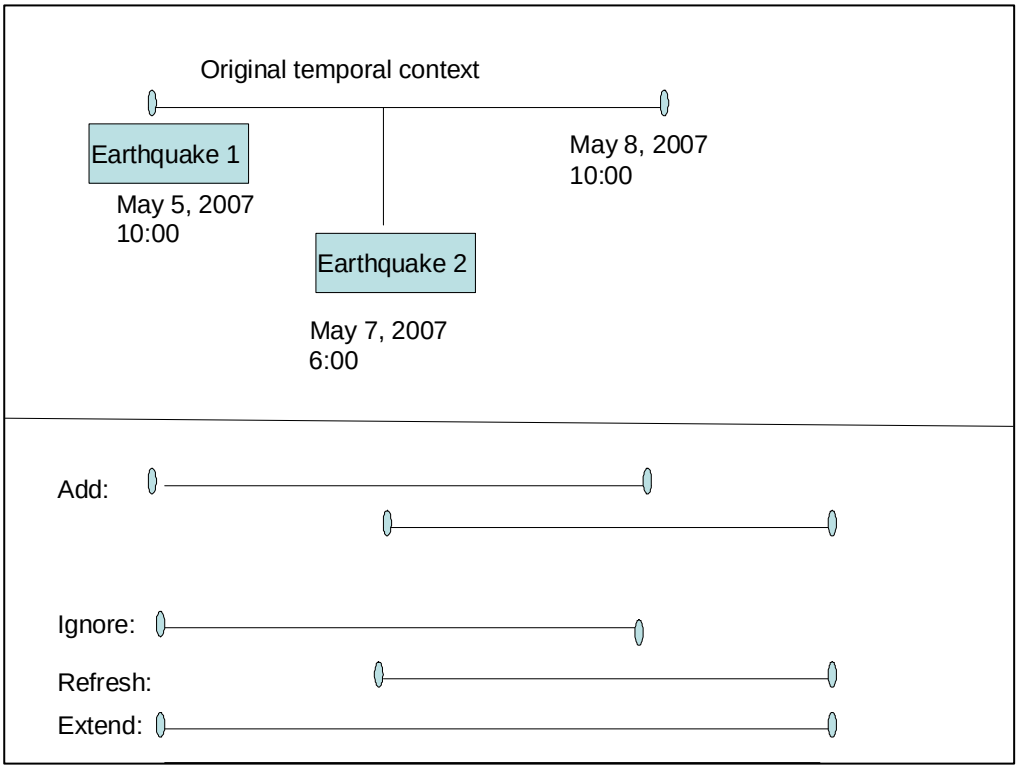


Figure 7.5 The various context initiator policy options.

This policy also applies to the event distance location context subtype. Recall the following example:

- Delivery trucks entering a zone 10 km distance from an accident (Event type = accident, Matching expression = all, Distance expression = < 10 km).

Suppose that a context instance has been established for this context and then a second accident event occurs, which should, in principle, open the same context instance again. The possibilities are:

- Add: Another context related to the new accident will be added to the spatial context.
- Ignore: The second accident is ignored; the focus is on the original accident only.
- Refresh: The location in this context always relates to the most recent accident.
- Extend: The context instance is extended to be the union of 10 km distance from both locations.

Each context type that we have discussed so far belong to just one of the four context dimensions. However in practice many applications use contexts that involve multiple dimensions. We refer to these as composite contexts.

7.7. Composite contexts

Event processing applications often use combinations of two or more of the simple context types that we have discussed so far. In particular a temporal context type is frequently used in combination with a segmentation context, and we show and example of this in figure 7.6.

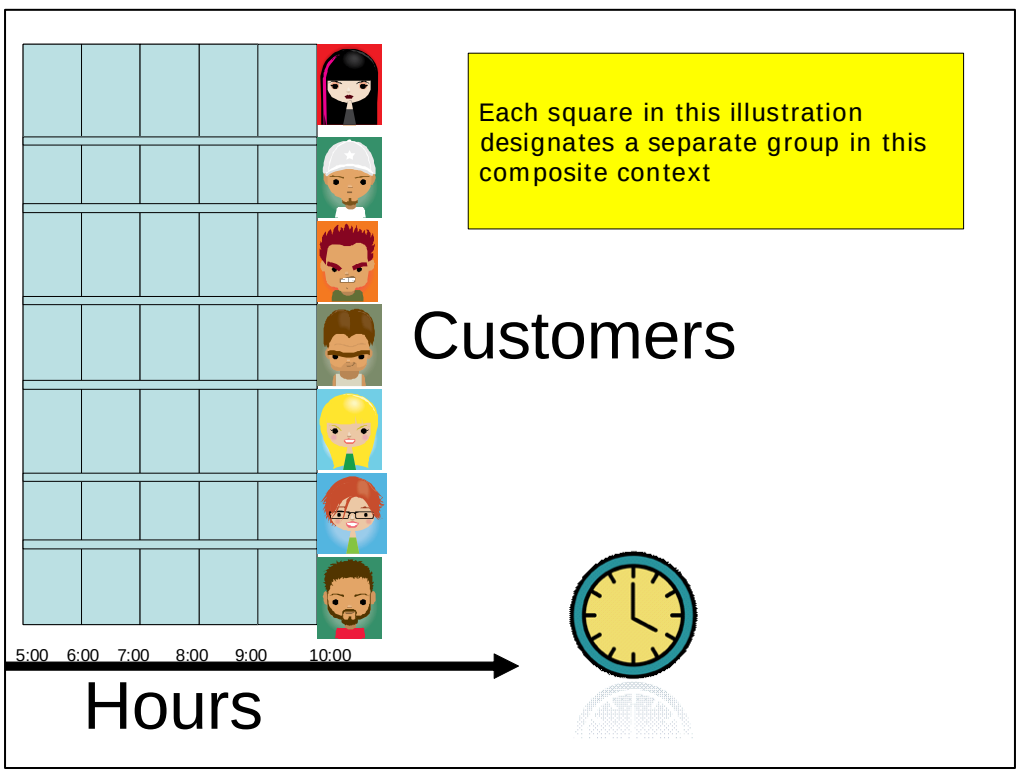


Figure 7.6 An example showing composition of segmentation context and temporal context. Each square is a separate context instance and designates the combination of an hour-long interval and a specific customer

Examples such as this one are covered by the following definition:

Definition

A composite context is a context that is composed from two or more contexts. The set of context instances for the composite context is the Cartesian product of the instance sets of its constituent contexts.

With two constituent contexts, the composition process works like this. Event instances are first classified into context instances using one of the two contexts. Each of these context instances is then further subdivided using the other context. This process is then repeated if there are more than two contexts involved. The contexts that are composed together may be of any type, and in practice they are often of different types as the following examples show:

- Composition of segmentation context and temporal context: This is illustrated in figure 7.6 where the segmentation context groups event instances by customer and the temporal context is a sliding fixed context with both duration and frequency of 1 hour. In the resulting composite context each context instance groups together the events that relate to a single customer for a single hour.
- Composition of spatial context and state context: In this example, imagine that the spatial context is fixed location relating to the city of Trento in Italy and the state context relates to the state of the weather (clear, cloudy, rain, snow). The composite context has four instances, one for each state of the weather applied to events that occur within the city of Trento.
- Composition of segmentation context and spatial context: In this example suppose that the segmentation context relates to car type and the spatial context is the distance from Malpensa Airport (near Milan, Italy). If the segmentation context has twenty different car types and the spatial context has three context instances (for example for distances 0-5KM, 5-10KM, 10-30KM) then the composite context will have sixty instances, one for every combination of car type and distance range.

These are three examples of composite contexts, all of them multidimensional. The first example combines segmentation-oriented (by customers) with fixed sliding temporal context (by hour); every combination of hour and customer creates another context instance. The second composite context combines state oriented (weather) and fixed location (neighborhoods). The third composite event combines segmentation-oriented context and spatial context.

In the next section we'll wrap up this chapter by showing how contexts are used in the Fast Flower Delivery example.

7.8 Contexts in the Fast Flower Delivery example

The following definition elements describe contexts that are used in the Fast Flower Delivery example.

Listing 7.1 The Driver Context

Context Identifier	Context dimension	Context type	Composition
Driver	Segmentation oriented	Segmentation	

Context instance parameter Attribute	Parameter Value
Driver	

This is a segmentation context that groups events by the driver

Listing 7.2 The Request context

Context Identifier	Context dimension	Context type	Composition
Request context	Segmentation oriented	Segmentation	

Context instance parameter Attribute	Parameter Value
Request Id	

This segmentation context groups events by request id

Listing 7.3 Monthly context

Context Identifier	Context dimension	Context type	Composition
Monthly	Temporal	Sliding Fixed Interval	

Context instance parameter	Parameter Value
Interval Period	Month
Interval Duration	Month

This context assigns events into contiguous month-long windows.

Listing 7.4 Bid Interval context

Context Identifier	Context dimension	Context type	Composition
Bid Interval	Temporal	Event Interval	

Context instance parameter	Parameter Value
Initiator	Bid Request
Terminator	+ 2 Minutes
Initiator policy	Ignore

This event interval temporal context opens a window each time that a bid request is issued; the window closes after two minutes.

Listing 7.5 Response Interval

```
Context Identifier      Context dimension      Context type        Composition
Response Interval       Temporal               Event Interval

Context instance parameter      Parameter Value
Initiator                       Bid Interval Termination
Terminator                       + 1 Minutes
Initiator policy                ignore
```

This context opens a context instance each time that a bid request is terminated; it ends after one minute.

Listing 7.6 Pick up interval

```
.
Context Identifier      Context dimension      Context type        Composition
Pick up interval        Temporal               Event Interval

Context instance parameter      Parameter Value
Initiator                       Delivery-Bid
Terminator                      End-of-Day event
Expiration offset
Initiator policy                Ignore
```

This context denotes the time interval in which pick up is expected. It expires when the time-out arrives.

Listing 7.7 Delivery interval

```
Context Identifier      Context dimension      Context type        Composition
Delivery interval       Temporal               Event Interval

Context instance parameter      Parameter Value
Initiator                       Assignment
Terminator                      Delivery Confirmation
Expiration offset               Assignment. Required Delivery time + 5 Mins.
Initiator policy                ignore
```

This context denotes the time interval in which delivery is expected. It expires when the time-out arrives.

Listing 7.8 Driver evaluation context

```
Context            Context          Context       Composition
Identifier         dimension        type
Driver                                            Segmentation = driver
Evaluation                                        temporal = confirmations
```

```
Context              Context            Context type          Composition
Identifier           dimension
confirmations        Temporal           Sliding event
                                        interval

Context instance parameter     Parameter Value
Event Types                    Delivery Confirmation
Event Count                     20
```

This context groups the Delivery Confirmation events for each driver into group of 20 events.

Listing 7.9 Active driver activity

```
Context          Context          Context       Composition
Identifier       dimension        type
Driver                                          Segmentation = driver
Activity                                        temporal = Daily when
                                                active

Context Identifier     Context dimension     Context type       Composition
Daily when active      Temporal              Event Interval

Context instance parameter     Parameter Value
Initiator                      Bid
Terminator                     Pick-Up Confirmation
Expiration offset              Assignment. Required Pick-Up Time + 5 Mins.
Initiator policy               Ignore
```

This context opens a context instance for every driver that has done at least one bid at that day.

Listing 7.10 Monthly context

```
Context              Context            Context type          Composition
Identifier           dimension
Monthly              Temporal           Sliding Fixed
                                        interval

Context instance parameter     Parameter Value
Interval Period                Month
Interval Duration              Month
```

This is a long-term fixed sliding temporal context that lasts for a month.

Listing 7.11 Monthly driver

```
Context          Context          Context       Composition
```

```
Identifier          dimension          type
Monthly             Composite                        Segmentation = Driver,
Driver                                               Temporal = Monthly
```

This context instances the relevant events according to the combination of month and driver

These are the ten contexts used in the Fast Flower Delivery example. In the next couple of chapters we'll see how they are used in conjunction with various types of EPA.

7.9 Context definitions in practice

The notion of time window exists in several languages. The CCL language (Aleri/Coral8) has some notion of window with keep policies that control the type of the window. Listing 7.12 provides a window example, with a sample of keep policies

Listing 7.12 CCL example for Window and keep policies

```
Windows are like streams, but store records
CREATE WINDOW Book_w SCHEMA Book_t KEEP ALL;
INSERT INTO Book_w
  SELECT * FROM Book_s;

Sample of KEEP policies (there are others):
KEEP LAST PER Id
KEEP 3 MINUTES
KEEP EVERY 3 MINUTES
KEEP UNTIL (”MON 17:00:00”)
KEEP 10 ROWS
KEEP LAST ROW
KEEP 10 ROWS PER Symbol
```

Note that the notion of CCL window has many of the functions of context we have discussed before, including several types of temporal contexts, and a composed context by segmentation; for more information about the semantics of keep policies in CCL, the reader is referred to the book's website.

Figure 7.7 shows an example definition of a temporal segmentation using AMiT[1]

---

[1] The AMiT language is not referred by the book's website, for further information about the AMiT language, the reader is referred to: Asaf Adi, Opher Etzion: Amit - the situation manager. VLDB J. (VLDB) 13(2):177-203 (2004) http://www.springerlink.com/content/nb1qa1d02vvdre00/

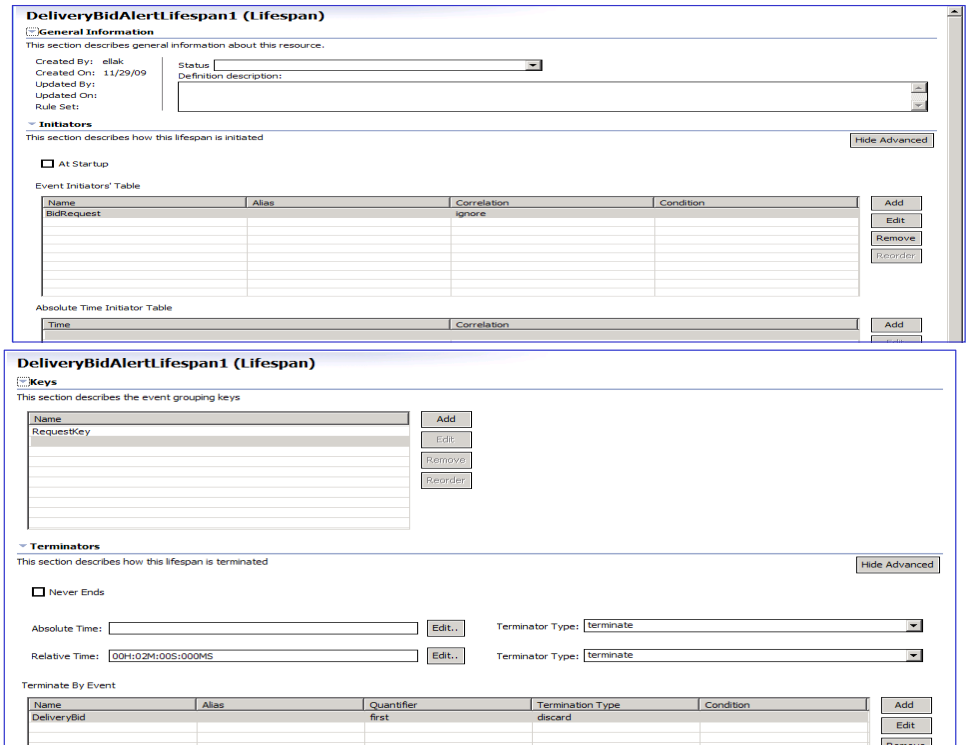


Figure 7.7 An example definition of temporal segmentation using AMiT. Note: AMiT is not available through the book's website.

In this example, there is a definition of event initiator in the highest frame, followed by definition of grouping key (segmentation context) and the terminator as an offset of 2:00 minutes. Various other languages support some of the context options in various forms. For deeper dive into various languages, the reader is referred to the book's website.

7.10 Summary

In this chapter we have discussed what we mean by context and how it is used in event processing applications. We have looked at the four dimensions of context: temporal-sequential, spatial, state oriented and segmentation oriented, and have examined several different context types, both single-dimensional and multi-dimensional. These concepts were demonstrated using the contexts of the Fast Flower delivery example.

In the chapters that follow we will look in more detail at how context is used with various types of event processing agent. Chapter 8 looks at filtering and transformation EPAs, and chapter 9 at pattern matching.

Additional reading

Asaf Adi, Opher Etzion: Amit - the situation manager. VLDB J. (VLDB) 13(2):177-203 (2004)
http://www.springerlink.com/content/nb1qa1d02vvdre00/

This article, mentioned before, contains an early definition of context.

Sharma Chakravarthy, Qingchun Jiang: Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing, Springer 2009.
http://www.amazon.com/Stream-Data-Processing-Perspective-Scheduling/dp/0387710027/ref=sr_1_1?ie=UTF8&s=books&qid=1259477906&sr=1-1

This book provides introduction to stream processing, and discusses the notion of window.

Exercises

7.1 Can you find other applications of the notion of context, besides event processing? Provide concrete examples.

7.2 Give an example event processing application that uses all the types of spatial context.

7.3 Give an example of a composite context which uses union of contexts and difference of contexts in addition to the more typical intersection of contexts

7.4 Looking at context composition of several contexts of type temporal interval, what other operators can be defined between such intervals?

7.5. Looking at context composition of several contexts of type spatial distance, what other operators can be defined by such contexts?

7.6 Take a specific stream processing language and show how its windowing expressions map to the context types discussed in this chapter.

7.7 Provide an example that employs spatio-temporal context that is the combination of spatial and temporal contexts.

7.8 Provide an example of interesting use of state-oriented context, listing the cases in which different behavior is required in different states.

8

Filtering and transformation

"There are painters who transform the sun to a yellow spot, but there are others who with the help of their art and their intelligence transform a yellow spot into the sun"

- Pablo Picasso

We now return to the examination of event processing networks that we started in chapter 6, to take a deeper look at the event processing agents (EPAs) that an EPN can contain. The most important kinds of EPA are `filter`, `transformation` and `pattern detection` agents, and in this chapter we will look at the first two of these. Pattern detection is a large topic, so we dedicate the whole of the next chapter to that. We will also, as promised in the chapter 7, look at the general question of how context is applied to EPA's of all kinds.

In this chapter we will cover

- The idea of a filter, the places where filtering can be performed in an Event Processing Agent, and languages used to specify filter expressions.
- The use of event context to perform filtering
- The different types of stateful and stateless transformation
- The effect of filtering and transformation on event header elements

As usual we will illustrate this with examples from the Fast Flower Delivery application and some examples from other fictional applications. We will start with the discussion of filtering and then move on to transformation later in the chapter.

8.1 Filtering in the Event Processing Network

Many event processing applications perform some kind of event filtering. An application could have event producers, such as sensors or news feeds, which ingest large numbers of events not all of which are relevant, and so it needs to filter out the irrelevant ones. Similarly an

application might have multiple consumers all interested in different things so the application needs to filter what it sends to each consumer. Also if the application is performing some kind of pattern detection (which we will look at it in the next chapter) it may need to filter the events it receives before passing them to the pattern detection step.

There are three places where we can specify filtering operations in an event processing network definition:

- On the input terminals of any Event Processing Agent or Event Consumer
- As part of the definition of certain Event Processing Agent types. In particular there is the Filter EPA type which is specifically dedicated to the task of event filtering
- As part of an Event Processing Context definition

In all these cases a filter is an operation which takes an event instance as input and then decides whether that instance is to be selected for further processing or not. We sometimes say that an instance that is selected has "passed" the filter, or has been "filtered in," whereas an instance that has not been selected has "failed" or has been "filtered out". We will look at each of these three cases in turn.

8.1.1 Filtering on an input terminal

Figure 8.1 shows filtering on an input terminal. In this illustration we have represented the different events emitted by the producer as geometric objects, and have shown an English-language filter expression "allow only triangle and square events". We will look at more formal real-life filter expressions in a minute.

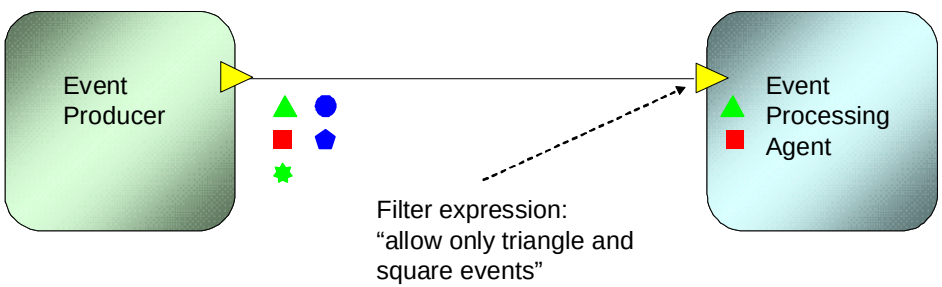


Figure 8.1 The Event Processing Agent has a filter expression on its input terminal that only allows in a subset of the events emitted by the Event Producer.

If a filter operation is attached to an input terminal, then only event instances that pass the filter actually make their way though the terminal, any other event instances are filtered out. There are several reasons why you might want to specify a filter operation on an input terminal:

- It keeps the specification of the filter separate from the rest of the EPA definition. This means that the actual filtering process could be implemented independently of the EPA implementation. For example filtering could be performed by the channel that transmits events from the producer to the consumer – or even by the producer itself.
- You will recall that we do not specify the internal behavior of an Event Consumer, so the input terminal is the only place on an event consumer where you can specify a filter.
- Some event processing agents have more than one input terminal, so you can specify different filter operations for each terminal

The input terminal's filter operation is specified by means of one or more filter expressions. Each filter expression takes the form of a Boolean function that can be evaluated each time an event is received by the input terminal. If the function returns *true* this means that the event should be been passed by the filter, if it returns *false* then the event has failed. If there's more than one expression attached to the terminal then all expressions must evaluate to *true* before an event can pass through the terminal. This means that effect of combining multiple filter expressions is equivalent to evaluating each one in turn and then logically conjuncting ("ANDing") their results. You will see in a minute that each filter operation in such a compound operation is logically independent from all the others so they can be evaluated in any order and you will still get the same result.

There are several different kinds of filter expression that can be attached to an input terminal, and it's possible to mix these different kinds of expression on a single terminal:

- An `event type` filter expression lists one or more event types (we described event types in chapter 3). The expression evaluates to true if the incoming event is an instance of any of these types.
- An `event header` filter expression is a function computed from the values of header attributes of the event instance.
- An `event content` filter expression is a function computed from the values of payload attributes of the event instance.

An event content filter is essentially a collection of attribute name/value assertions, for example `(attributeA = valueX) && (attributeB > valueC)`. Different event processing languages use different variations of syntax, with differing degrees of expressive

power, so in keeping with our principle of implementation neutrality we will describe a filter expression language that is based on the W3C XPATH[1] standard.

You may be familiar with XPATH as a language used to navigate around an XML document, or with its use as part of the XSLT language that's used to transform XML documents, and may now be wondering how it can be used to specify an event filter operation. Here's how it's done:

We start by viewing the event instance as a pair of XML documents – one for the header attributes and one for the payload. Note that we are only doing this for the purpose of explaining how the filter definition works – an implementation does not actually have to hold its event instances as real XML documents. The event instance's header attributes correspond to the top-level elements of the header XML document (the immediate children of the root element of that document), and the payload attributes correspond to the top-level elements of the payload document. The name of each XML element matches the name of the event attribute, and if the event attribute has a complex data type then the corresponding XML element contains child elements representing the structure of that complex data type.

If you have used the XPATH language you will have encountered XPATH path expressions. These are expressions of the form `/chapter/paragraph` that are used to point to elements or attributes in the XML document. In fact the XPATH language defines them to be functions that return "XML node-sets"[2], and we can use a path expression as a filter by interpreting the result as *false* if the node-set is empty, and *true* otherwise. As we have set up a correspondence between elements in our notional XML document and attributes in the event instance, a path expression can be used to test for the presence of one or more attributes in the event instance. This is best explained using some examples.

```
/Driver
```

A simple path expression like this can be used as a `content filter` to test that the payload contains a `Driver` attribute. If there's an attribute of this name (whatever its value), the expression returns *true* and the filter passes the event. A path expression can also contain multiple steps, like this:

```
/Location/Latitude
```

Expressions like this can be used if the event type is expected to contain structured attributes. In this example we are checking that the event instance contains an attribute

---

[1] There are two versions of XPATH, the older XPATH 1.0 version being the more widely used at the time of writing. Most of our discussion will assume XPATH 1.0 though, as we will see, the newer XPATH 2.0 has some advantages when it comes to handling dates and times.

[2] A "node-set" is a collection of nodes from the document. XPATH defines seven types of node (root, element, attribute, text, namespace, procedural instruction, comment) but it's chiefly element nodes that we are interested in here

called `Location` and that it has a child attribute called `Latitude`. You can also test events which contain attribute arrays with filters like this:

```
/Itinerary/Point [7]
```

This expression filters out any event that doesn't contain an `Itinerary` attribute with at least seven `Points` in it. Path expressions can also be combined together with an | operator:

```
/Driver | /Store
```

This causes an event to be passed if either of the two expressions is satisfied. In this example an event will be passed if it contains either a `Driver` attribute or a `Store` attribute (or both)[3].

You might not need to use path expressions like these if you already know the event type (for example because you have included an event type filter) and if that event type completely dictates the shape of the message. However you can have event types that allow some attributes to be optional, or that permit lists containing a variable number of entries and a path expression allows you to check that the attributes you need really are there. They are also useful when filtering a stream that contains multiple event types; our first example will pass any selects event instance that contains a `Driver` attribute, regardless of its type.

The XPATH language also allows you test the value of an XML node, so we can use that to filter events based on the values of one or more of their attributes.

```
/Ranking > 5
```

This is a simple numeric comparisons it selects events that contain a `Ranking` attribute that has a value greater than 5. An event instance would fail the filter if it had a `Ranking` value less than or equal to 5, and it would also fail if had an attribute called `Ranking` which had a non-numeric value (for example a character string like "High") or if it had no attribute called `Ranking` at all.

```
/Store = "Exotic Flowers"
```

This selects events that have a character string attribute called `Store` with the value "Exotic Flowers". If the event type contains structured attributes, then you can use filters to select events based on the contents of some part of these attributes:

```
/Location/Latitude > 0
```

---

[3] Recall that if you were to supply separate `/Driver` and `/Store` filter expressions, then they would be ANDed together, so that an event instance would only be passed if it contained both attributes.

This selects events which have a `Location` that is in the Northern hemisphere (positive latitude).

```
/Itinerary/Point[7]/Latitude > 0
```

This selects events containing an `Itinerary` attribute whose seventh point is in the Northern hemisphere (the remaining points of course could be in either hemisphere).

The XPATH expressions that we have looked at so far have just been path expressions and comparison expressions. We observed that a path expression returns an XML node-set and we defined the interpretation of the corresponding filter to be such that it passes an event if (and only if) this node-set has one or more nodes in it. Comparison expressions, when evaluated against an event instance, already return the value *true* or *false* so the natural interpretation is to pass an event if the expression returns *true* and fail it otherwise. We can extend this line of reasoning to allow any XPATH expression to be used as a filter. Table 8.1 shows how we do this:

Table 8.1 Interpreting XPATH filter expressions

Expression returns	Success condition	Example
Node-set	Node-set has at least one node in it	/OccurrenceTime
Boolean	Value is *true*	/Ranking > 5
Number	Value is non-zero	25
String	String has non-zero length	“OK”

We aren’t going to go into all the details of the XPATH language or the things can you can do with it, there are plenty of tutorials and reference books available that do that (we suggest one at the end of this chapter). However we will conclude this section with a few examples that show some of the power of the language.

XPATH has a number of built-in functions that return Boolean values and we start with a couple of examples that show this:

```
true()
```

This filter passes every event regardless of its content. A more useful Boolean function is not() which inverts the logical value of its argument.

```
not(/Credit)
```

This passes every event unless it has one or more attributes called `Credit`. The logical operators *and* and *or* are also available and can be used to combine other Boolean expressions:

```
/Credit or /Ranking > 5
```

This example passes any event that contains a `Credit` attribute (recall that `/Credit` by itself would return true in such a case) and also any event that has a `Ranking` attribute with value greater than 5.

The comparisons that we have looked at so far have all compared an attribute value against a literal constant contained in the filter expression, but it is also possible to compare the values of two attributes in the event with each other,

```
/Credit = /Debit
```

The XPATH interpretation of this expression is a little tricky. If the event instance contains exactly one `Credit` attribute and exactly one `Debit` attribute then the event is passed provided that the two attributes have the same value. If the event doesn't have `Credit` or `Debit` attributes then it is filtered out. If the event contains more than one `Credit` or `Debit` attribute then it's sufficient for there to be a match between any one of the `Credit` and any one of the `Debit` attributes for the event to be passed.

Our remaining examples show the use of some more XPATH built-in functions:

```
count(/Credit)-count(/Debit)
```

The count() function returns the number of nodes present in the node-set, so this filter passes an event provided that it has exactly the same number of `Credit` and `Debit` attributes (possibly none).

```
count(/Credit) mod 2
```

This example passes an event if it has an odd number of `Credit` attributes (recall from table 8.1 that if an expression returns a numeric result then the event is passed if that result is non-zero).

```
/Itinerary/Point[last()]/Latitude > 0
```

The last() function can be useful when dealing with variable length arrays. This example passes any event containing an `Itinerary` which ends up in the Northern hemisphere.

XPATH has a number of string handing functions, The next example shows one of them:

```
contains(/Store, "Exotic")
```

This filter passes any event that has a `Store` attribute whose value contains the substring "Exotic". We will conclude our examples by looking at a time comparison. Time handling is rather primitive in XPATH 1.0, but the XPATH 2.0 standard contains a number of useful time functions:

```
op:time-less-than(/OccurrenceTime,xs:time(10:00:00))
```

This tests the OccurrenceTime timestamp and passes an event if it occurred earlier than 10am.

8.1.2 Filtering in an event processing agent

The filter expressions that we have looked at so far are placed on input terminals, and have two common characteristics:

- They are simple pass/fail filters. If an event instance fails the filter test then it is not processed by the EPA or Event Consumer.
- They are stateless, that is to say the process of filtering one incoming event instance does not influence the way that any subsequent event instances are filtered by that terminal.

Some event processing agents involve a further filtering step that takes place logically after any input terminal filtering has been performed. This filtering forms part of the actual EPA logic and are not subject to the constraints that we just listed. These filters can be stateful, and events that fail them can continue to be processed by the EPA as well as those that pass. We will encounter one use of such filtering in the next chapter when we look at pattern detection, but for now we will take a closer look at the explicit `filter` EPA which we introduced in chapter 6, and which is illustrated in figure 8.2.

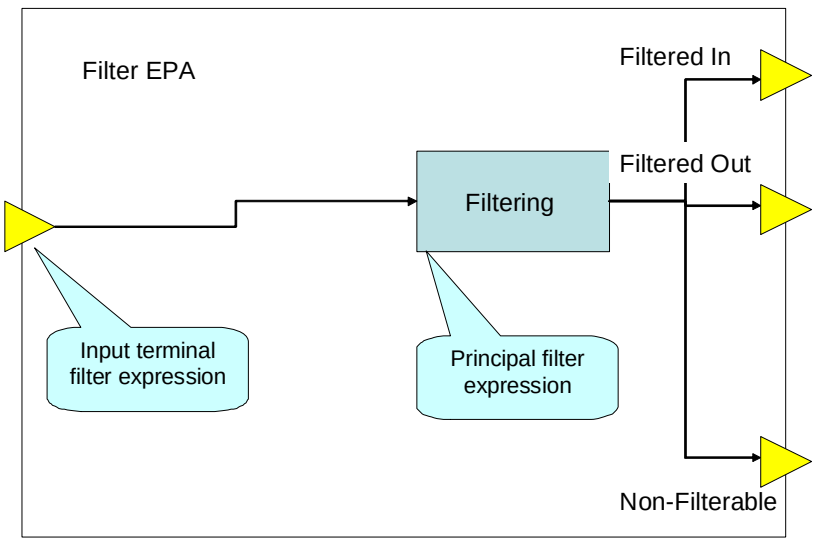


Figure 8.2 A schematic view of a Filter event processing agent

The `filter` EPA has a single input terminal which, like all input terminals, can have a filter expression associated with it. The output from this filter expression is then fed into the EPA's main filtering step, as shown in figure 8.2. The output from this step is then made available via three output terminals. Event instances that pass the filter are, as you might expect, delivered via the `Filtered In` terminal, but in addition event instances that fail the filter are made available via the `Filtered Out` terminal. This means that the Event Processing application designer can use a Filter EPA as a kind of "if / then / else" construct to route event instances to different parts of the Event Processing Network, depending on whether they pass or fail a particular filter[4]. Another way of thinking about it is that the `filter` EPA splits the incoming stream into two sub-streams, one containing the event instances that pass the filter, the other containing those that do not. You can subdivide the stream further by using a second `filter` EPA as shown in figure 8.3.

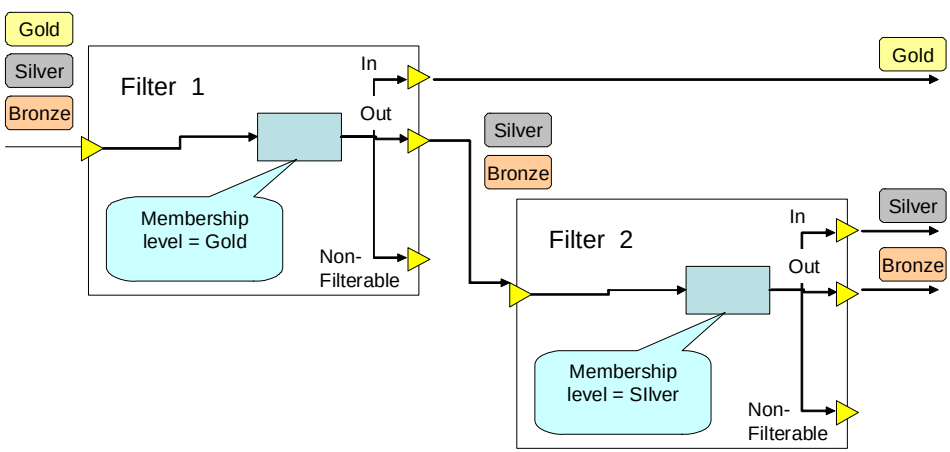


Figure 8.3 Using two successive filter EPAs to divide an incoming event stream into three sub-streams.

This example is taken from an imaginary loyalty card application. The application is processing a stream of incoming events relating to card-holders, and wants to apply different processing logic, depending on the card-holder's membership status[5]. The first filter separates out the Gold customer events from the others; these events are routed through its `Filtered In` terminal. The stream coming out of its `Filtered Out` terminal could

[4] Of course the Event Processing Network designer is not required to connect either the "Filtered In", or "Filtered Out" output terminals to anything else (event instances that are passed to an unconnected output terminal are simply discarded)

[5] In this fictional loyalty card scheme there are just three levels of membership, imaginatively called *Gold*, *Silver* and *Bronze*.

contain a mixture of Silver and Bronze customer events, so the application passes them to a second EPA which separates events relating to Silver customers from those relating to Bronze ones.

You will have seen that the `filter` EPA has an additional output terminal, which we refer to as the `Non-Filterable` terminal. This output terminal is used for the case where the incoming event instance is incompatible with the filter expression and the expression cannot be evaluated as either *true* or *false*, (some filter expression languages allow expressions that can fail when evaluated against certain input events, for example if the input event results in a divide by zero). If this happens then the incoming event is routed to the `Non-Filterable` terminal.

The filter EPA's principal filter expression can be one of the stateless expressions described in section 8.1.1 (topic filter, event header filter or event content filter), but we also permit some simple stateful filters. Stateful filters need a context within which to operate, so in order to discuss them, we need to return to the idea of an Event Processing context which we introduced in chapter 7.

8.1.3 Filtering and event processing contexts

You will recall from chapter 7 that an event processing context groups event instances into one or more subsets called *partitions.* The number of these partitions depends on the context definition:

- Some contexts give rise to just one context partition, for example a spatial context definition that specifies events that occur within a specific building.
- Some contexts can give rise to a fixed number of context partitions, for example a context that separates events based on their location's zip code.
- Some contexts, for example some temporal contexts, don't have a fixed number of partitions; partitions can be created and destroyed dynamically over time.

In all three cases, there might be event instances that are not classified into any of the partitions – particularly in the case where there's only one partition, and there can be event instances that are classified into more than one partition. An event processing agent (or event consumer) can be associated with a context as part of its definition. If you do this then any event instance that doesn't belong to any of the partitions of the context is automatically filtered out. This is true whether it's a `filter` EPA or any other kind of EPA and it is independent of the filter, if any, on the EPA's input terminal. As figure 8.4 shows, context filtering occurs after the input terminal filter and before any other filter step inside the EPA.

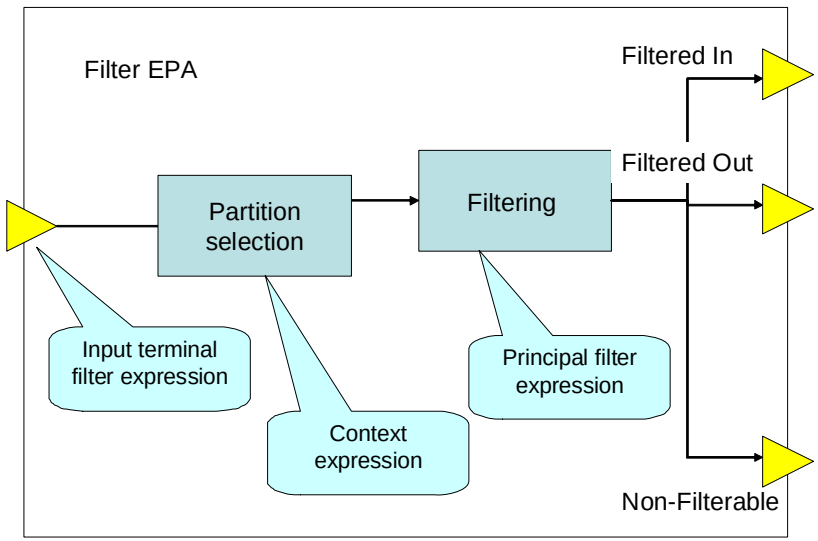


Figure 8.4 A filter EPA showing the logical position of the partition filter, after the Input terminal and before the EPA's internal filter step

This illustration shows the three filter steps that an incoming event has to negotiate in a `filter` EPA. It's important to note that this is purely a *logical* view of what is going on, an implementation is free to compile all the three filter steps together into a single operation provided that this achieves the same logical result.

In cases where an EPA (or event consumer) definition is associated with a context that has more than one partition then there's effectively a separate instance of that EPA (or consumer) associated with each partition. Each instance has an identical specification, for example the same input terminal filter expressions, but each instance only gets to process incoming events that belong to its particular partition of the context.

We will conclude this section by taking a look at how you can associate a context with a `filter` EPA in order to perform stateful filtering. The examples we give here are filters that reduce the volume of incoming events, and they make use of temporal contexts:

- First *m* out of *n* filter. This `filter` EPA has an expression that instructs it to count the incoming event instances and pass only the first *m* instances that it sees. If it is associated with a sliding event window of size *n*, then a new instance of the EPA is created to handle every block of *n* successive event instances, so the filter counter gets set to zero, and the EPA instance passes the first *m* event instances out of the *n* in that block. As an example a first 1 out of 2 filter would pass the first event that it receives, and every other event thereafter.
- Random *m* out of *n* filter. This filter is similar to the previous one, except that its filter

expression tells it to pass a random selection of *m* instances out of its context partition. Using this expression in combination with a sliding event window of size *n* we obtain a filter that passes a random *m* out of every *n* event instances.

- Rate limiting *k* events per second filter. The "pass first m" filter expression can be used in conjunction with a sliding fixed time interval context to ensure that the rate at which events emerge from the `Filtered In` output terminal does not exceed a specified rate. If the rate at which events arrive at the EPA (after passing through any input terminal filter) is less than the specified value of *k* events per second then they are all passed through to the `Filtered In` output terminal, but if it is higher then excess event instances are diverted to the `Filtered Out` terminal.

These filters operate in a rather unsubtle fashion; we will encounter more sophisticated stateful ways of filtering incoming events when we look at pattern detection in chapter 9.

We will now turn our attention to this chapter's other main focus area, transformation of event instances and event streams.

8.2 Transformation in depth

Transformation EPAs take one or more input event instances and create different output event instances that are based on them in some way. This transformation can be *stateless*, meaning that each incoming event instance is processed independently of any preceding event instances and each output event is derived from just one input event, or it can be *stateful* in which case an output event may have been derived from multiple input events in some way.

As we saw in chapter 6, we divide transformation EPAs up into a family of related EPA types. The family members, shown in figure 8.5, differ depending on whether they are stateful or stateless, and depending on the number of input and output terminals that they have.

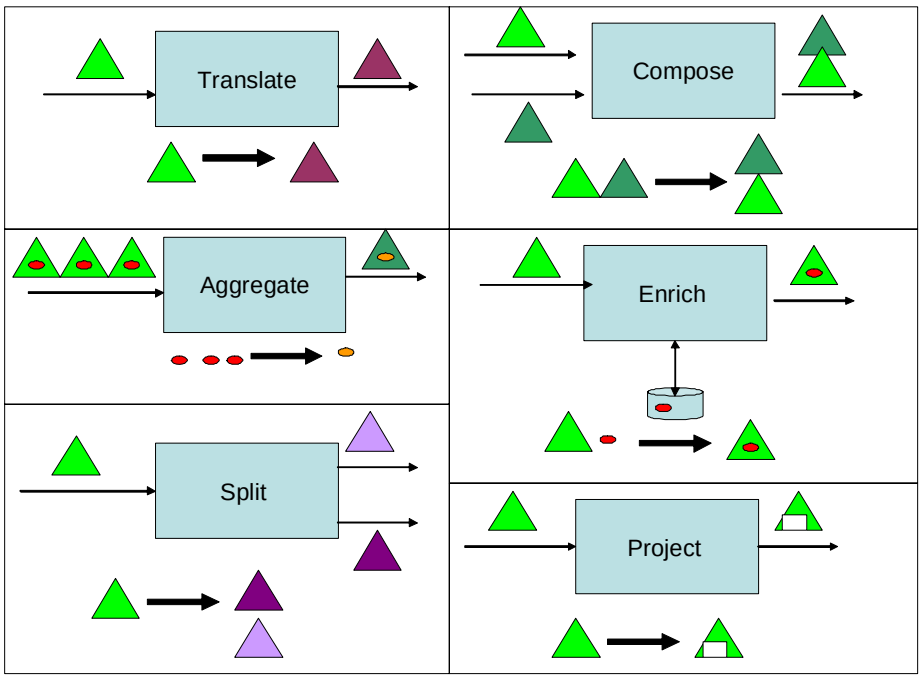


Figure 8.5 Classification of the different kinds of transformation Event Processing Agent

We will start by looking at the three stateless agents that have a single input and single output stream, and then move on to the `Split`, `Aggregate` and `Compose` agents. Our discussion will focus initially on transformation of the event instance's payload attributes and we will defer discussion of the header attributes until the end of this section.

8.2.1. Project, translate and enrich

These EPA types all have the same basic structure, which is shown in figure 8.6.

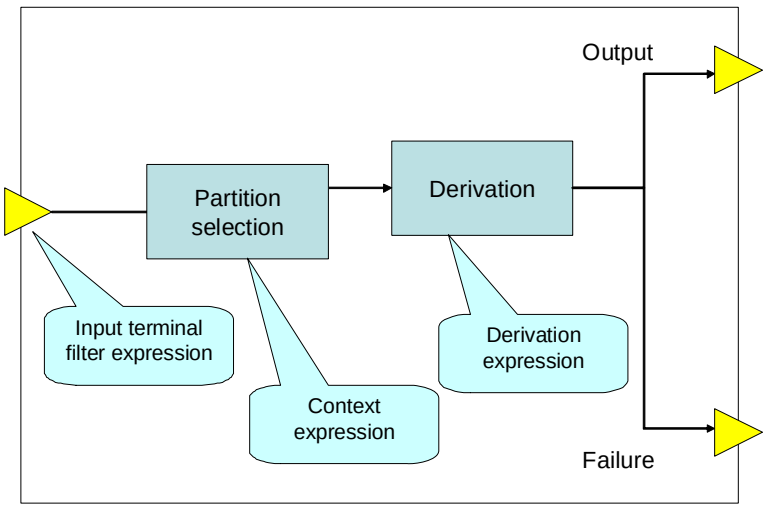


Figure 8.6 A schematic representation of the internal logic of a transformation EPA.

These EPAs have a single input terminal which, just like all other input terminals, can have a filter expression assigned to it. As with other EPAs there is then a context selection step; if the EPA is associated with a context then this step filters and routes the incoming event to the appropriate instance of this EPA in the way we discussed in section 8.1.3.

The heart of the transformation EPA is the derivation step which generates the output event instance. If the derivation step is successful then the output event is routed via the output terminal. In some cases it's possible for the derivation step to fail in some way. If this happens then the agent routes the original incoming event out through the Failure terminal, allowing it to be processed by error handling logic downstream in the Event Processing Network.

The EPA specification includes the mapping or derivation rules that describe how the output event's attributes are to be derived from the input. We refer to these as the *derivation expression*. If the output type just has simple attributes, then things are relatively straightforward: each attribute in the output event, if it is to be included at all, is either copied across from the input event, set to a fixed value, or set to a value computed from the input event and / or other input data. In simple transformation cases, for example ones where the derivation rules copy most of the attributes straight across and just make adjustments to a few of their values, then the output event instance can have the same type as the corresponding input event. However a transformation agent is permitted to make more radical modifications, resulting in an output event instance that is of a different type from the input.

THE PROJECT EPA

The `project` EPA is the simplest kind of transformation agent. In this EPA the payload of the output event instance is made up of a subset of the attributes from the input event; the specification of the `project` EPA can be as simple as the list of input attributes to be copied over. Figure 8.7 shows an example of an event being processed by a `project` EPA.

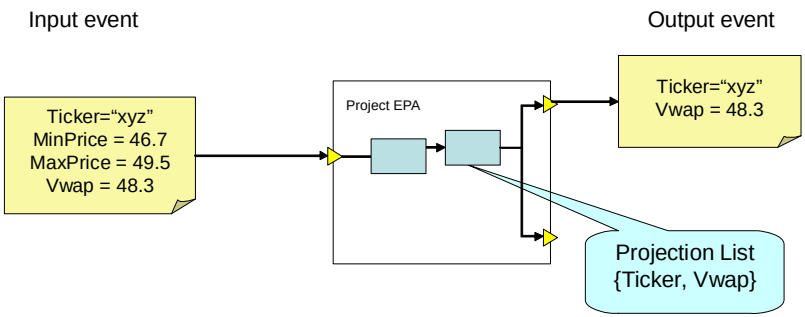


Figure 8.7 A `project` EPA can be used to simplify an event by removing one or more of its attributes

This example comes from a stock trading application, where the input event contains a Stock Ticker attribute, to identify the stock in question, along with Maximum, Minimum and Volume Weighted Average Price (Vwap) attributes. The derivation expression contains the list of attributes to be projected in to the output event – in this case we are only interested in the Ticker and Vwap attributes, so the MinPrice and MaxPrice are projected out of the event.

THE TRANSLATE EPA

The `translate` EPA is an extension of the `project` EPA. As well as being able to copy attributes across to the output event, this EPA can modify the values of copied attributes or even insert new attributes into the output event. The derivation expression for a `translate` EPA specifies how each output attribute is to be computed.

Figure 8.8 shows a simple example.

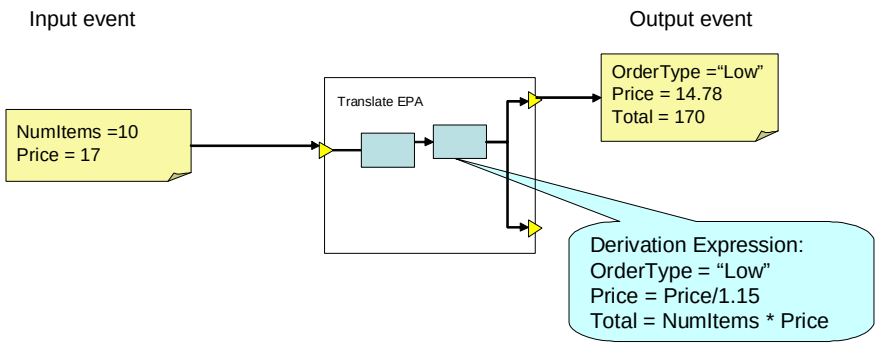


Figure 8.8 A translate EPA that modifies the attributes of the input event

In this example the EPA inserts the new attribute `OrderType` with a fixed value (“Low”), it adjusts the value of the `Price` attribute from the value supplied in the input event and it adds a new attribute called `Total`, computed from two attributes of the input event. The derivation expressions in this example are simple arithmetic assignments where the left hand side of the expression identifies an attribute in the output event, and variables in the right hand side refer to attributes in the input event. That is just a syntax we have used for illustration, in practice event processing languages may implement such translations as part of the language itself, as shown in examples later in this chapter, or in some cases they allow transformations to be written using conventional languages such as:

- Scripting languages such as Javascript, PHP or PERL. These have powerful string handling functions, and are useful if you want to manipulate the values of the input attributes
- General purpose programming languages such as Java
- XSL (Extensible Stylesheet Language) Transformation (XSLT)
- These languages can be used just to express simple assignments, one for each attribute in the output event, but when handling event types that have more complex structures, such as lists, arrays and structured elements, it is sometimes helpful to be able to use loops and other program control structures in the language itself.

The XSLT language was designed for the purpose of converting one XML document into another, or for converting an XML document into a non-XML format. However it can be used to derive the output event from the input event, regardless of whether the events in question are actually serialized as XML documents. We can use it in a manner similar to the way that XPATH is used for filtering (see section 8.1.1). We start by viewing the attributes of the input event instance as an XML document, regardless of whether it is actually serialized as such. The event instance’s attributes correspond to the top-level elements of the header XML document (the immediate children of the root element of that document). The name of each

XML element matches the name of the event attribute, and if the event attribute has a complex data type then the corresponding XML element contains child elements representing the structure of that complex data type. We take a similar approach with the output event, and then map the effect of our XSLT stylesheet onto the underlying attributes in the input and output event instances.

We will illustrate this with the example that we saw in figure 8.8. Our input event, represented as a document, looks like this:

```
<event>
     <NumItems>10</NumItems>
<Price>17</Price>
</event>
```

Note that this is a logical representation, and an implementation does not necessarily create an actual document at runtime. Similarly the output event can be represented like this:

```
<event>
<OrderType>Low</OrderType>
     <Price>14.78</Price>
<Total>170</Total>
</event>
```

Listing 8.1 shows an XSLT stylesheet that can be used to transform the input event to the output event.

Listing 8.1 XSLT stylesheet to perform the transformation

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0" >

  <xsl:template match="/event">
      <event>
          <OrderType> Low </OrderType>
          <Price>
            <xsl:value-of select="round (100*//Price div 1.15) div 100" />
          </Price>
          <Total>
            <xsl:value-of select="//Price * //NumItems" />
          </Total>
      </event>
  </xsl:template>
</xsl:stylesheet>
```

This uses an XSLT template to construct the output event. The <OrderType>, <Price> and <Total> tags insert the corresponding XML tags into the output. The <xsl:value-of> elements fill in the computed values. Note the computation of `Price` includes a rounding operation, as otherwise it would be set to the value 14.782608695652176.

THE ENRICH EPA

The `enrich` EPA can copy, modify or insert new attributes, just like the `translate` EPA, but its derivation step can take input from a global state element, as well as from the input event. It can thus add data to the output event that was not present in the input. As you can see from figure 8.9, the enrich EPA has two input terminals, one for the input event instance and one for the global state information.

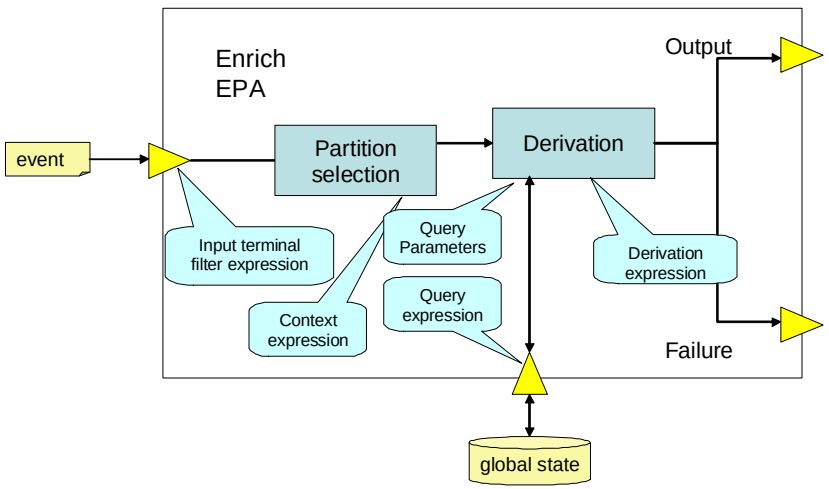
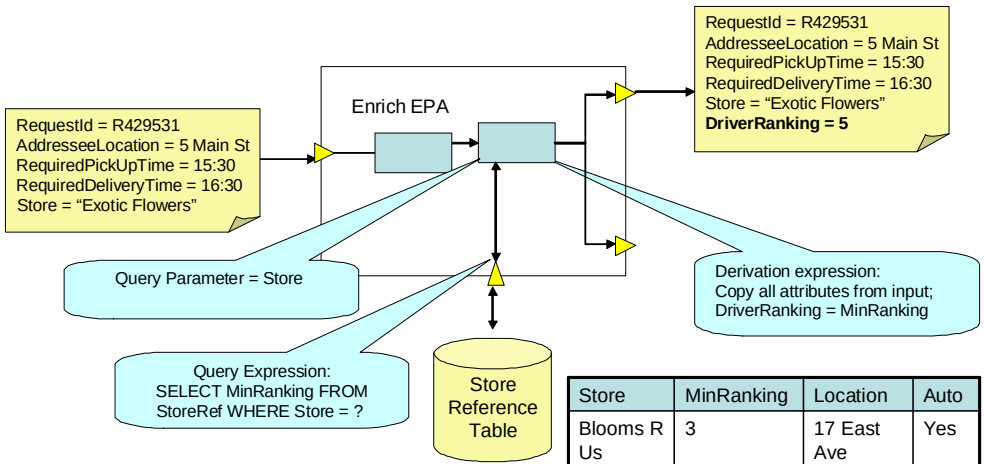


Figure 8.9 Schematic representation of an `enrich` EPA.

You will recall from chapter 6 that the global state element encapsulates data that can be accessed by one or more Event Processing Agents. The `Enrich` EPA uses a global state element as a source of Reference Data (recall from chapter 6 that reference data is relatively slow-changing data, often external to the Event Processing application, affecting the processing performed by the Event Processing application). It uses this reference data when building the output event.

In Figure 8.10 we show an example taken from the Fast Flower Delivery application. This should help explain both the concept and the specification of an `enrich` EPA. You will see that it is a very simple example – a single attribute from the input event is used as a key to return a single value from a database table which is then added to the output event. However you should be able to see how the mechanism described here can also be used to handle more complicated cases.



Store	MinRanking	Location	Auto
Blooms R Us	3	17 East Ave	Yes
Exotic Flowers	5	33 High St	No
…	…	…	

Figure 8.10 This Enrich EPA uses the Store attribute of the input event to look up the minimum driver ranking that is acceptable to this Store, and adds it as an attribute to the output event.

The enrichment operation shown here happens right at the start of the Fast Flower Delivery event flow. A flower store (in this case "Exotic Flowers") submits a `DeliveryRequest` event, shown at the top left of the illustration. When the Exotic Flowers store signed up for the Fast Flower Delivery application a profile record was created for it and this is held centrally, along with profiles from other stores, in the Store Reference table shown at bottom right. Among these details is the minimum ranking that the store is prepared to accept (you will recall that one of the things the Fast Flower Delivery application does is to assign rankings to its drivers).

When it comes to assign a driver to this request, the Fast Flower Delivery application needs to know the minimum ranking that will be acceptable to the store but, as you can see from the diagram, this information is not contained in the original Delivery Request. The application therefore uses the `enrich` EPA shown to look up this value from the Store Reference table and add it as a new output to the `DeliveryRequest` event. You can see the result in the output event at the top right.

Now let's look at the three bits of specification needed to make this work (remember that this EPA has to work for all stores, not just for "Exotic Flower"). Each time it receives an

input event, the `enrich` EPA is going to issue a query to the global store, and the first bit of specification we encounter is the Query Parameter list. This identifies one or more attributes[6] from the input event that are to be used in the query. In our example we just need one, the Store name, since this is used as the key to look up the data we need.

Next we come to the specification of the query itself, which you can see below and also attached to the global state input terminal in figure 8.10. Different global store implementations support different kinds of query, the simplest kind being one that just supports a basic keyed lookup like that provided by a Java hashtable. In our example we are assuming that the Store Reference data is held in a relational database table and that access to this data is via a SQL statement.

```
SELECT MinRanking FROM StoreRef WHERE Store = ?
```

This is a very simple query, which just looks up the reference data for a single store and returns the ranking value from it. The ? is a placeholder for a parameter, and each time an input event is received a parameter from the input event, as specified by Query Parameter list, is substituted in place of the ?. In the example of Figure 8.10 it's the value "Exotic Flowers" that is used. This query returns just one piece of information, the MinRanking value.

Lastly we come to the derivation expressions, which indicate how the output event is to be constructed. These are similar to those in the `translate` EPA except that their computations can now use the values returned by the Query as well as the values of attributes in the input event. In our simple example we just copy across the attributes from the input event and add a single new attribute, `DriverRanking`, containing the result from the query.

Before we leave the `enrich` EPA, you may have noticed that our query returned exactly one row from the table. What should an enrich EPA do if the query returns more than one row, or none at all? The case where no rows are returned is likely to be an error, so in that case the EPA routes the original, unchanged event through to the `Failure` output terminal (if your application actually expects that some events don't get enriched that's fine, there's nothing to stop it from continuing to process events that are emitted via the `Failure` terminal).

There are several possible actions that an `enrich` EPA can take if the query returns multiple rows, but what's appropriate here depends on the nature of the query and the meaning of the data in the table, so the EPA specification contains a policy that indicates which one to take. The three options are:

- First: Use only the first row that is returned
- Last: Use only the last row that is returned
- Every: A separate output event is generated, one for each row that is returned

---

[6] It could identify them simply by attribute name, or (if more complex input event types are being used) by XPATH expressions

- Combine: A single output event is returned, but the derivation rules have access to all the rows when preparing the output.

The combine option is useful if the output event type supports structured data types, as it could be used to build a list of data items in that event. For example suppose the input event is an indication that there's been a power failure in a floor of a building. An `enrich` EPA could augment that event with the list of all the systems impacted by that failure, obtained by querying a facilities database. We will now look at partial converse of this, the `split` EPA which can be used to pull a single event apart into constituent parts.

8.2.2 Split

The `split` EPA takes a single input event and breaks it apart to produce multiple events as output. As you can see from figure 8.11, a `split` EPA can have multiple output terminals. In an application one might want the different output events (which could be of different types) to be processed differently, so it is convenient to have them presented on different output terminals.

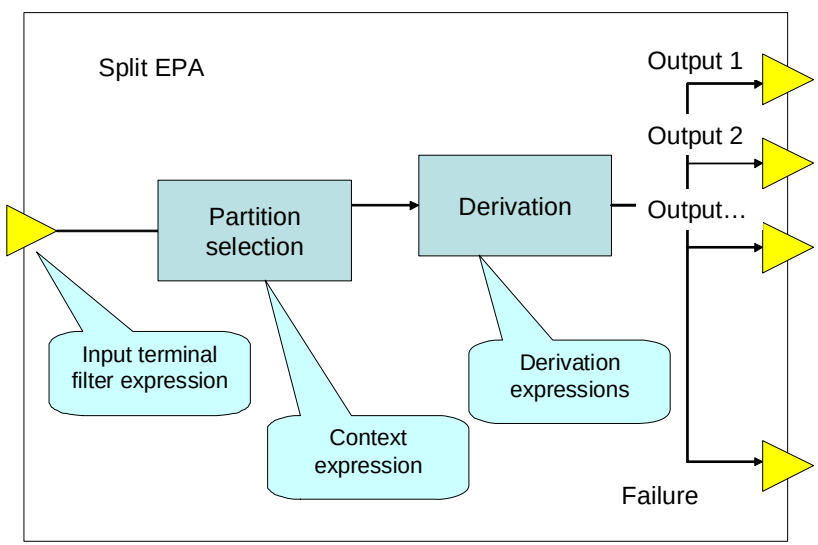


Figure 8.11 Schematic representation of a `split` EPA

A `split` EPA can have filtering on its input terminal and a context selection step, just like every other EPA. The difference between this EPA and the `translate` EPA that we looked at

earlier lies in the derivation step, since in the `split` EPA the derivation expressions are capable of producing multiple output events.

A split EPA can be used in circumstances where an application receives a complicated event object and has multiple consumers each interested in only seeing one particular aspect of the original event; an extreme example would be a split that extracts each individual attribute from the original event and forwards it on as a separate event instance in its own right. It's also useful in cases where the input event instance is actually a batching together of several individual events, for example a composite event or the result of the enrich-with-combine operation that we looked at in the previous section. We might need to split such an event into its constituent parts so that they can be processed individually.

There are three ways of specifying a `split` EPA, depending on the nature of the split to be performed. We will start by describing the *static* approach which is useful when the input event has a fixed structure, or where there's a well understood set of events that we wish to pull out of it. We then describe the *iterative* approach which is useful if the input event contains lists and repeating structures of arbitrary length. Both these approaches can use the same transformation languages that we discussed in section 8.2.1. Finally there is the special case of the composite event type.

STATIC SPLITTING

A static split specification looks just like a translate specification except that:

- It consists of multiple derivation expressions, instead of just one
- Each derivation expression is associated with one of the output terminals.

When an input event is received it is processed independently by all of the derivation expressions, so if you have five derivation expressions the process will result in five output event instances. These event instances are then routed to their associated output terminals. Note that you can have more than one derivation expression assigned to the same terminal, in particular you can have them all assigned to Output 1 terminal. This would mean that the five output events would be emitted, one after the other, through Output 1.

The derivation expressions could well be projection expressions, each extracting a different set of attributes from the input event. Each expression could extract a completely different set of attributes from all the others but this doesn't have to be the case, for example in a stock trading application you might well want all the output events to contain a Tick Symbol attribute.

ITERATIVE SPLITTING

A static splitting specification will always (except in failure cases) produce the same number of output events. In contrast, with an iterative split the number of output events is determined by the content of the input event. Iterative splits are useful if the input event type contains variable length lists or repeating structures of variable length.

An iterative split has a single derivation expression typically written in XSLT, a scripting language (such as Javascript, PERL or PHP) or a general purpose programming language such as Java. These languages contain loops or other program control structures (in the case of XSLT there are matching constructs) that make it possible to walk through the incoming event instance and unpack it into a variable number of output events. However you encounter a slight difficulty when using XSLT, as this language only generates a single output stream, yet we want to use it to generate multiple output events.

We can get round this problem by using a single XSLT stylesheet to output a kind of composite event. This is something that takes the form of a single event, but its attributes are in fact themselves all events – in fact they are the events that make up the output of the iterative split operation. The EPA can then this output from XSLT and unpack it mechanistically, routing the events that it extracts out via the appropriate output terminals. This unpacking process is a special case of our third kind of split, the composite event split.

SPLITTING COMPOSITE EVENTS.

In chapter 3 we came across the idea of a composite event type. This is a special type of event which is actually made up of a collection of other event types. As a result, an instance of a composite event contains a collection of other event objects, bundled up together either for convenience, or because there is some causal or other connection between them. There's enough information contained in the composite event type definition to let someone extract the member events from a composite, and so a split EPA can be configured to decompose a composite event just by being given its type; no XSLT stylesheet or further derivation expression is required.

8.2.3 Aggregate

The kinds of transformation we have looked at so far have all been stateless ones. Each input event is processed independently of any others and gives rise to at most one output event. In contrast our two remaining kids of transformation, `aggregate` and `compose`, are stateful. An output event from one of these kinds of EPA can contain information derived from more than one input event. As they are stateful, they operate within an event processing context, like the stateful filters that we discussed in section 8.1.3.

The `aggregate` EPA is widely used in stream processing. It takes a single stream of input events, groups the event instances from this stream into context partitions and then derives output events from them. You can see from figure 8.12 that the logical structure of an `aggregate` EPA is similar to that of a `translate` EPA, except that there is an additional step between the partition detection and output event derivation.

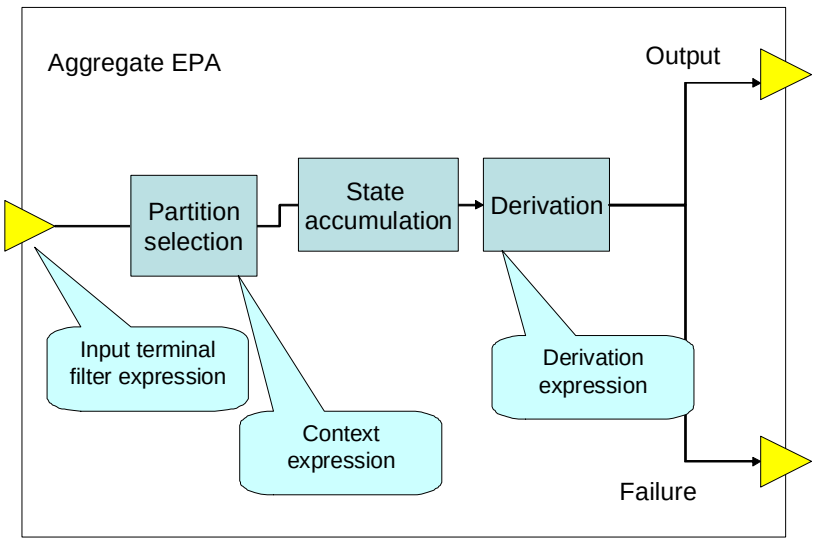


Figure 8.12 A schematic representation of an `aggregate` EPA, showing the state accumulation step

In this additional step the input events that belong to the current context partition (or in some implementations data derived from these events) are gathered together so that they are available when the derivation step runs. Note that the derivation step does not necessarily run every time an input event is received, in fact the default is for it to run only when a temporal context partition closes.

There are two main kinds of `aggregate` EPA. One of these is the reverse of the composite event split that we mentioned in the previous section; it takes the events in the current context partition and combines them to form a single composite event which it then routes through the output terminal.

For the remainder of this section we will concentrate on the other kind of `aggregate` EPA, one which is arguably more interesting. In this kind of aggregate we don't simply bundle together the events that are waiting in the context partition, instead we use a derivation expression to construct an output event. This derivation expression is similar to a derivation expression for the `translate` EPA, except that instead of reading attributes from a single input event, it has a repertoire of *derivation functions* that it can use to compute information from the collection of events in the context partition. To see what that means, let's look at the example in figure 8.13 which comes from the Fast Flower Delivery application.

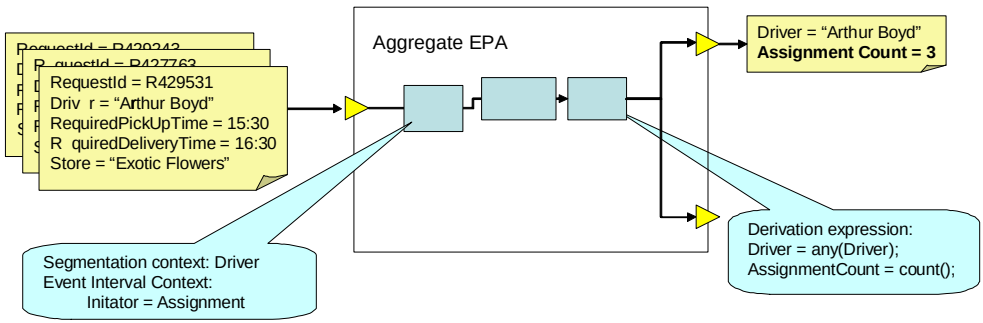


Figure 8.13 This example from the Fast Flowers Delivery application shows how an aggregate EPA can be used to count the number of Assignment events were associated with a given driver on any one day

The Fast Flowers Delivery application uses an `aggregate` EPA to count the number of assignments that each driver receives during the course of each day; this information is then used to help assign a ranking to each driver. Note that phrase "each driver receives during the course of each day": this means that the application needs to maintain a separate total for each driver, and the counting process needs to start afresh every day. That suggests that we need to use a separate instance of the EPA for each driver and that we need to track each day separately. By now you've probably realized that the way to do this is via a context definition, and you can see the one that we are using at the bottom left of figure 8.13. It is a composite context made up of a combination of a segmentation context and a temporal context. The segmentation part assigns each driver to a separate instance of the EPA. The temporal part ensures that each instance outputs its result at the end of the day, and a new instance (for each driver) is created each day. In actual fact that's not quite what happens in this example since we have decided we don't want to receive any output event if a given driver has chosen to have the day off. To do this we use an event-initiated temporal context. That means that the new EPA instance, for any particular driver, is only created when that driver receives his or her first assignment of the day.

As we approach the end of each day, we should have as many instances of the EPA as there have been active drivers. Each instance has been collecting up its input events[7] and at the end of the day they each run their derivation step and output their results. The cleverness of this EPA specification lies mainly in the context definition and the derivation expression, shown at the bottom right of figure 8.13, is relatively simple. We want the output event to contain the name of the driver and the count for that day, and to produce these we use a couple of derivation functions.

[7] Some implementations literally save all the input event instances and aggregate them at the end, others compute partial totals as they go along. Either way the result is as if the events have all been saved.

The driver's name can be taken from any of the event instances in the context partition (because the segmentation context uses the `Driver` attribute all to segment the input events, all the events in a given context partition will have the same `Driver` attribute value). The `any` derivation function instructs the derivation step to pick an arbitrary instance and extract the value of the named attribute from it. The `count()` derivation function, as you might expect, returns the number of instances in the current context partition. In our example you can see that driver Arthur Boyd received three assignment events.

Several of our derivation functions take an attribute name or XPATH expression as an input parameter. Such a function takes all the event instances in the current context partition that contain this attribute[8], extracts the values of the given attribute from each event and then performs a specific operation on this set of attribute values.

We've already discussed a couple of derivation functions, here is a longer list:

Table 8.2 Derivation functions that can be used in an aggregate EPA

Name	Argument	Returns
First	Attribute Name	Attribute value taken from the first event that arrived in the context partition
Last	Attribute Name	Attribute value taken from the last event that arrived in the context partition
Any	Attribute Name	Attribute value taken from an arbitrary event in the context partition
Min	Attribute Name	Smallest of all the values of the attribute
Max	Attribute Name	Largest of all the values of the attribute
Sum	Attribute Name	Sum of all the values of the attribute
Avg	Attribute Name	Arithmetic mean of all the values of the attribute
Distinct	Attribute Name	Number of distinct values of the attribute
Concat	Attribute Name	List of all the attribute values
DConcat	Attribute Name	List of all the distinct attribute values
Count	-	Number of event instances in the current context partition
PartitionCount	-	Number of active partitions for the current context
GlobalCount	-	Number of event instances in all active partitions of the context

[8] Aggregate EPAs are often used in situations where all input events have the same type, in which case the attribute is likely to be present in all the events in the context partition.

In the example in figure 8.13 each instance of the EPA runs its derivation step only once, when the temporal context partition ends (in this case at then end of a day). This is the default behavior for an aggregate EPA, but there are some situations in which you might want to receive intermediate output results before the end of the context partition. To allow this, we have one further configuration parameter for an `aggregate` EPA, and that is the *aggregation frequency*. A frequency value of one means that the derivation step runs every time a new input event is received into the context partition, a frequency value of two means it runs for every second event, and so on. Note that you would normally only use this parameter if your temporal context has non-overlapping windows.

8.2.4 Compose

Our final transformation EPA is the compose EPA. This is stateful, like `aggregate`, but instead of operating on successive events in one stream it takes in two input streams, which we refer to as the *left stream* and the *right stream*, and it matches event instances from one stream against instance form the other, in other word it performs a *join*[9] operation on the two streams. Figure 8.14 shows the logical construction of a `compose` EPA.

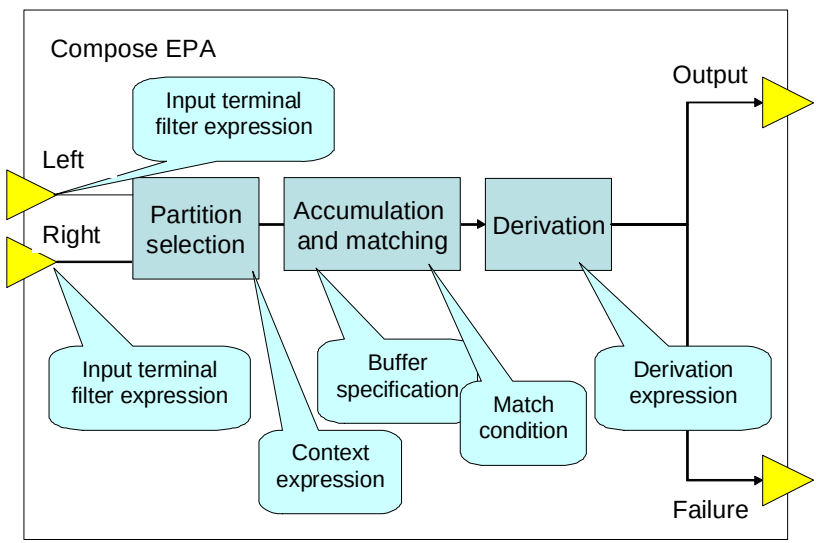


Figure 8.14 A schematic representation of the `compose` EPA.

[9] The terminology *left*, *right* and *join* is borrowed from relational algebra, as the effect of this EPA is similar to a JOIN operation on a pair of database tables

As with all EPAs, there is an optional filter step on the input terminals, followed by a context step that routes incoming events to the appropriate EPA instance (or instances). Each instance accumulates these events and runs a matching operation that compares event instances from one input stream against those from the other. The matching operation runs like this: each time a new event instance is received, from either one of the two input terminals, it is paired with events that have previously been received from the other terminal, and this pair of events is then tested to see if they meet the match condition. If they do meet the condition then the derivation step runs and produces an output event, based on the two event instances that make up the pair.

We will illustrate this using the simple example of figure 8.15. In this example suppose that a highway authority wishes to measure the speed of vehicle traveling over a particular section of highway. It installs a camera at either end, one to produce an `arrival` event whenever a vehicle enters the section, and the other to produce a `departure` event when a vehicle leaves, and then has to match the arrival event for a particular vehicle with the departure event for the same vehicle so that it can see how long the vehicle has spent in the section of road.

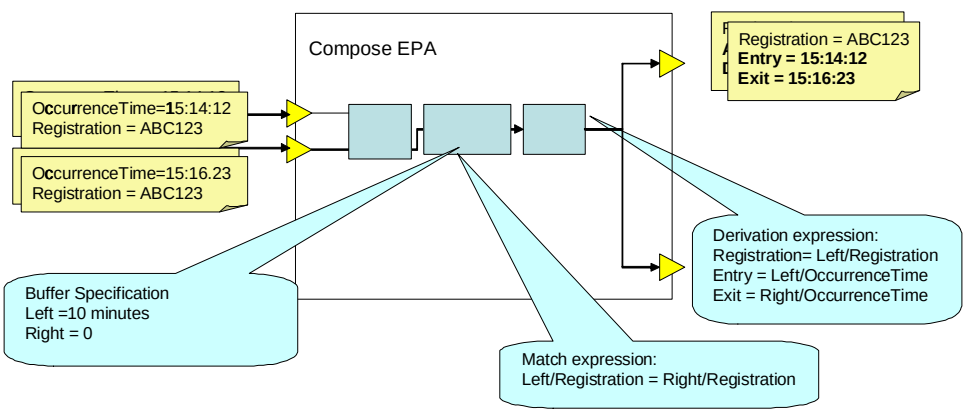


Figure 8.15. Use of a `compose` EPA used to match up vehicle arrival and departure times

You can see some `arrival` and `departure` events at the top left. They both have the same format, containing a vehicle registration mark and an occurrence timestamp (for simplicity we only show the time part of this timestamp). The `arrival` events come into the EPA through the left terminal and the `departure` events through the right one. The match condition in this example is a simple equality test (we are interested in matching up arrival and departure events for the same vehicle). Note that the attributes we are matching on have the same name in both events in the pair, so we have to distinguish them by referring

to them as Left/Registration and Right/Registration. Similarly the derivation expression has to distinguish between Left/OccurrenceTime and Right/OccurrenceTime when constructing an output event.

If the compose EPA were to hold on to every event instance that it received, then it could over time build up a huge number of events. Not only would this be bad for performance, remember that each time an event is received it has to be matched against all the accumulated events from the other stream, but it's often the case that we only want to see matches that happen reasonably close to each other in time. In our example, suppose that this section of highway were to lie on a driver's way to work and she drives along it every day. We wouldn't want arrivals from Monday to be matched with departures from Tuesday, or – even worse – for departures from Monday to be matched with arrivals from Tuesday. To cope with this, the EPA has a buffer specification which controls how many (or how long) events from each input terminal are kept. In this example we have asked for `arrival` events to be kept for 10 minutes (it's only a short section of road that is being monitored and there's no where to stop along it), and we have asked for `departure` events to be discarded as soon as they have been matched.

The matching and accumulation step is controlled by a number of parameters. We have met a few in the example that we have just looked at, but here is a fuller list:

- Left buffer specification; Controls how many event instances from the left buffer should be retained. It can be specified either as a count of instances or as a time interval.
- Right buffer specification; Controls how many event instances from the right buffer should be retained. It can be specified either as a count of instances or as a time interval.
- Unmatched Left Policy; States what should happen when an event is evicted from the left buffer if that event hasn't been matched with anything prior to eviction.
- Unmatched Right Policy; States what should happen when an event is evicted from the right buffer if that event hasn't been matched with anything prior to eviction.
- Match condition. The condition used to judge whether an event from the left stream matches one from the right stream. It can be a simple equality test, such as Left/A = Left/B, or a more complex XPATH expression involving both events, such as count(Left/A) = count(Right/B) + 7.

The default behavior for the unmatched policy is simply to drop the unmatched event and not produce an output event. However the policy can be set to `forward`, in which case it is forwarded to the derivation step as part of a pair along with a null event from the other stream. In our example, if we had set an unmatched left policy of `forward`, then an unmatched arrival event would result in an output event containing the vehicle registration and the arrival time, but no departure time. A `compose` EPA with an unmatched policy of `forward` is sometimes called an *outer join*.

Some buffer size settings are particularly noteworthy. If one of the buffer sizes is set to zero (as in our example) then the operation becomes a *one-way join*. Events from that stream are compared against earlier events from the other stream, but not the other way round. If both buffer sizes are non-zero then the operation is a *two-way join*. If one of the buffer sizes is set to one then only the last event is retained for that stream, this is useful in circumstances where each incoming event on that stream invalidates any earlier event, making a match against such events worthless.

8.2.5 Header attributes and validation

You will recall that in chapter 3 we described a number of header attributes that may (and in some cases must) be present in an event instance. These header attributes, if present, can be read and used as input by context expressions, derivation expressions and the compose EPA's matching condition, but we have been a bit vague as to how their values get set in an output event. That is the question that we will tackle in this section, and which concludes our discussion of event transformation.

The first attribute to consider is the `event identity`. You will recall that this is a system-generated value used to distinguish a particular event instance from any other. It is clear that in a complex transformation, for example an `aggregate`, the output event is different from any of the input events and so must have a different identity, whereas in the case of a `filter` EPA you would expect an event to preserve its original identity. We take the view that events that emerge from the output terminal of a transform EPA are different from the events that go in, even in cases where the transform does not actually result in any change to the input event and so should have distinct system-generated `event identity` values. However in cases where a transform EPA has detected an error and routes an incoming event to its Failure terminal, then it should keep its original `event identity`, along with all its other header attributes. For the rest of this discussion we will concern ourselves only with events that emerge from the output terminal.

You may recall from chapter 2 that EPA output terminals specify the type of event that they emit, so by default this is used as the output `event type`. In some special cases, it's possible that the derivation step might wish to override this at runtime and indicate that the event that it emits is a more specialized subtype of that type, and so it does have power to do this. However, even if it doesn't do this, it is important that the event emitted by the output terminal does conform to the `event type` that is advertised by that terminal. As we have seen, the derivation step has complete power to set the payload attributes of an event. So how do we make sure that this is the case? There are two approaches:

- Static analysis of the derivation expression. In simple cases it's relatively easy to detect cases where the derivation expression won't produce the correct output. For example if the derivation expression consists of a set of output attribute assignments, and there's a mandatory attribute in the event with no corresponding assignment, then the output event instance is not going to be valid. Static analysis techniques can

be built into a tool used to construct the derivation expression in the first place, or it can be applied by a separate verification tool

- Runtime validation. In cases where the output of the derivation step is highly variable (for example where the derivation expression is an XSLT program dealing with highly variable input) it's not always easy to perform static analysis, so the alternative is to have the EPA implementation perform a validation step at runtime. This checks the output event instance against the output terminal's type definition and only allows the event to pass through the terminal if it is valid (if the validation fails then the EPA emits a failure event through the failure terminal). Runtime validation can be costly in performance terms, so implementations don't always offer this, and if they do they allow it to be turned off[10].

Since the output event is a new event instance, the default is for its `Occurrence Time` and `Detection Time` timestamps both to be set to the time at which the transformation EPA creates it. However in the simpler transformation cases, for example `project` or `enrich`, where there's a clear relationship between input and output events it often makes sense for the output event to keep the `Occurrence Time` value from the input event, so the derivation step needs to have the power to do this.

One can go through a similar reasoning process with regard to the remaining attributes. This is summarized in table 8.3 which shows the values set by default (if the derivation expressions say nothing) and indicate which of these values can be overridden by explicit derivation expressions.

Table 8.3 The interaction of a transform EPA with event header attributes

Attribute Name	Default output value	Settable by derivation step?
Event Identity	New system-generated identity	No
Event Type	Specified by output terminal definition	Yes
Occurrence Time	Current time	Yes
Detection Time	Current time	No[11]
Event Annotation	Not present	Yes
Event Certainty	Not present	Yes
Event Source	Identity of the EPA	Yes

[10] You would not want to use runtime validation if you have performed satisfactory static analysis. Also it's common practice to have it turned on during testing, but turned off when putting the application into production

[11] See chapter 11 where we discuss some problems associated with Detection Time of derived events

Having concluded our in-depth look at both stateless and stateful transformation, it is time to turn once more to the Fast Flower Delivery application and give more details of the transform EPAs that were mentioned in the chapter 6.

8.3 Examples in the Fast Flowers Delivery application

Chapter 6 mentioned that the Fast Flower Delivery application involves several filter or transformation EPAs. We have actually looked at two of them already:

- The Bid Request creator, which is an enrich EPA, was illustrated in figure 8.10
- The Daily Assignments calculator, an aggregate EPA, was illustrated in figure 8.13

The remaining examples are the Location Service EPA, the Automatic or Manual Matching EPAs and the Daily Statistics Creator EPA.

LOCATION SERVICE EPA

This is a `translate` EPA whose job is to compare the raw latitude/longitude readings that come from a driver's GPS device against a set of geofences to determine what part of the city the driver is currently in. We show a version of this in figure 8.16 where we have assumed, appropriately enough, that our flower delivery application is set in the city of Florence, and for simplicity we have divided the city up into four regions, meeting at the point with latitude 44.778° North, longitude 11.259° East

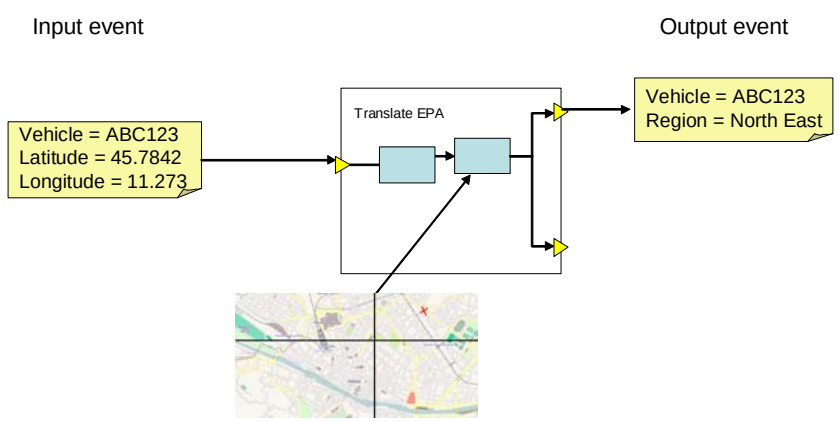


Figure 8.16 Locating a Fast Flower delivery driver in the city of Florence.

This process of converting a latitude/longitude reading to a geographical point or region is technically known as *reverse geocoding* and there are number of commercial products and

online services that offer it. We will show here how you can do it, in our very simple case, using Javascript.

Listing 8.2: Javascript implementation of a simple reverse geocode translation

```
output.Region = reverse_geocode(input.Latitude, input.Longitude);        #1

function reverse_geocode(latitude, longitude) {                          #2
     if (latitude > 43.7783)
             if (longitude > 11.259)
                     return "North East";
             else
                     return "North West";
     else if (longitude > 11.259)
             return "South East";
     else
             return "South West";
}
```

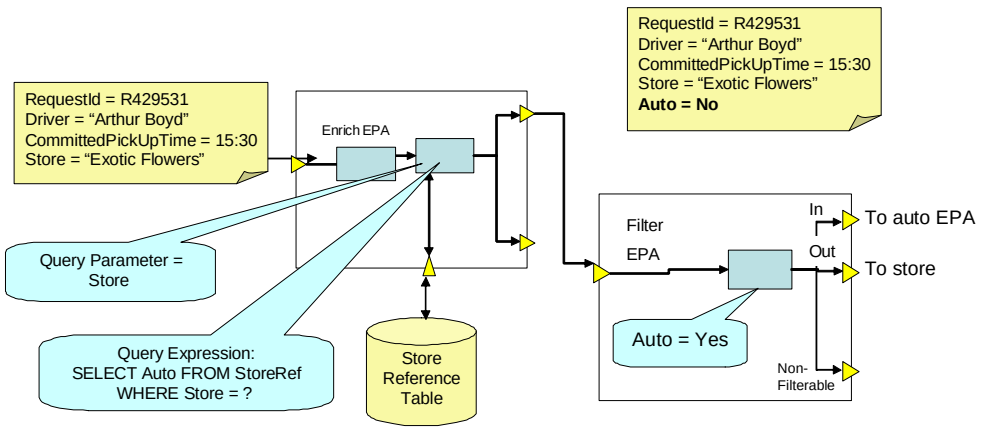
Cueballs in code and text

#1 Derivation expression
#2 Conversion function

This tests the latitude and longitude values from the input event against the borders of the four regions (conveniently these are lines of constant longitude and latitude so the tests are simple). For clarity we have split the conversion code out from the derivation expression #1 into a separate function #2.

AUTOMATIC OR MANUAL MATCHING EPAS

When a driver submits a `delivery bid` event in the Fast Flowers Delivery application, the system has to decide whether to forward the event back to the requesting flower store, or whether to send it on to the automatic assignment process. This decision is actually performed by two EPAs, as shown in figure 8.17.



Store	MinRanking	Location	Auto
Blooms R Us	3	17 East Ave	Yes
Exotic Flowers	5	33 High St	No
…	…	…	

Figure 8.17 The Delivery Bid request is first enriched with the store's automatic/manual assignment preference and then routed using a filter EPA.

Delivery Bid requests are first sent to an `enrich` EPA which uses the same Store Reference table that we met in section 8.2.1. This adds an additional attribute to the event, to show whether the store wishes to do its own assignment or not. The output from this EPA is then passed to a `filter` EPA which then routes the event based on this attribute.

8.4 Filtering and transformation in practice

In this chapter we have discussed the use of XPATH and XSLT, but have mentioned that fact that some products use different languages. In this section we provide some examples of filtering and transformation from current event processing languages. As a reminder, you can find out more details of some of these languages, and see examples of them being used to implement the Fast Flower Delivery application by visiting this book's website.

We will start with Figure 8.18 which shows an example of filtering from Streambase.

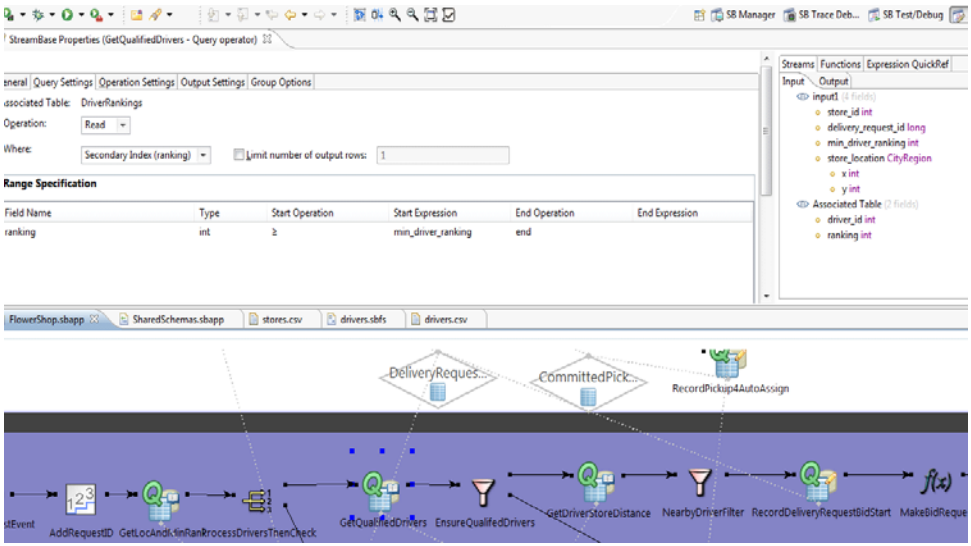


Figure 8.18 An example of a filter from the Streambase product

This example shows the filter from the Fast Flowery Deliver application that is used to select those drivers who satisfy the requesting store's ranking constraint. Our second filter example is a rule based programming example taken from Rulecore, shown in Listing 8.3

Listing 8.3 A Rulecore example of filtering

```
<Rule name="CreateAutomaticAssignments" limit="?" evalMode="once"
level="2">
<Description>This is rule CreateAutomaticAssignments</Description>
<Initialize>
<Assert>
<Event>
<base:XPath>sim:EventDef[@eventType="BidRequest"]</base:XPath>
</Event>
<Expression>
<Property name=”Store”>
<InList name=”AutomaticAssignmentStore”/>
</Property>
</Expression>
</Assert>
</Initialize>
<Views>
<ViewRef name="CreateAutomaticAssignments">
<base:XPath>sim:ViewDef[@name=”CreateAutomaticAssignments”]</base:XPath>
</ViewRef>
</Views>
<Situations>
<SituationRef name="CreateAutomaticAssignments">
<base:XPath>sim:SituationDef[@name="CreateAutomaticAssignments"]</base:XPat
```

```
h>
</SituationRef>
</Situations>
<Actions>
<SituationDetected situationName="CreateAutomaticAssignments">
<ActionRef name="CreateAutomaticAssignments" eventVisibility="external">
<base:XPath>sim:ActionDef[@name="CreateAutomaticAssignments"]</base:XPath
>
</ActionRef>
</SituationDetected>
</Actions>
</Rule>
```

The rule shown in listing 8.3 implements the "Automatic or Manual matching" EPAs that we mentioned in the previous section.

Next we show some transformation examples. Figure 8.19 shows an example from an Apama implementation of the Fast Flower Delivery application. This example shows some code that enriches a DeliveryRequest event by adding a location attribute to it.

```
70    /**
71     * Update the driver's current location, in region.
72     */
73    action updateCurrentLocation(string loc) {
74        edrListener.quit();
75        currentDriverRegion:=loc;
76        // because the current location is changed so what enriched delivery
77        // requests to care for is also changed -- therefore re-establish edr
78        // listener based on the changed current location
79        EnrichedDeliveryRequest edr;
80        edrListener:=on all EnrichedDeliveryRequest(region=loc, ranking<=currentDriverRanking):edr {
81            route BidRequest(edr.dr.requestId, edr.dr.store, driver,
82                  edr.dr.requiredPickupTime, edr.dr.requiredDeliveryTime);
83        }
84
85        updateGUI(driver, loc);
86    }
```

Figure 8.19 A transformation example in Apama.

Listing 8.4 shows the `Bid Request creator` EPA implemented using Esper. This EPA enriches the `Delivery Request` event with the names of drivers who satisfy the ranking and location constraints (see figure 8.10).

Listing 8.4 Bid Request creator example in Esper

```
insert into BidRequest(requestId, store, location, pickupTime,
deliveryTime,
storeManual)
select d.requestId, d.store, d.location, d.pickupTime, d.deliveryTime,
s.manual
```

```
from
DeliveryRequest d unidirectional,
GPSLocationW g
//,sql:DomainDB['select ranking from Driver where driver = ${g.driver} and
ranking
> ${d.minimumRanking}']
,method:Domain.driverRankLookup(g.driver) r
,method:Domain.isStoreManualLookup(d.store) s
where Geo.distanceKM(g.location, d.location) < 10
and r.ranking >= d.minimumRanking;
```

As you can see, there is variety of language styles used to express similar functionality. We will examine these various programming styles further in Chapter 10.

8.5 Summary

Filtering and transformation are two of the three main "intermediary" functional capabilities provided in an event processing network. In this chapter we have looked in depth at the various ways in which filtering can be specified (on an input terminal, in a context expression or in a dedicated `filter` EPA), and we have discussed how XPATH can be used to give a standardized way of describing common filter operations.

We have also looked at the various different kinds of stateful and stateless transformation, and examined the logical internal structure of these different kinds of EPA. We have observed how XSLT can be used in many of these transformation EPAS to derive the output event. We have also discussed the treatment of event header attributes and touched on the issue of event validation.

Our final stateful EPA was the `compose` EPA, used to join a pair of input event streams by looking for matches or correlations between events in these streams. We will develop this idea of matching events further in the next chapter when we complete our detailed review of the EPN model by turning to our final group of Event Processing Agents, the pattern detection EPAs.

Additional reading

G Hohpe, B Woolf: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions (Addison Wesley 2004).

This book is primarily about the use of Message Oriented Middleware to deliver Enterprise Application Integration solutions, but in many cases there is a direct read-across between the messaging patterns it describes and the way that events are handled in an event processing network. In particular the stateless filter and transformation patterns described in this chapter have counterparts described in this book.

Bob DuCharme. XSLT Quickly, (Manning Publications 2001), http://www.manning.com/ducharme/

This book is a tutorial for XSLT and XPATH. In this chapter we showed how these languages can be used for filtering and transformation.

Exercises

8.1 Would the "Non-filterable" output terminal ever be used in an EPA that has an XPATH filter expression? Explain the reasons for your answer.

8.2 The filtering example shown in figure 8.3 uses two filter EPAs connected "in series", that is to say one after the other, to separate Gold, Silver and Bronze customer events. Show how the same effect can be achieved using EPAs connected in parallel to one other. What are the advantages and disadvantages of these two approaches?

8.3 Can you think of example of a `Filter` EPA where it would be useful to have a filter expression on the input terminal in addition to the EPA's principal filter?

8.4 Some event processing applications can encounter situations in which there are multiple event instances corresponding to a single event occurrence, or where there are multiple event instances that aren't sufficiently different from each other to merit separate consideration. Explain how the concepts described in this chapter can be used to remove such duplicate event instances from a single stream of incoming events. How would you have to change this to handle multiple incoming streams (for example streams from different event producers)?

8.5 The compose example we showed in figure 8.15 could have used a segmentation context to partition incoming events by vehicle registration. What would the match condition look like in this case? Are there any implementation considerations that would favor one approach over the other?

8.6 A `compose` EPA implementation can be made more efficient if the match condition only contains simple equality tests. Explain why this is the case.

8.7 Can you suggest more derivation functions to be added to the list in table 8.2?

8.8 What are the advantages and disadvantages of having a derived event which is the result of filtering, translation, enrichment and projection keep the identity of the original event identity as opposed to having it gain a new event identity?

9

Detecting Event Patterns

"Art is the imposing of a pattern on experience, and our aesthetic enjoyment is recognition of the pattern."

- Alfred North Whitehead

It's no exaggeration to say that event pattern detection is the jewel in the crown of event processing. Pattern detection lets us go beyond individual events to look for specific collections of events and the relationship between them. A pattern that is detected has a meaning that goes beyond the occurrence of any single event. Consider a personalized healthcare system where a patient is hooked up to multiple monitors; the individual events reported by the monitors might not in themselves be significant, but a combination of measurements occurring in a certain order might indicate a problem that cannot be detected by looking at any single monitor separately.

In this chapter we shall discuss:

- Pattern definition and its role in an EPA
- Pattern categories
- Specific patterns of various types
- Pattern oriented policies
- Patterns in the Fast Flower Delivery use case

When we discuss patterns in this chapter, we will do this in a language-independent way so you can use your language of choice through our website. Before you get to that point, we should start by introducing the concept of event patterns.

9.1 Introduction to event patterns

Imagine that you have just taken a flight and you are in the baggage reclaim hall, watching the various pieces of luggage travel past you on the carousel. In your mind's eye you have a picture of your suitcase, with the ribbon you tied to it to make it look somehow different from the others, and you are trying to match this mental picture with each piece of luggage as it flows past you. You are, without thinking about it, performing a pattern matching process. Pattern detection in event processing is similar. Instead of luggage on a carousel you have a stream of incoming event instances, and instead of a passenger you have a piece of event processing logic, which we model as a `pattern detection` EPA. This EPA is equipped with a pattern (the equivalent of the passenger's mental picture) and it examines the incoming event stream looking for an event, or sometimes a combination of events, that matches this pattern.

9.1.1 The pattern matching process

Recall that in Chapter 6 we showed that an EPA can be considered as having three steps:

- The filtering step, in which the relevant events are selected
- The matching step that selects subsets of these events
- The derivation step that takes the output from the matching step and uses it to derive new events.

Many EPA types leave out the matching step, and the EPA just does filtering and/or derivation. However a pattern detection EPA uses the matching step to perform its pattern detection. This detection process takes a set of event instances from the filtering step (we call these the *relevant events)* examines them to see if they match the specific conditions of the pattern that it is looking for, and if they do then it outputs a subset of the events (the *pattern matching set*) to the derivation step.

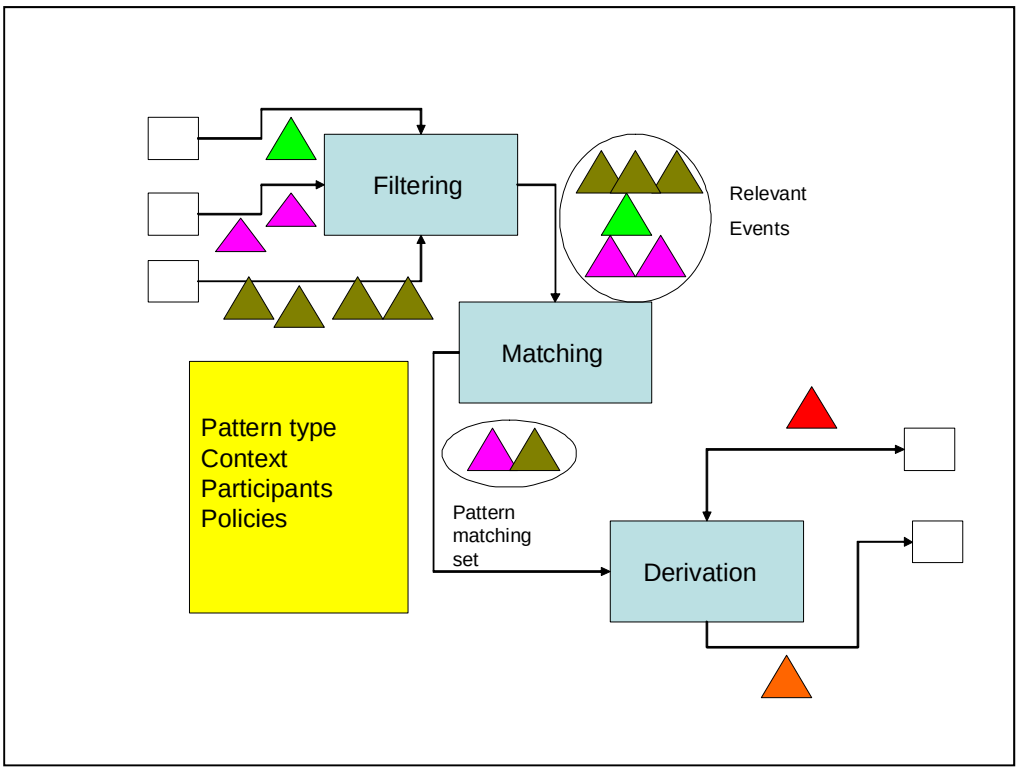


Figure 9.1 The logical structure of a pattern detection EPA, showing the three logical parts of the EPA, along with the "pattern signature" that controls the pattern detection process.

In figure 9.1 we show the logical structure of a pattern detection EPA. The event processing *pattern signature*, shown in the rectangular box, specifies the particular pattern detection process that is to be performed by the matching step. In this example the pattern detection process takes a set of four relevant events and creates a matching set containing two events of two different types, which then serves as an input to the derivation step.

9.1.2 Pattern definitions

We are going to devote the bulk of this chapter to a description of some common types of event pattern but before we can get to that we need to define some of the terms that we will be using. We start with a definition of the event pattern itself.

Definition

A *Pattern* is a function that takes a collection of input event instances and produces a matching set that consists of zero or more of those input events

- The pattern function is defined by the pattern signature which consists of pattern type, participant set, pattern context, pattern assertions and pattern policies. We will now give formal definitions for each of these components of the signature.

Definition

The *Pattern type* is a label that determines the meaning and intention of the pattern and specifies the particular kind of matching function to be used

Some examples of *pattern type* are: `sequence`, `absence` and `moving north`. We will discuss pattern types further in section 9.1.3, and give detailed definitions of some important pattern types in sections 9.2 and 9.3.

Definition

The *participant set* is a predefined set of event types that form part of the pattern matching function. The order of these event types has importance for some pattern functions.

Each pattern function refers to this collection of event types; you will see how the participant set is used when we describe the way the different pattern types work in sections 9.2 and 9.3.

Definition

A *pattern context* is a context associated with the pattern

Each pattern matching operation is executed within a context, as described in Chapter 7; a context may be either a basic context or a composite context. Note that if a context is not specified then there is a default universal context that is independent of time, location and state and has a single context partition.

Definition

A Pattern assertion is an assertion that is used as part of the matching process.

The idea of an assertion was defined in chapter 8 when we discussed filtering. Every pattern signature can contain a *relevance* assertion, which determines which event instances are to be included in the pattern evaluation. Some pattern types may include assertions that play

other roles in the evaluation of the pattern, for example some pattern types require *threshold* assertions. In the discussion of pattern types that follows we will usually assume the relevance assertion as a given and not mention it explicitly. We will however mention any other kinds of assertion that can appear in the signature of a given pattern type.

Unlike the filtering assertion, which is a condition on a single event instance, a pattern assertion may apply to multiple events, possibly of different types. This is best explained with a simple example. Suppose the participant set consists of the event types {E1, E2, E3}, and the relevance assertion is:

E1.A > E2. B and E2.B > E3.C ;

In this case only a combination of events of types E1, E2, E3 that satisfy this assertion are to be considered as relevant to the matching process.

An example of a case where there are multiple assertions within a single pattern signature is an auction process. Consider a process in which a bid is qualified to participate only if the bidder has met a minimum reserve price as defined by the auction call for this specific item. We can use the `count` event processing pattern to check whether there are sufficient bids; this pattern is used to raise an alert if the number of bids is less than or equal to three.

In this case there are two assertions:

1. Bid. Amount ≥ Auction-Call. Minimal-Bid) -- a relevance assertion
2. Count ≤ 3 -- A threshold assertion for the Count Pattern type.

Note that we will define the count pattern, along with other threshold patterns, in section 9.3.1.

Definition

A pattern policy is a named parameter that disambiguates the semantics of the pattern and the pattern matching process.

We have already introduced the notion of policies in chapter 7 in relation to temporal contexts. We will discuss pattern policies in section 9.4.

Next we move to explain a couple of general terms that are used in all pattern type definitions: they are *relevant events* and *pattern matching set*.

Definition

The *relevant events* for a specific pattern are those event instances that occur within the pattern's context, which are instances of the event types listed in the participant set list and which satisfy the relevance assertion.

Now we have defined relevant events we are ready to define the pattern matching set

Definition

A *Pattern matching set* is the output of the pattern matching process; it is a subset of the relevant events.

A pattern matching process takes events as an input and creates a pattern matching set, consisting of the event instances that satisfy the pattern. Let's look at some examples:

- Recall the automatic assignment function from the Fast Flower Delivery example that we just looked at. In this case the relevant events are all the `Delivery Bid` events that have an appropriate pick-up time, while the pattern matching set consists of the single `Delivery Bid` event that has been selected.
- An event processing application designed to detect speculative traders. This application looks for traders that have bought and later sold more than $1M of the same security in the same day. In this case, the relevant events are all the `security buy` or `security sell` events with value of at least $1M. The pattern matching set consists of a pair of events {`security buy`, `security sell`} which satisfy the condition (same customer, same day, and same security). Note that this pattern may yield multiple pattern matching set instances in a single day.

We will conclude this section with an example of a pattern signature, taken from the Fast Flower Delivery application. You will recall that in that application a store can request that the system automatically assigns a driver to a delivery. In such cases the system receives `Delivery Bid` events from drivers that relate to a particular `Bid Request` and uses a pattern to select a driver that to be assigned to the delivery. The signature of this pattern is:

- `Pattern Type: any`
- `Participant set: {Delivery Bid}`
- `Context: segmentation by Request Id, Temporal Interval = (Initiator = Delivery Bid, offset = 2 minutes)`
- `Relevance assertion: Delivery Bid. Committed Pickup Time < Bid Request. Required Pickup Time + 5 minutes`
- `Policies: Repeated type = First, Cardinality = Single`

The relevance assertion checks the delivery bid against the bid request, to make sure that the driver is going to be able commit to an appropriate pickup time. Note that the `Bid Request` event is involved in this assertion, but the `Bid Request` event type is not part of the participant set as the Bid Request event instance is not part of the pattern output.

Next we describe the different categories of event pattern.

9.1.3 Event pattern categories and types

The semantics of the pattern matching operation are determined by the pattern type; in this section we list a number of different pattern types. The patterns defined in this book are patterns that we have found in use surveying a relatively large sample of applications; however we don't claim that it is complete list, and we expect new pattern types to emerge over time. Patterns also vary in frequency of use, and the level of support for these patterns varies by event processing product.

We classify these pattern types into several categories:

- Basic event patterns: these are simple patterns that relate to basic operations on event types or on collections of event types and are described in Section 9.2. They are divided into: logical operator patterns, threshold patterns, relative patterns and modal patterns. An example of a basic pattern is the `all` pattern that designates a conjunction of events.
- Dimensional patterns: these are patterns that relate to time, space, or a combination of time and space. This category is described in Section 9.3 and is divided to temporal patterns, spatial patterns, and spatiotemporal patterns. Examples are: `sequence` (temporal), `min distance` (spatial), `constantly moving north` (spatiotemporal)

Some event processing languages provide built-in support for patterns as primitive constructs in the language, while in other languages applications implement patterns using a composition of language constructs. Either way, the pattern abstraction plays a major part in the design of event based applications. Now, let's meet the patterns themselves.

9.2. Basic patterns

Basic patterns are the most common kind of pattern found in event processing applications. They consist of logical operator patterns, threshold patterns, relative patterns and modal patterns. We start with the simplest: the logical operator patterns.

9.2.1 Logical operator patterns

These are the most basic patterns, based on the three common logical operators: conjunction, disjunction and negation. The patterns discussed here are: `all` (for conjunction), `any` (for disjunction), and `absence` (for negation)

THE ALL PATTERN

This pattern stands for a conjunction of occurrences of events of all the event types in the participant set.

> The `all` pattern is satisfied when the relevant event set contains at least one instance of each event type in the participant set

As an example, let the `all` pattern be defined with {flight reserved, car reserved, hotel reserved} as the participant set. The pattern is matched if an instance of each of these event types, matching the relevance assertions, occurs within the defined context. The order of event occurrences is immaterial.

We show an illustration of this example in figure 9.2. Let's assume that we asked a travel agent to make these three reservations and send us an Email when they are all done. There could be different people in the travel agency dealing in parallel in these three type of reservation, and each of them send an event when their particular reservation has been made. As you can see in the illustration, the last reservation occurs at 11:02, at which point the pattern is matched and the confirmation Email can be sent.

In addition this example demonstrates the need for assertions and policies. The application might require that order should be made only if the car rental company and the hotel are partners of the airline so as to contribute frequent flyer points. This could be achieved by augmenting the all pattern with an assertion that states that the pattern is not matched unless the events reservations are with partner organizations. A policy may be needed to define what should happen in cases where there are multiple hotel reservations; we will return to this question in section 9.4.

If the pattern is matched, the pattern matching set contains the three events, and the derivation phase may create a single event that contains attributes from all three.

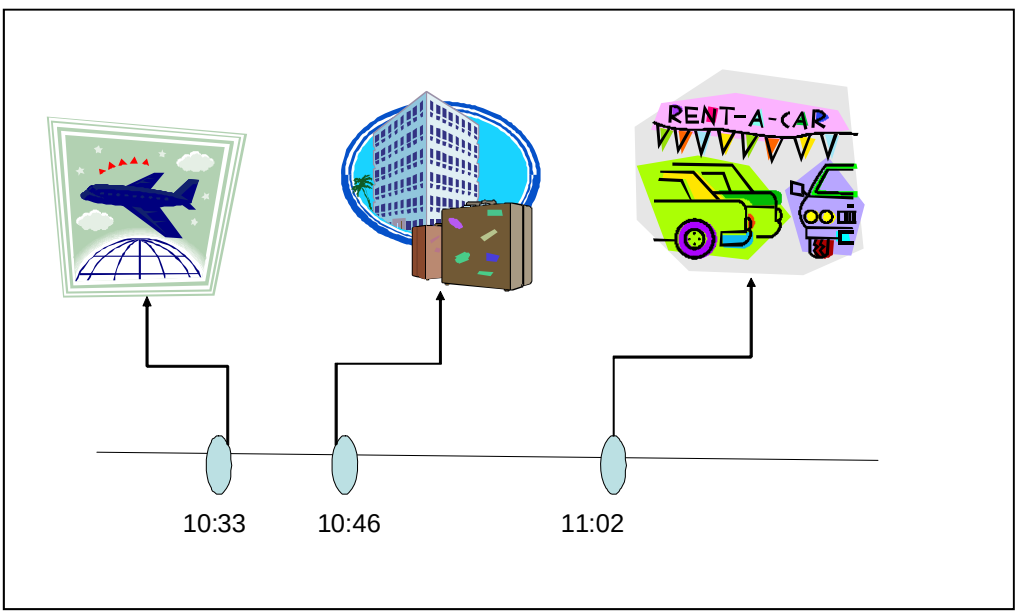


Figure 9.2 Illustration of "all" pattern – an instance of every event type listed in the participant set must be present for the pattern to be matched. In this case the participant set has three types of event (flight reservation, car reservation and hotel reservation) and there's a reservation event of each type.

Next we discuss the `any` pattern. In contrast to the `all` pattern, the `any` pattern just requires the occurrence of one of the event types from the participant set.

THE ANY PATTERN

This pattern stands for a disjunction of occurrences of event types from the participant set.

> The `any` pattern is satisfied if the relevant event set contains an instance of any of the event types in the participant set.

As an example, suppose you have purchased a new house, but have not sold the old one. To complete the payment on the new house you would have as a participant set {lottery win, loan advanced, old house sold}. Any of these events would match the pattern, and the matching set in this case is a singleton, consisting of the event in question.

THE ABSENCE PATTERN

This pattern is sometimes referred to as the "non-event event pattern"; it stands for the absence of any events with certain specified characteristics.

> The absence pattern is satisfied when there are no relevant events

This pattern can be used with no assertion just to test for the absence of any relevant events, and so it is sometimes called the "non-event" pattern. For example the absence pattern is satisfied if we have participant set {E1, E2, E3} but there is no event in the context that is an instance of any of these three event types. If this pattern is associated with a fixed interval temporal context then it can be used to detect time-outs, and in such cases it is sometimes called the "time-out" pattern.

We use this pattern as a way of specifying the various time-out alerts in the Fast Flower Delivery application. For example there is a Pick-up alert, which detects when a driver misses the pick-up deadline. This is modeled using the `absence` pattern; the pattern participant set consists of a single event {`Pick-up Confirmation`} and the temporal context starts with the `Assignment` event and terminates after a time offset, calculated as the Required Pick Up time + 5 minutes.

If no `Pick-up Confirmation` event occurs within the temporal context, then the absence pattern is detected. Note that the matching set of an absence pattern is empty.

Turning to the example shown in Figure 9.2, suppose we have the signature:

```
Pattern: absence
Participant set:  {Car Reservation}
```

Context: temporal event interval context with the initiator = Flight Reservation and terminator = Hotel Reservation.

This pattern is satisfied in this example, since the context interval lasts from 10:33 to 10:46 and there is no car reservation in that time period.

Next we move to discuss threshold patterns.

9.2.2 Threshold patterns

Threshold patterns contain some kind of threshold condition. They select events that cause this threshold condition to be satisfied.

THE COUNT PATTERN

The count pattern counts the number of relevant event instances and tests this value against a threshold assertion. The assertion typically consists of one of the following relations: >, <, =, ≥, ≤, ≠.

> The count pattern is satisfied when the number of instances in the relevant event set satisfies the count threshold assertion.

The matching set in this case includes all the events that were counted, thus it may include multiple events of the same type. Note that in some cases this pattern can be detected while the collection is incomplete, for example if the threshold assertion is ">5" and we get a sixth relevant event then the threshold assertion is satisfied, regardless of what happens later.

An example of the use of such a pattern is a customer satisfaction application that detects when customer sends at least (≥) three complaints to a call center within a single day, as this indicates that the customer is likely to be unhappy with the service being offered. Figure 9.3 illustrates this example.

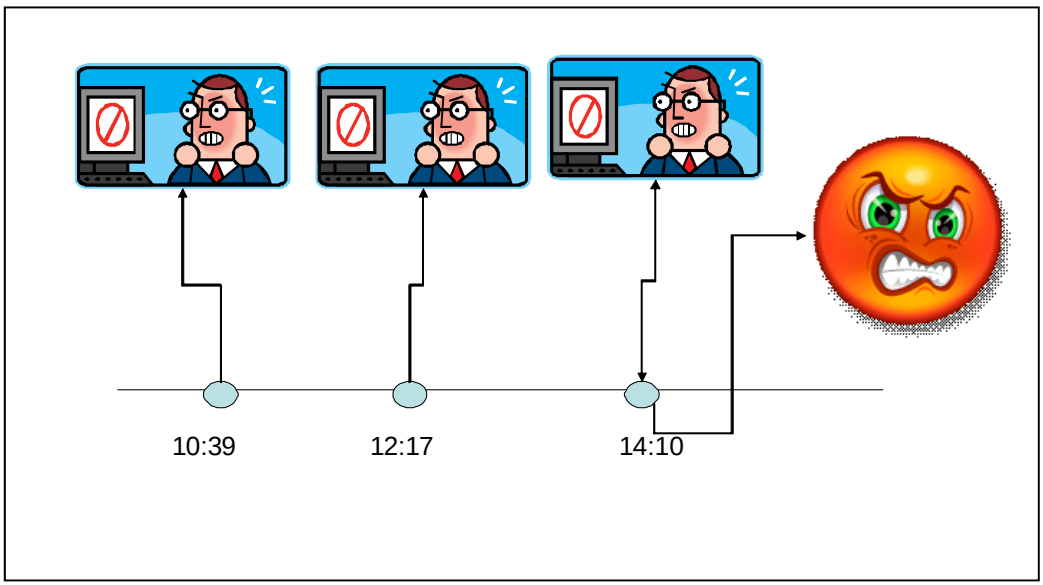


Figure 9.3 An illustration of a count pattern where three instances of the same event type – customer complaint - match the pattern and suggest the likelihood of a frustrated customer.

In this case the context is a single working day. If the pattern match is evaluated each time a new event is received, then it is possible to detect the pattern at 14:10. If the EPA does not run the match operation until the context window has closed, then detection does not occur until this has happened – in this case that is at the end of the working day.

THE VALUE MAX PATTERN

In the `value max` pattern the matching operation examines a given attribute of each event instance and tests its maximum value against the threshold assertion.

> The value max pattern is satisfied when the maximal value of a specific attribute over all the relevant events satisfies the value max threshold assertion.

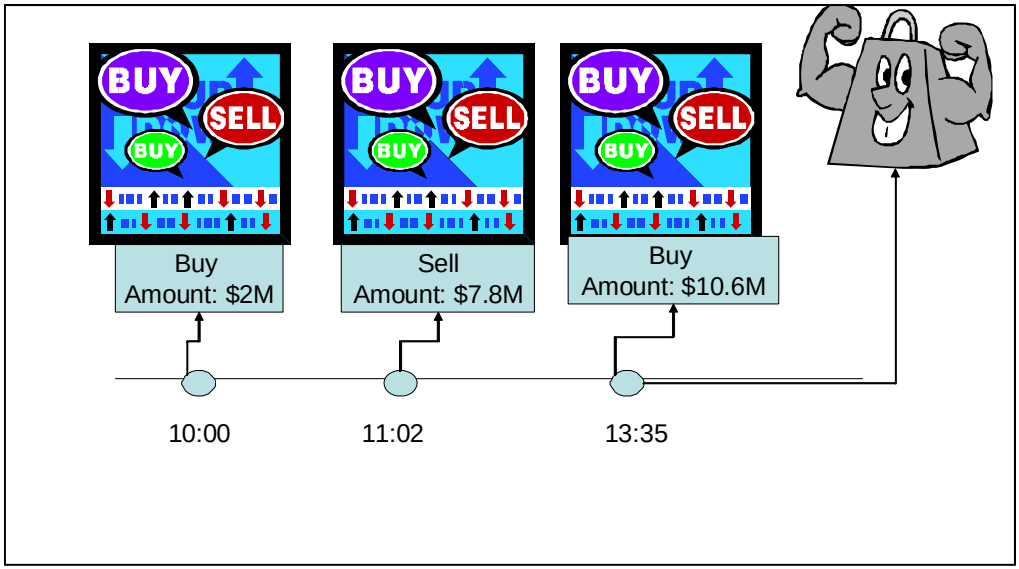


Figure 9.4. A trading example where the relevant event sett contains three events (two Buy events and one Sell event). The value max pattern will detect a match as the third event exceeds the $10M threshold.

The threshold assertion can use the binary relations >, <, =, ≥, ≤, ≠. The event types that participate in the participant set should all contain a numeric attribute with a name identical to the attribute mentioned in the assertion. As with the `count` pattern, this pattern can be matched before the full set of relevant events has been assembled.

To see the `max` pattern in use, suppose we are interested in identifying high-spending daily investors. We decide to look, each day, for an investor who makes either a security buy or a security sell transaction with a value of $10M or more. The participants set = {`security buy`, `security sell`} and the assertion is amount > 10M. Assume that a certain customer has made two buy transactions and one sell transaction in the same day, where the last transaction was for a sum of $10.6M. In this case the matching set consists of a single event, the `Security` Buy event which satisfies this condition. Note that we need a policy to determine what to include in the matching set if the maximal value is shared by two event instances. this example, if there is another event with amount of $10.6M.

Next we move to the value min pattern, which is the twin of the previous one.

THE VALUE MIN PATTERN

The `value min` pattern is very similar to the value max pattern; except that this time it is the minimal value of a specific numeric attribute that is tested against the threshold assertion. The assertion can use the relations: >, <, =, ≥, ≤, ≠.

> The value min pattern is satisfied when the minimal value of a specific attribute over all the relevant events satisfies the value min threshold assertion.

As with `value max`, all the event types in the participant set should have a numeric attribute with a name identical to the attribute mentioned in the assertion. Again, this pattern can be matched before the full set of relevant events has been assembled.

We can look at the same example in figure 9.4 and see what would happen using `value min`, with the assertion `amount > $5M`. This pattern would look for customers whose minimal transaction amount for the day is $5M. Note that the customer described in figure 9.4 does not match the pattern since this customer has a transaction with an amount of $2M which is less than the $5M threshold.

THE VALUE AVERAGE PATTERN

The `value average` pattern belongs to the same threshold family, but is somewhat different in its semantics. In this pattern it is the average (arithmetic mean) value of a specific numeric attribute over the event instances that is tested against the threshold assertion. Again the threshold assertion can use the relations >, <, =, ≥, ≤, ≠

> The value average pattern is satisfied when the value of a specific attribute, averaged over all the relevant events, satisfies the value average threshold assertion.

As with `value max` and `value min` all event types in the participants set should have a numeric attribute with a name identical to the attribute mentioned in the assertion.
The two main differences from the `value max` and `value min` patterns are:

- While the average can be calculated incrementally, the decision as to whether there is a pattern match can be done only when all the events have been collected together, typically when the context is terminated.
- If the pattern is satisfied then all events that formed the average are placed in the pattern matching set.
- For an example we can again refer to figure 9.4. Suppose that we have a value average pattern for the amount attribute, with a threshold assertion amount > $5M. In this case the average for amount is $6.8M so the pattern is matched.

The `functor` is a pattern that can be used to denote other functions that can be used. Examples of such functors are: variance, standard deviation, median. Many others are possible. We mention them for the sake of completeness and will not discuss these functors in details.

9.2.3 Relative patterns

The threshold patterns described above compare some statistical function of an event collection with some threshold value; relative patterns, on the other hand, are just concerned with finding the event that contains the minimal or maximal value of some numeric attribute within the relevant event set. There are two relative patterns: `relative min` and `relative max`. Their definitions are straightforward.

THE RELATIVE MAX PATTERN

The `relative max` pattern looks for the event instance that contains the highest value of a given attribute.

> The relative max pattern is satisfied by the event which has the maximal value of a specific attribute over all the relevant events.

This pattern could be used to find the bid events that propose the highest amount of money in an auction. Note that the matching set may contain multiple events since there may be multiple events that have the maximal value for the attribute.

The definition of the relative min pattern is similar.

THE RELATIVE MIN PATTERN

The `relative min` pattern looks for the event instance containing the lowest value of a given attribute.

> The relative min pattern is satisfied by the event which has the minimal value of a specific attribute over all the relevant events.

Going back to the example in Figure 9.4, the relevant event set contains three events. The `relative Min` pattern for the attribute Amount would select the Buy event that occurred at 10:00 as at only $2M this event contains the smallest value of Amount. The `relative max` pattern for the same attribute would select the Buy event that occurred at 13:35 with the Amount of $10.6M.

9.2.4 Modal patterns

Modal patterns are patterns that take an assertion and check to see if it is satisfied by the entire relevant event set or just some members of the set [1].

To explain modal patterns we will return to the world of call centers and problem handling and use the example shown in Figure 9.5. In this example problems can be reported through web, Email, phone and fax and are converted to instances of a single `Problem assignment` event type. In the course of the morning in question a service representative has been assigned four different problems. Each problem has a problem type, a severity (1 – critical, 2 – urgent, 3 – regular) and a customer type (gold or silver).

[1] The modal patterns described here are similar to the operators used in modal logic : necessity corresponds to the always pattern, and possibility to sometimes.

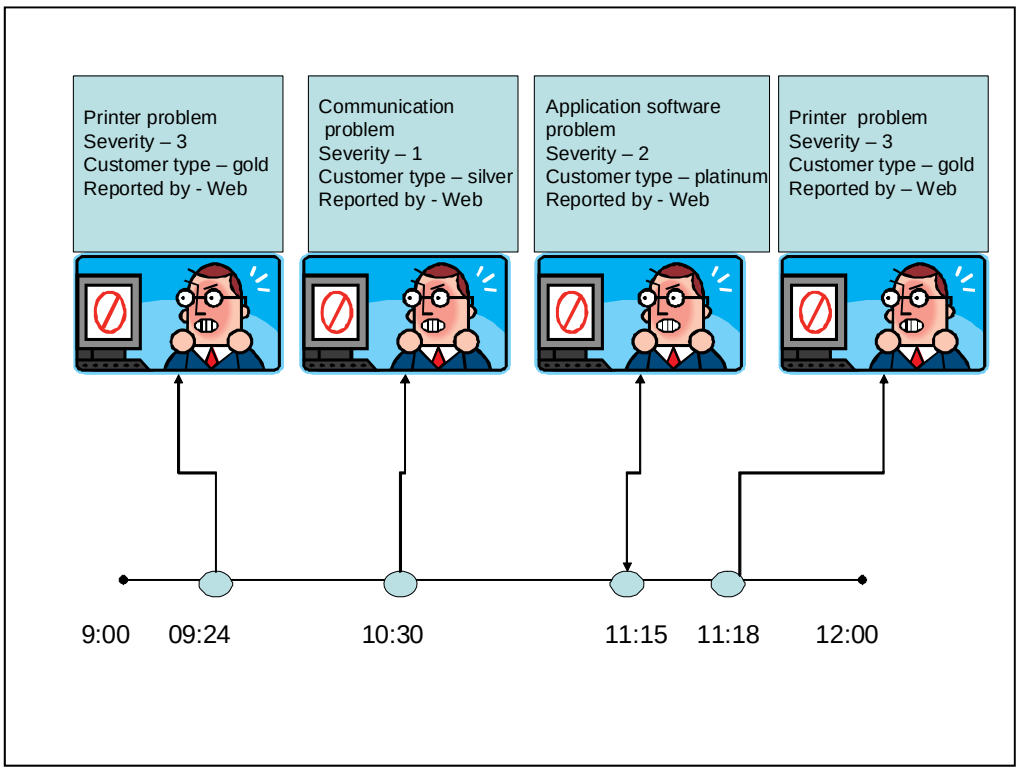


Figure 9.5 An example to illustrate modal patterns. In this example problems are assigned to a particular service representative, during the morning shift that spans between 9:00 – 12:00.

THE ALWAYS PATTERN

`Always` is a modal pattern which is matched if all event instances in the relevant event set satisfy some assertion

> The `always` pattern is satisfied when all the relevant events satisfy the always pattern assertion.

Looking back to figure 9.5, consider the following pattern specification (operating in the context of the morning shift context for this particular service representative)

```
Pattern: always
```

```
Participant set: {Problem assignment}
Pattern assertion: "reported by = Web"
```

In this case the pattern would detect a match as all problems were indeed reported by the Web. This pattern can be detected only when all relevant events are available. The pattern matching set contains all the relevant events.

THE SOMETIMES PATTERN

The `sometimes` pattern is another modal pattern as defined below.

> The `sometimes` pattern is satisfied when there is at least one relevant event that satisfies the sometimes pattern assertion

As you can see from the definition the `Sometimes` pattern generates a match if there is an event instance in the relevant event set that satisfies the assertion. Returning to figure 9.5, suppose we have the following pattern specification (operating in the context of the morning shift for this particular service representative)

```
Pattern:  sometimes
Participant set:  {Problem assignment}
Pattern assertion:  "Severity = 1"
```

This pattern is matched, as a severity 1 problem was assigned at 10:30. As this shows, this pattern can be detected incrementally and not necessarily at the end of the shift. The matching set that is produced by this pattern is determined by additional policy options. It could contain all matching events, just the first or just the last matching event.

THE NOT SELECTED PATTERN.

The `not selected pattern` is a different type of modal pattern. It can be thought as a second level modal pattern since it involves two matching steps. The first step involves matching using one or more patterns like those we have already discussed. The second step then takes the matching sets obtained from these patterns and looks to see if they include all of the original events instances or not.

> **Definition**
>
> The `not selected` pattern is satisfied when there is a relevant event which is not a member of any matching set of the patterns specified in the not selected assertion.

The signature for the non selected pattern has a special assertion that lists the first-step pattern or patterns that this pattern refers to.

Figure 9.6 shows an example of this pattern in use, as part of a used books trading application. The application has created a market for a popular textbook, and accepts two types of event: `book offer` events which contain a lower limit on the price, and `book order` events which have an upper limit on the price. During the course of the day the application uses an `all` pattern to match book offers to book orders At the end of the day, when trading has closed, there may be some unmatched events, either because there were more of one kind of event than the other, or because some orders or offers contained unrealistic price limits. In Figure 9.6 we can see that book offer 1 was matched against order 2, and book offer 2 was matched against order 3. This leaves book offer 3, order 1 and order 4 as unpaired events, and they thus fit the `not selected` pattern with respect to our `all` pattern. In this case the `not selected` pattern would produce a pattern matching set that consists of {book offer 3, book order 1, book order 4}.

This pattern is useful for purposes of handling exceptions of various kinds, referring to alternatives, or issuing alerts. For example you can use it to alert a customer that his orders are never matched as he always makes unrealistic bids.

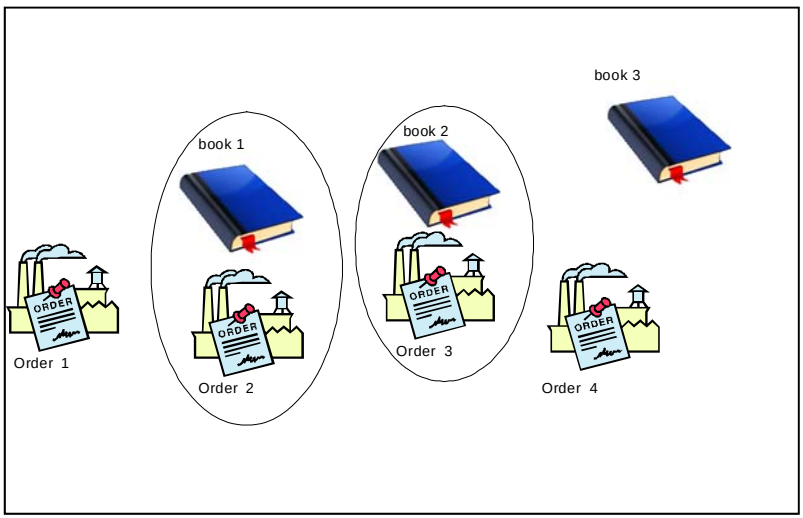


Figure 9.6 A used book selling application that shows use of the `not selected` pattern. Book offer events are shown at the top, and book orders at the bottom of the picture. Two orders and one offer are left unpaired, and thus match the not selected pattern.

This concludes the discussion of basic event patterns, next we move to look at dimensional patterns.

9.3 Dimensional patterns

Dimensional patterns are patterns that relate to the time dimension, to the space dimension or to a combination of both. While temporal patterns are very common, the spatial and spatio-temporal dimensions have not been explored in traditional event processing applications, but we observe that the use of these dimensions is growing and so we have included some examples of these patterns. As in other dimension types, we assume that the list of patterns will grow with time.

In this chapter we define patterns that assume that an event happens at a single point in time, and that its location is also abstracted as a point in space. Later in the book we will discuss more advanced patterns based on temporal intervals, and on spatial areas.

Temporal patterns are patterns in which time plays a major role. We start the discussion of temporal patterns by looking at the most common pattern, the sequence pattern.

9.3.1 The sequence pattern

The sequence pattern is similar to the all pattern, except that it requires the event instances to occur in a particular order.

> The sequence pattern is satisfied when the relevant event set contains at least one event instance for each event type in the participant set, and the order of the event instances is identical to the order of the event types in the participant set.

In most patterns the order of the event types that participate in the participant set is immaterial, however in the sequence pattern, the participant set becomes a participant sequence, and it is totally ordered.

Figure 9.7 shows a sequence example. The pattern is satisfied if a patient is released from hospital and then re-admitted to the hospital within 48 hours for the same reason as the original admission The participant sequence is <patient release, patient admission> in that order. The sequence pattern, like the all pattern, is a conjunction of the types in the participant set; however, in the all pattern the order is immaterial. As we saw with the temporal context discussion in chapter 7, there are several ways to define the ordering of the relevant event set (for example occurrence time, detection time or position in the stream) so we define an ordering policy parameter that specifies which approach is to be used. We discuss this further in section 9.4.

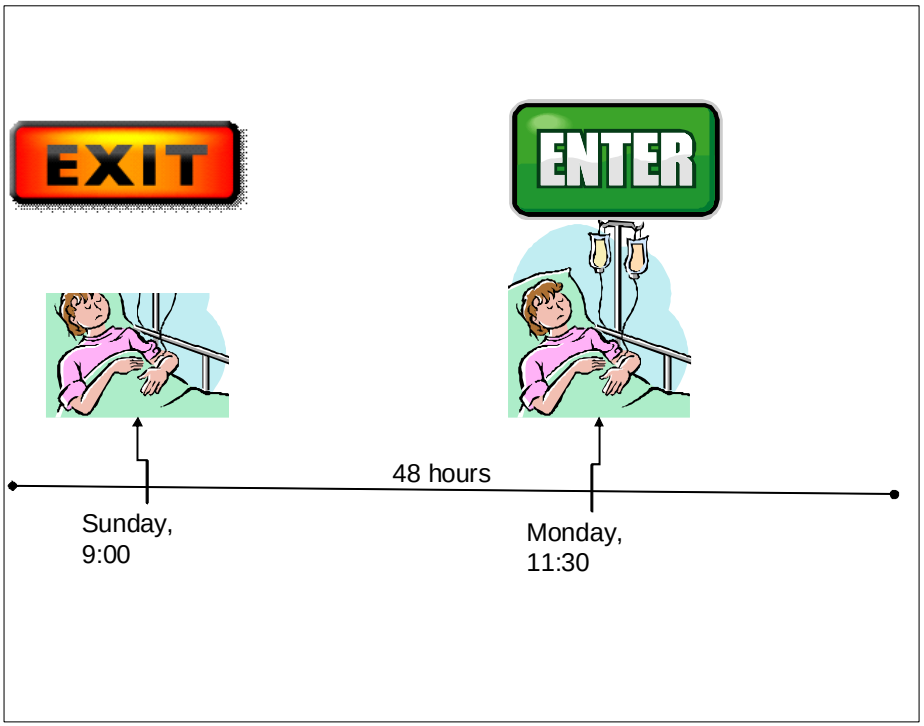


Figure 9.7. A sequence pattern example – a patient that is re-admitted to the hospital within 48 hours of having been discharged.

9.3.2 Trend patterns

Trend patterns are patterns that trace the value of a specific attribute over time. The participant set is always a singleton, as these patterns relate only to a single event type. In addition the instances of this event type must make up a time series, meaning that they must be temporally totally ordered. This order may be based either on occurrence time or detection time or position in the input stream (the ordering approach to be used is determined by the ordering policy that we will discuss in Section 9.4). We use the notation e1 << e2 to denote that input event instance e1 is before event instance e2 in this temporal order.

The trend patterns are: `increasing`, `decreasing`, `non increasing`, `non decreasing`, `stable` and `mixed`. We start by discussing the increasing pattern.

THE INCREASING PATTERN

The increasing pattern is satisfied when the value of a given attribute increases strictly monotonically as we move forwards through the sequence of relevant events.

> The `increasing` pattern is satisfied by an attribute A if for all the relevant events, e1 << e2 $\Rightarrow$[2] e1.A < e2.A

The remaining definitions are quite similar.

THE DECREASING PATTERN

The `decreasing` pattern is satisfied when the value of a given attribute decreases strictly monotonically as we move forwards through the sequence of relevant events.

> The `decreasing` pattern is satisfied by an attribute A if for all the relevant events, e1 << e2 $\Rightarrow$ e1.A > e2.A

THE STABLE PATTERN

The `stable` pattern is satisfied when the value of a given attribute does not change within the context. Note that in this case the order of the event instances is irrelevant.

> The `stable` pattern is satisfied by an attribute A if for all the relevant events, e1 << e2 $\Rightarrow$ e1.A = e2.A

THE NON INCREASING PATTERN

The `non increasing` pattern is satisfied when the value of a given attribute does not increase within the given context.

> The `non increasing` pattern is satisfied by an attribute A if for all relevant events e1 << e2 $\Rightarrow$ e1.A $\geq$ e2.A

Likewise we define the non decreasing pattern.

THE NON DECREASING PATTERN

This pattern is satisfied when the value of a given attribute does not decrease within the given context

> The `non decreasing` pattern is satisfied by an attribute A if for all relevant events e1 << e2 $\Rightarrow$ e1.A $\leq$ e2.A

The complement to all these patterns is the `mixed` pattern. This is matched if none of the trends defined so far is satisfied.

---

[2] The symbol $\Rightarrow$ denotes implication

THE MIXED PATTERN

The `mixed` pattern designates that the value of a given attribute both increases and decreases in different portions of the time series.

> The `mixed` pattern is satisfied by an attribute A, if the relevant event set contains event instances e1, e2, e3, e4 such that:
>
> e1 << e2 and e1.A < e2.A and e3 << e4 and e3.A > e4.A [3]

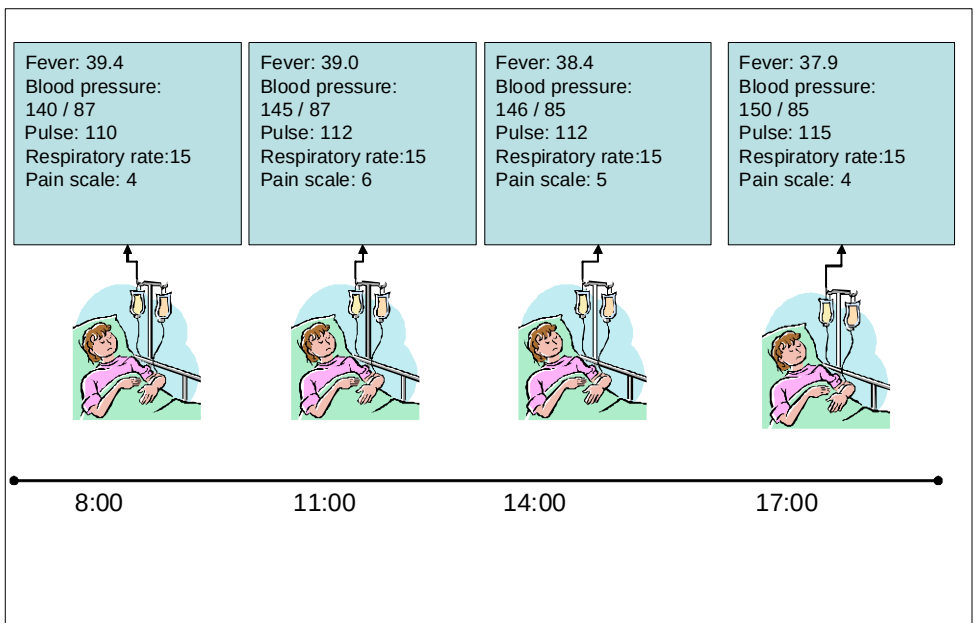


Figure 9.8 A health monitor example used to illustrate various temporal trend patterns.

We will use the example shown in figure 9.8 to illustrate these trend patterns. In this example a patient's vital signs, including a subjective pain scale, are taken every three hours.

Looking at the figure we see that the following patterns can be detected:

- The value of the fever attribute satisfies the `decreasing` pattern;
- The value of the systolic blood pressure attribute (the first of the pair) satisfies the

[3] e1, e2, e3 and e4 don't have to be four distinct instances, for example e2 could be the same event as e3.

`increasing` pattern

- The value of the diastolic blood pressure (the second of the pair) satisfies the `non increasing` pattern
- The value of the pulse attribute satisfies the `non decreasing` pattern
- The value of the respiratory rate attribute satisfies the `stable` pattern
- The value of the pain scale attribute satisfies the `mixed` pattern.

The only trend that may be detected without having all the event instances is the mixed pattern, all other patterns require the entire relevant portion of the time series, but it may makes sense for the EPA implementing a trend pattern to emit intermediate results.

Trend patterns are commonly used in stream processing systems which process continuous time series, typically in batches.

9.3.3 Spatial patterns

Spatial patterns are patterns that are satisfied based on distance between the locations of the events. They include the distance patterns: `min distance`, `max distance`, and `average distance`. These distance patterns can be either absolute or relative. Absolute distance patterns are concerned with the distance of an event's location from a fixed point, typically the location of a particular object. Relative distance patterns are concerned with the distances between events in the relevant event set.

Distance patterns can be implemented using the standard threshold patterns that we discussed in section 9.3.1, but for reasons of convenience, we present them here as patterns in their own right. We define the absolute distance patterns first.

THE MIN DISTANCE PATTERN

The `min distance` pattern deal with the distance between events and a given object.

> The min distance pattern is satisfied when the minimal distance of all the relevant events from a given point satisfies the min distance threshold assertion

In the Fast Flower Delivery example, we could discover whether there were any delivery vans less than 20 km from a given florist in the last five minutes. This can be done by using the min distance pattern and setting the threshold as ≤ 20 km.

In a similar fashion we define the max distance and average distance patterns

THE MAX DISTANCE PATTERN

The `max distance` pattern is similar, except that it is concerned with the maximal distance of all events from the given object.

> The max distance pattern is satisfied when the maximal distance of all the relevant events from a given point satisfies the max distance threshold assertion

The average distance pattern definition is next.

THE AVERAGE DISTANCE PATTERN

The average distance pattern again relates to the distance between events and entities.

> The average distance pattern is satisfied when the average distance of all the relevant events from a given point satisfies the average distance threshold assertion

Figure 9.9 uses part of the Fast Flower Delivery application to demonstrate these three patterns.

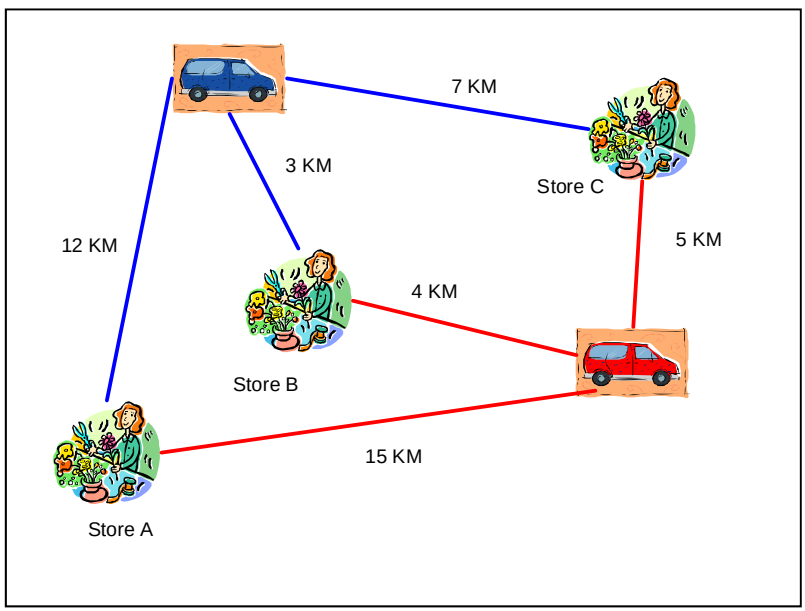


Figure 9.9. An example showing the location of two delivery vans and three flower stores in the Fast Flower Delivery application. This is to demonstrate absolute distance patterns

To keep the illustration simple, we assume that the relevant context is a sliding interval in a certain geographical area that contains GPS readings for the blue and red vans. The distances are calculated as part of this pattern evaluation relative to the fixed locations of stores A, B, C. Observing this illustration, here are examples of some supported patterns:

Min distance relative to Store A > 10 km
Max distance relative to Store B < 5 km
Average distance relative to Store C ≤ 7 km

Next we move to discuss relative distance patterns. These relate to distances between the locations of events in the relevant event set. The first of these is relative min distance.

THE RELATIVE MIN DISTANCE

The `relative min` distance, as the name suggests, evaluates the minimal distance among all pairs of events in the relevant event set.

> The `relative min distance` pattern is satisfied when the minimal distance between any two relevant events satisfies the min distance threshold assertion.

To show this in use, we will consider a law enforcement application that analyzes burglary reports looking for patterns of similar-looking burglary events. One hypothesis is that there could be a burglar who never commits two crimes in the same neighborhood, so as to camouflage his tracks. To look for this the application uses a `relative min distance` pattern to detect when there is a set of similar burglaries always separated by a distance of at least 20 km.

THE RELATIVE MAX PATTERN

The definition of the next pattern, relative max distance, is similar.

> The relative max distance pattern is satisfied when the maximal distance between any two relevant events satisfies the max threshold assertion

Using the burglary story again, the relative max distance pattern can be used to look for lazy burglars that always operate within a single neighborhood. We could for example look for a maximal distance of 5 km between similar burglaries.

THE RELATIVE AVERAGE DISTANCE PATTERN

Our third relative distance pattern looks at the average distance between events:

> The relative average distance pattern is satisfied when the average distance between any two relevant events satisfies the relative average threshold assertion

This pattern could be useful looking for a burglar who generally stays in a particular neighborhood, but now and then takes a journey further.

Figure 9.10 illustrates these three relative patterns.

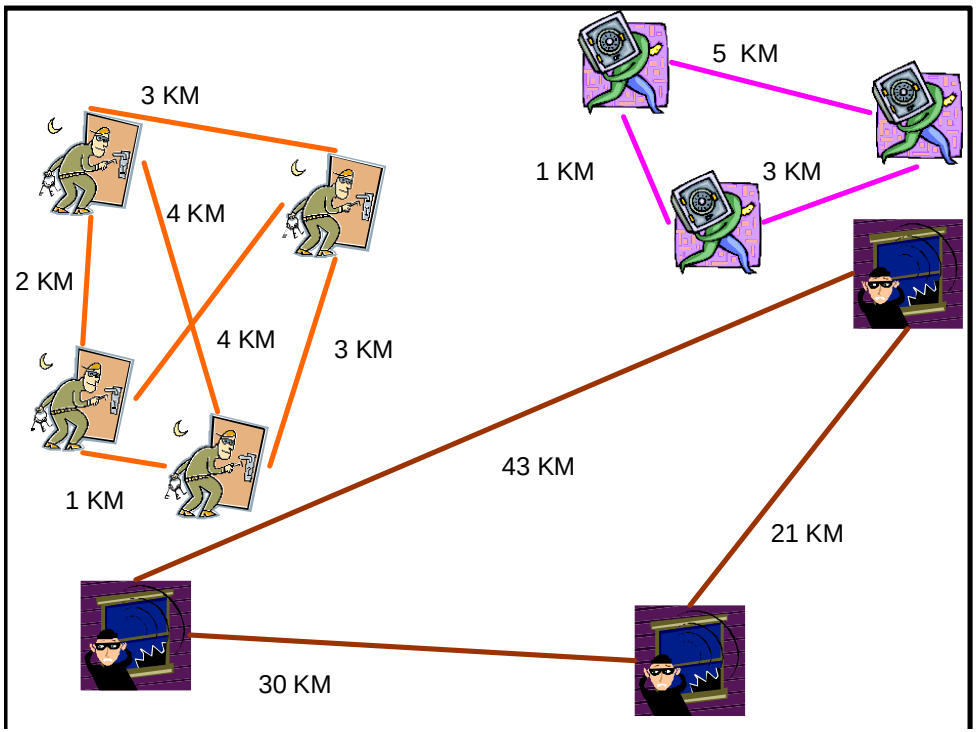


Figure 9.10 Examples to demonstrate three relative distance patterns. Going clockwise from the top right they are relative max distance, relative min distance and relative average distance.

Figure 9.10 illustrates three different segmentation-oriented contexts. One of them relates to burglaries where a door is broken, one relates to burglaries where the burglar gains entry by breaking a window, and the third includes the theft of heavy appliances. Looking at the figure you can see that these contexts satisfy the following patterns:

- The relative max distance pattern with threshold < 5 km is satisfied in the door breaking context
- The relative min distance pattern with threshold > 20 km is satisfied in the window breaking context
- The relative average distance pattern with threshold ≤ 3 km is satisfied in the heavy lifter context.

9.3.4 Spatiotemporal patterns

Spatiotemporal patterns look at time series of events and determine spatial trends over time. Patterns are of the following types: moving in a consistent direction, moving in mixed direction, stationary, moving closer to, moving away from.

As with temporal trend patterns we assume that the participant set contains just one event type, so all the relevant events are of a single type. Moreover the patterns require that the relevant events themselves constitute a time series, meaning that they are temporally totally ordered (as with temporal trend patterns the definition of this order depends on the ordering policy, and we will return to that in section 9.4). We will use the notation: e1 << e2 to mean that event instance e1 comes before e2 in this ordering.

THE MOVING IN A CONTSANT DIRECTON PATTERN

This pattern is actually a family of patterns, such as `moving north`, or `moving south`. For example the `moving south` pattern would be satisfied by a vehicle that is transmitting GPS readings of its position while it is traveling from Bologna to Florence

> The `moving in a constant direction` pattern is satisfied if there exists a direction from the set {north, south, east, west, northeast, northwest, southeast, southwest} such that for any pair of relevant events e1, e2 we have e1 << e2 $\Rightarrow$ e2 lies in that direction relative to e1.

The complementary pattern is the pattern moving in a mixed direction, discussed next.

THE MOVING IN A MIXED DIRECTION PATTERN

This pattern is complementary pattern indicating that no consistent direction can be found among the relevant events in the time context being considered

> The `moving in a mixed direction` pattern is satisfied if none of the eight `moving in a consistent direction patterns` is satisfied

THE STATIONARY PATTERN

This pattern is self-explanatory

> The `stationary` pattern is satisfied if the location of all relevant events is identical.

. We have one final spatiotemporal pattern, and this one refers to an external object.

THE MOVING TOWARD PATTERN

This is a pattern that determines movement towards some object

The `moving toward` pattern is satisfied when for any pair of relevant events e1, e2 we have e1 << e2 ⇒ the location of e2 is closer to a certain object then the location of e1.

Note that this pattern may be true for several objects at the same time

We illustrate these patterns in figure 9.11, which shows movements of aircraft (the events in question are periodic position reports from these aircraft).

In figure 9.11, the helicopter flies across Central America from the Pacific Ocean to the Atlantic Ocean. This satisfies the `moving consistently eastward` pattern. It also satisfies `moving toward` France (and for that matter also `moving toward` Germany). The biplane is going around in circles over the Pacific Ocean, and so it satisfies the `mixed direction` pattern, while the small helicopter is hovering `stationary` over Greenland.

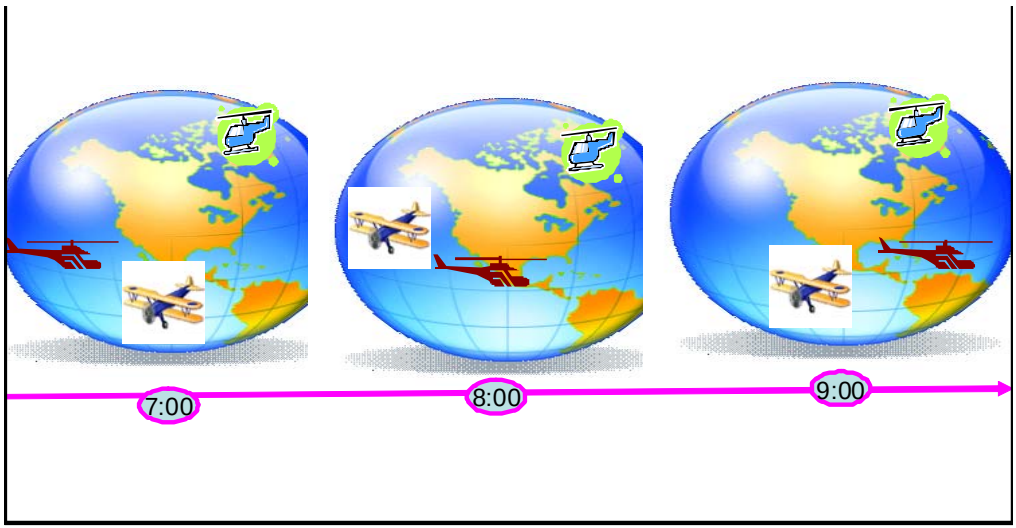


Figure 9.11 illustrations of spatiotemporal patterns where the events are reports of aircraft location.

One can imagine many more patterns, for example patterns that refer to medians, quartiles and other statistical measurements. The list given in this chapter contains the most commonly-used patterns, and will probably grow over time. We now turn our attention to Policies. These are integral part of pattern interpretations and they are discussed next.

9.4 Pattern policies

Some of the patterns that we have looked at in the previous sections can be interpreted in more than one way. Consider for example the `any` pattern, which is satisfied if any of the specified event types occurs. If there's just one qualifying event instance then it is obvious

that the matching set should contain just that event instance, but if there's more than one matching event then things aren't so straightforward. You might imagine that the matching set should contain all the events that match, and for some applications that is indeed the right things to do. However you will recall that in section 9.1.2 we showed how you we use the `any` pattern in the Fast Flower Delivery application to perform automatic matching between `Delivery Bids` and `Delivery Bid Requests`. In this case we need to ensure that there is no more than a single match. Furthermore, for reasons of fairness, we want the matching operation to select the first bidder.

The role of a policy is to disambiguate the semantics of a pattern matching operation, answering questions such as "how many events should be included in the matching set?" or "what kind of temporal ordering is to be used?" In this chapter we discuss four kinds of policy:

- Repeated type policy: Determines the semantics when the relevant event set contains multiple events of the same type.
- Order policy: Determines how temporal order is defined
- Matching cardinality policy: Determines how many matching sets are produced within a single context
- Consumption policy: Determines whether an event that is included within one matching set can be included in another matching set.

Note that different policies may apply to different members of the relevant event set.

We now discuss these policies in depth, and as we do so we will recommend a default for each kind of policy, which will often be sufficient, however there are occasions when different behavior is required.

9.4.1 Repeated type policies

Definition

A *repeated type* condition occurs when the relevant event set contains more than one event instance of the same event type.

This is normal for some patterns, including patterns that operate on time series like the spatiotemporal patterns. However there are other patterns in which the presence of two events of the same type can cause semantic ambiguity. Consider the `all` pattern and recall the example illustrated in Figure 9.2, where the participant set consists of {flight reservation, car reservation, hotel reservation}. In figure 9.12 we see the same example, but this time there are two events of type `flight reservation`, two events of type `car reservation`, and three events of the type `hotel reservation`. In the `all` pattern each matching set has to include a single instance of each of these event types, so there are

twelve possible matching sets that could be created. The question is which one of them should really be generated? You could take a set theoretic approach and say that the answer is the Cartesian product that contains all twelve possibilities. This could be the answer in certain cases, but typically in event processing systems the intuitive interpretation of the all pattern is not a Cartesian product, but instead a conjunction of individual events.

Definition

A *Repeated type policy* is a semantic abstraction that defines the behavior when a repeated type condition occurs in a pattern's relevant event set. The possible policies are: override, every, first, last, with largest value, with smallest value

A pattern can specify a different repeated type policy for each of the event types in the participant set.

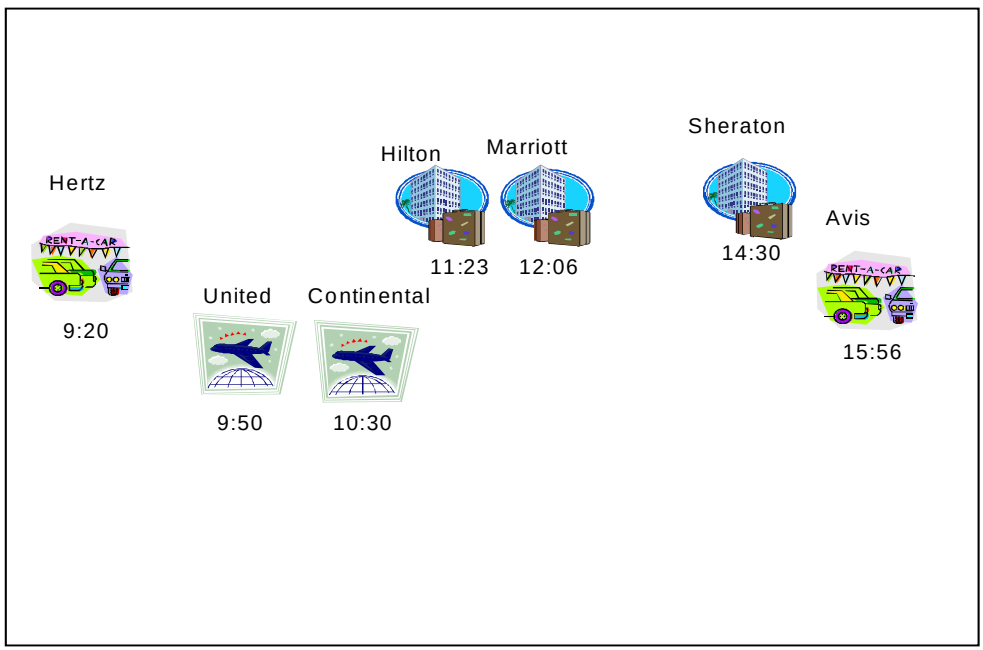


Figure 9.12 An example of the `all` pattern where there are multiple events of the same type

The interpretation of these various policies is as follows:

- Override: the result set keeps at most one event of each event type. Each time a new event instance is encountered it overrides any previous instances of the same event type. Overridden event instances are erased from the relevant event set. This is our recommended default.

- Every: every event instance is kept, and all possible matching sets are produced. This is the Cartesian product interpretation.
- First: all instances are kept in the relevant event set, but only the first instance of each type is used for matching.
- Last: only the last instance of each event type is used for matching. This is different from the override policy in that previous instances are not removed from the relevant event set and can still be used.
- With Maximal value [attribute name]: the event or events with the maximal value of a given attribute are used for matching.
- With Minimal value [attribute name]: the event or events with the minimal value of a given attribute are used for matching.

These policies don't say how often or when the matching operation takes place. That's decided by the cardinality policy that we will meet in section 9.3.

In the example illustrated in figure 9.12:

- If there is an `override` policy for each event type, then at any point in time there will be at most one relevant event for each event type. At the end of the time interval there is a single combination of events {Continental flight reservation, Sheraton hotel reservation, Avis car reservation}
- The `every` policy would generate twelve matching sets containing all twelve combinations.
- The `first` policy semantics is still ambiguous, and additional policies (or the use of defaults) are needed to disambiguate it. If there is a single matching set for the context, then the matching set is {Hertz car reservation, Hilton hotel reservation and United flight reservation}. However, there is another matching set that may be generated later that includes {Marriott hotel reservation, Avis car reservation and Continental flight reservation}, we'll discuss this issue again when talking about matching cardinality policy.
- Likewise the `last` policy is ambiguous and needs to be disambiguated by other policies (or using defaults). We leave the exercise of finding the possible matching sets to the reader.
- A `Minimal value` policy could be used to select the cheapest option for each of the three reservations.

Next we discuss order policies.

9.4.2 Order policies

Definition

An *order policy* is a semantic abstraction that defines the meaning of the << temporal order of the event instances in the relevant event set. The possible policies are: by occurrence time, by detection time, by user-defined attribute, or by stream position.

The order policy is applicable to all temporal or spatiotemporal patterns. The possible policies are:

- By occurrence time: in this case the order relation is determined by comparing the occurrence time attribute in each event instance. This means that the order should reflect the order in which the events happened in reality. This is our recommended default, if occurrence time is supported.
- By detection time: in this case the order relation is determined by comparing the detection time attribute in the event instances. This means that the order should reflect the order in which events are detected by the event processing system. Note that this order may not be identical to the order in which events happened in reality.
- By user-defined attribute: Some event payloads contain a timestamp, sequence number or some other attribute that increases over time, and this can be used to determine the order. For example the Delivery Request events in the Fast Flower Delivery application could be ordered using their Delivery Time attribute.
- By stream position: In this case the order to be used is the order in which the events are delivered to an EPA from the channel that feeds it. Some channel implementations are designed so that this order is the same as the order in which events were delivered to the channel.

Note that the first three of these policies don't guarantee that the order is unique as there could be two or more relevant events that have the same timestamp or user-defined attribute value.

9.4.3 Cardinality policies

Definition

A *cardinality policy* is a semantic abstraction that controls how many matching sets are created and also determines the time when they are created. The possible policies are: single, single deferred, unrestricted and bounded.

The following policies are detailed and demonstrated through the travel reservation example that we saw in figure 9.12. The various policies are:

- Single: The matching set is generated as soon as the matching conditions are satisfied. When this has been done no further action is performed within this context partition, so

no more matching sets are generated.

- Single deferred: As with the single policy, the single deferred policy also results in no more than one matching set per context partition. The difference is that for the Single deferred policy, the matching process is not performed until the entire relevant event set has been assembled (in other words until all the events in the context partition have been processed). This is the only possible policy for those patterns that require processing of the entire relevant event set, e.g. absence or consistently moving north.
- Note that the two policies may yield different results, as demonstrated in Figure 9.13
- Unrestricted: Under this policy, there are no restrictions on the quantity of matching sets. A matching set is generated every time that the matching conditions are satisfied. This is our recommended default.
- Bounded: Under this policy, there is an upper bound on the number of matching sets that can be generated within a context partition, e.g. Restrict to no more than five matching sets.

We use figure 9.13 to help show the effect of these policies.

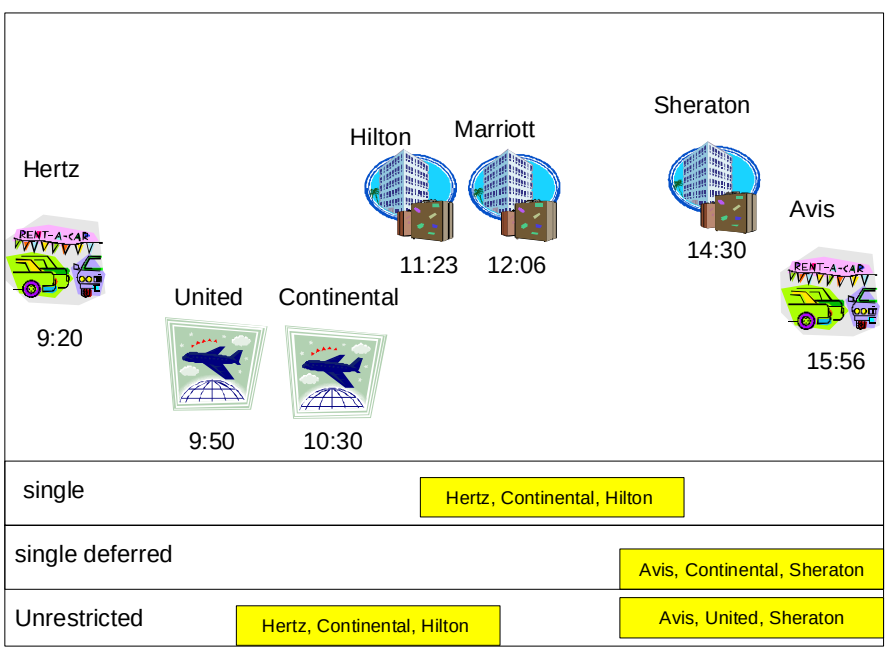


Figure 9.13 Illustration used to show the effect different cardinality policies

Figure 9.13 illustrates these different cardinality policies, assuming that the repeated type policy is Last.

- Under the single policy, there is a single matching set that is created as soon as all the three event types have been detected. This happens in 11:23, when the Hilton hotel reservation is made. Note that there have been two airline reservations events at this point, so under the Last policy, it is the Continental reservation that is selected.
- Under the single deferred policy, the matching set is not created until after the last event has been received. As the repeated type policy is Last, this means that the matching set consists of the Avis, Continental and Sheraton reservations, being last instances of their respective event types. You can see that this has produced a different result from the single policy, since under the single policy the matching set is created at a time when only a subset of the relevant events has been detected.
- Under the unrestricted policy, the first matching set is the same as the one generated by the single policy, namely {Hertz, Continental, Hilton}. Processing then continues, but what happens next depends on whether it is permissible to use the same event instance in two matching sets or not. If it is permissible, then three more matching sets will be generated, the last of them being the same as the one generated under the single deferred policy. However in figure 9.13 we have made the assumption that each reservation can participate in only one matching set, The Hertz reservation has been used up, so no more matching sets get generated until the Avis reservation is processed. You will see that this second matching set is {Avis, United, Sheraton} even though the United reservation came earlier than the Continental one, since the United reservation was also used in the first matching set.

We resolve the ambiguity that we observed in the unrestricted policy example, by introducing our fourth and final policy type – a consumption policy.

9.4.4 Consumption policies

Definition

A *consumption policy* is a semantic abstraction that defines whether an event instance is consumed as soon as it is included in a matching set, or whether it can be included in subsequent matching sets. Possible consumption policies are: consume, reuse and bounded reuse.

The consumption policies are rather straightforward:

- Consume: under this policy, an event instance is consumed when it is included in a matching set. This means that is removed from the relevant event set and so it cannot take part in any further matching for this particular pattern within the same context. This is our recommended default.
- Reuse: under this policy, an event can participate in an unrestricted number of matching

sets.

- Bounded reuse: under this policy, the number of times in which an event can be reused for a particular pattern within the same context.

Recall the example illustrated in figure 9.13. in the unrestricted case, the Continental reservation is consumed if the consumption policy is consume, leaving the United flight reservation as the last (and only) flight reservation; however, if the consumption policy is reuse, then the Continental flight reservation is the one selected for the matching set.

To conclude this section, policies are vital to disambiguate the patterns' semantics in a fashion that leaves the meaningful space of possibilities. The defaults that we have recommended are adequate for many applications, however some applications may need to use different policies.

9.5 Patterns reference table

To summarize the previous sections we bring information about all the patterns that we have mentioned into a single table.

Table 9.1 Event pattern reference table

Category	Pattern	Pattern Assertion	Requires all relevant events?	Singleton Participant set?
Basic	All			
	Any			
Threshold	Count	Count threshold	X	
	Value max	Value max threshold	(X)[4]	
	Value min	Value min threshold	(X)	
	Value average	Value average threshold	X	
Relative	Relative min		X	
	Relative max		X	
Modal	Absence		X	
	Always	Always assertion	X	

[4] (X) means – true for certain cases.

	Sometimes	Sometimes assertion		
	Not selected		X	
Temporal	Sequence			
	Increasing	Evaluated attribute	X	X
	Decreasing	Evaluated attribute	X	X
	Stable	Evaluated attribute	X	X
	Non increasing	Evaluated attribute	X	X
	Non decreasing	Evaluated attribute	X	X
	Mixed	Evaluated attribute	X	X
Spatial	Min distance	Min distance threshold	(X)	
	Max distance	Max distance threshold	(X)	
	Average distance	Average distance threshold	X	
	Relative Min distance	Min distance threshold	(X)	
	Relative Max distance	Max distance threshold	(X)	
	Relative Average distance	Average distance threshold	X	
Spatio-temporal	Moving in consistent direction	Direction	X	X
	Moving in mixed direction		X	X
	Stationary		X	X
	Moving Towards		X	X

As indicated before, this set of patterns is extendable, and will be updated periodically on the book's website. We will now look at some of these patterns being used in the Fast Flower Delivery application.

9.6 The Fast Flower Delivery patterns

In this section we express all the patterns used in the Fast Flower Delivery application. Each of these patterns will be presented in a separate listing, followed by comments where required.

Listing 9.1 automatic matching

```
Pattern    Pattern  Context   Participant  Assertions    Policies
```

Name	Type		Set		
Automatic Matching process	Any	Bid Interval	Delivery Bid	Delivery Bid. Committed Pick Up time < Bid Request. Required Pick Up Time + 5 minutes	Repeated type = first; cardinality = single

This pattern generates a matching set that contains the first Delivery Bid whose committed pick up time matches the required pick up time. It employs two policies, the repeated type policy selects the first bidder, while the cardinality policy guarantees a single match.

Listing 9.2 No bidders

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
No bidders	Absence	Bid Interval	Delivery Bid	Delivery Bid. Committed Pick Up time < Bid Request. Required Pick Up Time + 5 minutes	

This is an example of time-out detection. It generates an event if no bidders that satisfy the Pick Up assertion were detected during the bid request interval context.

Listing 9.3 Assignment not done

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Assignment not done	Absence	Response Interval	Manual Assignment		

This is a time-out detection, indicating that the manual assignment decision was not performed on time.

Listing 9.4 Pick up alert

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Pick up alert	**Absence**	**Pick up Interval**	**Pick up Confirmation**		

This is a time-out detection indicating that pick up was not done on time.

Listing 9.5 Delivery Alert

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Delivery alert	**Absence**	**Delivery Interval**	**Delivery Confirmation**		

This is a time-out detection, indicating that the delivery was not done on time.

Listing 9.6 Ranking Increase

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Ranking Increase	**Absence**	**Driver Evaluation**	**Delivery Alert**		

This pattern detects drivers who did not have any delivery alerts within the driver evaluation context. It is used to decide whether to give the driver a ranking increase.

Listing 9.7 Ranking decrease

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Ranking Decrease	**Count**	**Driver Evaluation**	**Delivery Alert**	**> 5**	

This pattern detects drivers who had more than five delivery alerts within the driver evaluation context.

Listing 9.8 Improve note

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Improve note	**Sequence**	**Forever**	**Ranking Decrease, Ranking Increase**		**Repeated type = override**

This pattern detects when a driver had a ranking increase after a ranking decrease. The override guarantees that there will always be a single one of each of these events in the relevant events set. Forever is a universal context that is always true.

Listing 9.9 permanent weak driver

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Permanent Weak Driver	**Always**	**Monthly Driver**	**Daily Assignment**	**Assignments Number < 5**	

This pattern detects when a driver has had fewer than five assignments in each of his working days during the month.

Listing 9.10 Idle Driver

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Idle Driver	Sometimes	Monthly Driver	Daily Assignment	Assignments Number = 0	

This pattern detects that a driver had at least one working day during the month without any assignment.

Listing 9.11 Consistent Weak Driver

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Consistent Weak Driver	Always	Monthly Driver	Daily Assignment, Daily Statistics	Assignments Number < Daily Mean – 2 * Daily STDV	

This pattern detects that a driver has a consistently low number of assignments. Note that this assertion employs two derived event types, `Daily Assignment`, and `Daily Statistics`.

Listing 9.12 Consistent strong driver

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Consistent Strong Driver	Always	Monthly Driver	Daily Assignment, Daily Statistics	Assignment Number > Daily Mean + 2 * Daily STDV	

This pattern detects that a driver has a consistently high number of assignments. Note that this assertion employs two derived event types, both of them derived, `Daily Assignment` and `Daily Statistics`.

Listing 9.13 Improving Driver

Pattern Name	Pattern Type	Context	Participant Set	Assertions	Policies
Improving Driver	Non Decreasing	Monthly Driver	Daily Assignment	Assignments Number	

This pattern detects drivers whose daily assignment rate stays level or increases over the month, so as to designate them as improving. This is done on a monthly basis.

These examples from the Fast Flower Delivery application don't cover all types of pattern that we have mentioned in this chapter. We hope that solving the exercises at the end of the chapter will provide the interested reader an opportunity to experiment with additional patterns.

9.7 Pattern detection in practice

In this section we show some examples of pattern detection in current event processing languages, note that full details about the languages can be found in the book's website

Figure 9.14 shows an Apama example of finding the five highest bidders from the Fast Flower Delivery example:

```
386                 // if manual, the system selects the 5 highest ranked bidders
387                 // and send them to the store
388                 if not isAuto then {
389                     sequence<integer> keys :=bids.keys();
390                     keys.sort();
391                     keys.reverse();
392                     integer rank, i:=0;
393                     DeliveryBid bid;
394                     for rank in keys {
395                         if i=5 then {break;}
396                         for bid in bids[rank] {
397                             if i=5 then {break;}
398                             route RankedDeliveryBid(bid.requestId, bid.store, bid.driver,
399                                   bid.committedPickUpTime, dr.requiredDeliveryTime);
400                             i:=i+1;
401                         }
402                     }
```

Figure 9.13 An example from Apama of how to find the five highest bidders from the Fast Flower Delivery Example.

Listing 9.14 shows the Rulecore code for the No Bidders alert.

Listing 9.14 Rulecore example of the No Bidders alert

```
Rule Definition
<Rule name="NoBidders" limit="10000" evalMode="once" level="2">
  <Description>This is rule NoBidders</Description>
  <Initialize>
    <Assert>
      <Event>
        <base:XPath>sim:EventDef[@eventType="BidRequest"]</base:XPath>
```

```
      </Event>
    </Assert>
  </Initialize>
  <Views>
    <ViewRef name="NoBidders">
      <base:XPath>sim:ViewDef[@name=”NoBidders”]</base:XPath>
    </ViewRef>
  </Views>
  <Situations>
    <SituationRef name="NoBidders">
      <base:XPath>sim:SituationDef[@name="NoBidders"]</base:XPath>
    </SituationRef>
  </Situations>
  <Actions>
    <SituationDetected situationName="NoBidders">
      <ActionRef name="NoBidders" eventVisibility="external">
        <base:XPath>sim:ActionDef[@name="NoBidders"]</base:XPath>
      </ActionRef>
    </SituationDetected>
  </Actions>
</Rule>

View Definition
<View name="NoBidRequests">
  <Description>This is view NoBidRequests</Description>
  <Properties>
    <Type>
      <Event>
        <base:XPath>sim:EventDef[@eventType="BidRequest"]</base:XPath>
      </Event>
      <Event>
        <base:XPath>sim:EventDef[@eventType="DeliveryBid"]</base:XPath>
      </Event>
      <Event>

<base:XPath>sim:EventDef[@eventType="DeliveryRequestCancellation"]</base:XP
ath>
      </Event>
    </Type>
    <Match>
      <Property name="RequestId">
    </Match>
  </Properties>
</View>

Situation Definition
<SituationDef name="NoBidders">
  <Detector>
    <All>
      <After timeframe=”00:02:00”>
        <Not>
          <EventPickup keep="last" see="new" name="bid" evalMode="once">

<base:XPath>sim:Views/sim:View/sim:Events/sim:Event[@eventType="DeliveryBid
"]</base:XPath>
          </EventPickup>
```

```
        </Not>
      </After>
      <Not>
        <EventPickup keep="last" see="new" name="cancel" evalMode="once">

<base:XPath>sim:Views/sim:View/sim:Events/sim:Event[@eventType="DeliveryReq
uestCancellation"]</base:XPath>
        </EventPickup>
      </Not>
    </All>
  </Detector>
</SituationDef>

Action Definition
<ActionDef name="NoBidders">
  <Event>
    <EventDef>
      <base:XPath>sim:EventDef[@eventType="NoBiddersAlert" and
@eventClass="user"]</base:XPath>
    </EventDef>
    <Body>
      <XsltBuilder>
        <Stylesheet><![CDATA[ <?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:user="http://www.rulecore.com/2008/user"
xmlns:base="http://www.rulecore.com/2008/base" version="1.0">
  <xsl:template match="child::*">
    <base:EventBody>
      <xsl:for-each
select="user:Views/user:View[@default='true']/user:Properties">
        <user:RequestId>
          <xsl:value-of
select="descendant::user:MatchedProperty[@name='RequestId']/user:Value/chil
d::text()"/>
        </user:RequestId>
      </xsl:for-each>
   </base:EventBody>
  </xsl:template>
</xsl:stylesheet>
    </Body>
  </Event>
</ActionDef>
```

Figure 9.14 shows the Streambase example for automatic matching.

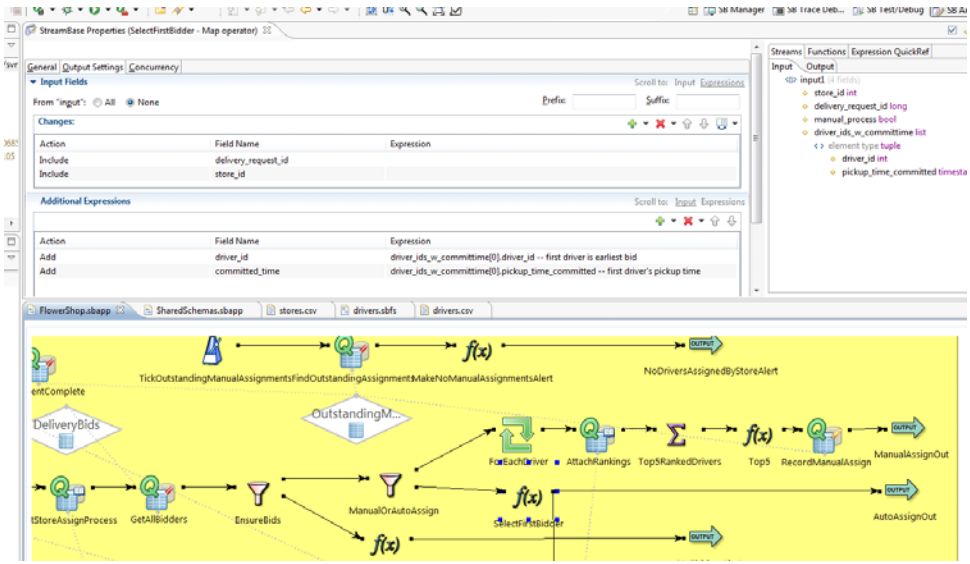


Figure 9.14 Streambase example for the automatic matching pattern.

The following snippet shows the Bid Alert in Esper, using SQL oriented programming.

```
/**
* No bid after 2 mins of a request
*/
insert into AlertW(requestId, message, driver)
select d.requestId, "no bidder", ""
from pattern[
every d=DeliveryRequest -> (timer:interval(120 sec) and not
DeliveryBid(requestId = d.requestId))
];
```

As you noticed, there are several different programming styles to express the same functionality, in chapter 10 we survey the various programming styles. We now summarize this chapter.

9.8 Summary

In this chapter we discussed the notion of event processing patterns, the central concept in contemporary event processing. We explained the idea of patterns and gave formal definitions of the terms *pattern*, *participant set* and *relevant event set*. We then discussed a number of pattern types: basic patterns, modal patterns and dimensional patterns. We also introduced the notion of pattern matching policies to tune up the semantics of patterns and discussed a number of these policies. .

This chapter concludes this part of the book, the "deep dive" into all the concepts and building blocks used when constructing event-based applications. The final part of this book deals with some advanced topics related to implementing applications and looks at some future directions of event processing

Additional reading

David Luckham: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Professional May 2002.

http://www.amazon.com/Power-Events-Introduction-Processing-Distributed/dp/0201727897/ref=sr_1_2?ie=UTF8&s=books&qid=1258816511&sr=8-2

This book introduces the notion of event patterns, and the Rapide language for event patterns.

Opher Etzion: Event processing architecture and patterns, Tutorial in DEBS 2008.
http://www.slideshare.net/opher.etzion/tutorial-in-debs-2008-presentation

This is a tutorial about event processing patterns in the form of slides presentation.

Exercises

1. Add three more EPAs to the Fast Flower Delivery example that use patterns that were not used in section 9.7.
2. Which pattern would you use if the only thing the need to be done is an assertion over different attributes of different events, such as: E1.A > E2.B > E3.C
3. What are the relationships among these event collections: the collection of input events to the EPA, the matching set, the collection of output events?
4. What are the relationships among these event type collections: the collection of input event types, the participant set, the collection of output event types
5. What is the difference between relevance assertions and pattern-specific assertions?

6. The set of patterns we have given is not minimal. If you were given the goal of reducing the set of patterns by expressing some of the patterns in terms of other patterns, which patterns could you have removed in this way?
7. Can you find think of new EPA related to the Fast Flower Delivery application that applies the `Not Selected` pattern? Write the exact pattern signature, and explain what it could be used for.
8. The `Not Selected` pattern is a second level modal pattern since it relates to the pattern detection process itself. Can you think of other useful second level modal patterns?
9. Is it useful to allow entities that are external to the EPA to query the internal state of the pattern matching process (for instance to see the collection of relevant events that have arrived so far and that have not yet been matched)? If yes, to what purpose? Show an example.
10. A higher level pattern, or template, is a construct that packages a single or composed pattern with parameters. State how you could define time out as a higher level pattern. Can you find another such higher level pattern, looking at the patterns of the Fast Flower Delivery example? Can you think of additional cases outside this example that using this idea would be useful?
11. List the cases in which the patterns value max and value min can be evaluated incrementally.
12. For some patterns all the relevant events need to be present in order for it to be possible to evaluate the pattern. The meaning of "all" in that sentence is clear when a pattern is being used with a temporal context, but what does it mean in the case of a spatial only context? Is there a way to know that the relevant set is complete in that case?
13. For the sequence pattern, show examples where it makes sense to order the events according to each ordering policy.
14. Can you think of additional spatiotemporal patterns? If so, define such patterns.
15. Show example where the mixed trend pattern could be used
16. Explain the benefit of externalizing policies to be separate entities.
17. What is the difference between the override and last synonym policies? Give an example where they produce the same result and an example where they produce different results.
18. What is the difference between the single and single deferred consumption policies? Give an example where they produce the same result, and an example where they produce different results
19. There are some patterns that require that the participant set to be a singleton. What do you think is the rationale for this restriction?

10

Engineering and implementation considerations

"In theory, there is no difference between theory and practice; in practice, there is."
- Chuck Reid

We have devoted most of this book to explaining the principles of event processing, while giving some examples that show these principles being used in practice. In this chapter we change our emphasis and focus explicitly on the implementation of event processing applications. The implementation related topics we discuss are: language styles, non-functional properties and performance optimizations. These are the major engineering topics behind implementation of event processing applications.

10.1 Event processing programming in practice

At the time of writing, there are no programming language standards for event processing. There are various programming styles, and various approaches within these programming styles. The building block approach that we use in this book is a kind of meta-level language and, as you can see from the different code samples we have given, there are various ways to implement each of these building blocks. In this section we survey some of the most common event processing programming styles, both the style of the language itself and the type of development environment used with it.

In this section we will look at two styles, which we term the *stream-oriented* style and the *rule-oriented* style. There is a third style, the *imperative style*, where the logic is coded in a C or Java style language. There are several languages like this, but no standard way to

express them, so we refer the reader to examples of each language[1]. We also survey some different types of development environment.

This section is based on the tutorial made by the EPTS Language Analysis group[2] in July 2009.

10.1.1 Stream oriented programming style

The stream oriented programming style is rooted in dataflow programming. In essence a dataflow graph is a directed graph consists of nodes and edges, the nodes represent processing elements, while the edges represent data flowing between these nodes. The paradigm is one of continuous queries, sometimes called *operators*, constantly running in the nodes, while their results flow through the edges in the data flow graph. Note that the EPN discussed in this book can be implemented as a data flow graph.

The languages used to describe the queries are inspired by SQL and relational algebra, though not all of them are actually based on SQL. As noted when we discussed stream computing in Chapter 2, streams are not necessarily streams of events, and indeed some of the roots of stream programming come from signal processing. When we are using it for event processing, the data flowing in the system represent event objects, thus has the appropriate event semantics. The input/output model of the stream dataflow graph is publish/subscribe[3]

Event instances are represented as records, and are often referred to as *tuples* following the relational model's terminology. A stream is a continuous flow of events, in most cases all of the same event type, considered to be tuples of the same relation. The stream may be unbounded and be active forever. This means that, unlike the conventional relational model where a query is executed against an entire table of data, in the continuous query model a query can only execute against a bounded subset of the stream. The stream is broken up into a sequence of *windows* and the query is performed successively against each window. Windows in stream processing correspond to the temporal context concept that we defined in Chapter 7 (and for this very reason we sometimes refer to temporal context partitions as windows).

Figure 10.1 shows a data flow graph, in which streams are in the edges and operations on streams are performed by the nodes. This data flow graph is taken from the SPADE language[4].

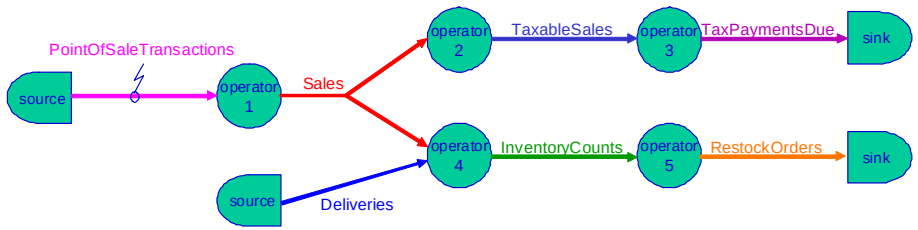
---

[1] The Apama language is a good example of an imperative style language and we showed an example of it in Chapter 8.

[2] The full tutorial made by the EPTS language analysis group in ACM DEBS 2009 is available in: http://www.slideshare.net/opher.etzion/debs2009-event-processing-languages-tutorial

[3] See Chapter 2 for discussion of publish/subscribe.

[4] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, Myungcheol Doo: SPADE: the system s declarative stream processing engine. SIGMOD Conference 2008: 1123-1134. http://portal.acm.org/citation.cfm?doid=1376616.1376729




```
stream PointOfSaleTransactions(…) := Source()[…]{…}
stream Sales(…) := operator1(PointOfSaleTransactions)[…]{…}
stream TaxableSales(…) := operator2(Sales)[…]{…}
stream TaxPaymentsDue(…) := operator3(TaxableSales)[…]{…}
Null := Sink(TaxPaymentsDue)[…]{…}
stream Deliveries(…) := Source()[…]{…}
stream InventoryCounts(…) := operator4(Sales;Deliveries)[…]{…}
stream RestockOrders(…) := operator5(InventoryCounts)[…]{…}
Null := Sink(RestockOrders)[…]{…}
```

Figure 10.1 An example of a data flow graph with streams on the edges, and operators on the nodes.

There is another way of representing the graph, shown in Figure 10.2 which is taken from the Aleri language. In this representation the streams are shown as nodes, and the edges describe the flow. One of the stream nodes, an aggregation node, is detailed in the box below, showing the filter's derivation.

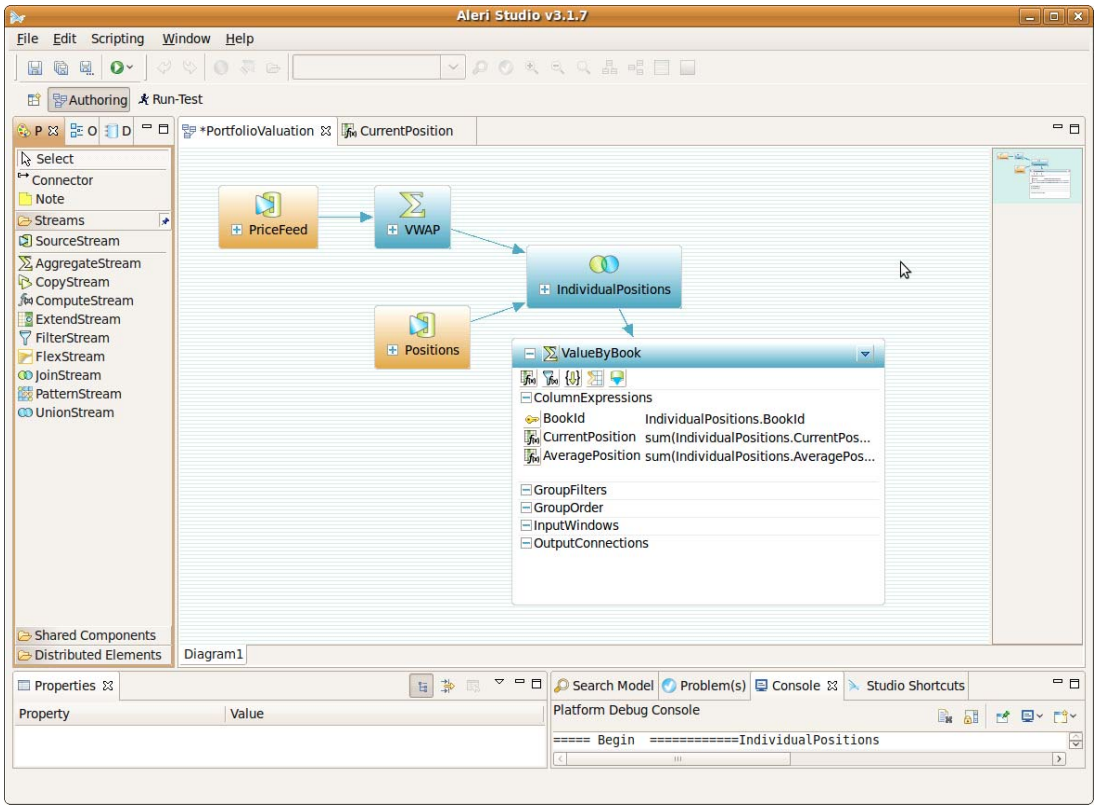


Figure 10.2 An example of data flow with streams on the nodes taken from the Aleri language.

You can see from this that there are several ways to model stream processing. We now show some examples of stream processing code. Note that these are just samples, we refer anyone who wants to learn details of a particular language to the fuller examples and references given on the book's website.

Here is an example of a query in the CQL language (developed in the Stanford Stream project)

```
Select Sum(O.cost)
From Orders O, Fulfillments F                #1
[Range 1 Day]                                #2
Where F.clerk = “Sue”
   And O.customer = “Joe”
   And O.orderID = F.orderID                 #3
```

Cueballs in code and text

#1 Input streams
#2 Window.
#3 Match condition

This query composes the two input streams shown at #1, applying a temporal context #2 to the second input stream and using the match condition #3. This query adds up the total cost of orders placed each day by customer "Joe" that were fulfilled by clerk "Sue".

Our next example is a more complex CQL query; this one includes a segmentation context and sampling. Sampling is a form of filtering used in stream computing as the amount of events in each window may be very high.

```
Select F.clerk, Max(O.cost)
From Orders O, Fulfillments F
[Partition By clerk Rows 5]        #1
10% Sample                         #2
Where O.orderID = F.orderID
Group By F.clerk                   #3
```

Cueballs in code and text

#1 Window
#2 Sampling directive
#3 Segmentation context

This query takes a 10% sample of the `Fulfillments` stream, and extracts the five most recent fulfillments for each clerk (this is specified by the window specification #1 and the sampling directive #2). As in regular SQL, the Group By #3 means that the query then computes the maximum order cost for each of these groups of five fulfillments. The combination of Partition By and Group By is equivalent to a segmentation context in our model.

Here's an example of a different language, also SQL-based. This one uses CCL, the language used by Coral8 now part of Aleri.

```
CREATE STREAM Vwap_s
  SCHEMA (Symbol STRING, Vwap FLOAT);              #1
INSERT INTO Vwap_s
  SELECT Symbol, sum(Qty * Price)/sum(Qty)         #2
FROM Trades_s
KEEP 30 MINUTES                              #3
GROUP BY Symbol;                                   #4
```

Cueballs in code and text

#1 Output stream and its schema

#2 Derivation rules
#3 Window definition
#4 Segmentation context

This query takes an input stream of Trade events and produces an output stream of derived events #1 containing volume weighted average prices. The calculation is specified by the derivation rules #2, which follow standard SQL syntax (as was the case with the CQL examples). There is a 30 minute time window #3, and the GROUP BY clause #4 establishes a segmentation context meaning that VWAP calculations are performed independently for every different stock symbol encountered in the input stream.

We end with an example of an operator written in a stream processing language which does not look like SQL.

Listing 10.1 An example of an operator written in a stream-processing language

```
stream VWAPAggregator@day                                    #1
(ticker:String, svwap:Float, svolume:Float)

:= Aggregate
(TradeFilter@day <count(15), count(1), pergroup>)            #2

[ticker]                                                     #3

{ Any(ticker),                                               #4
Sum(myvwap),                                        #4
Sum(volume) }                                                #4

 partitionFor(TradeQuote@day),                               #5
 ComputingPool[mod(@day-1,NCNT)]                             #5
```

Cueballs in code and text

#1 Output stream
#2 Window definition
#3 Segmentation context
#4 Derivation rules
#5 Partitioning directive

This is a Spade aggregate operator, which takes a set of VWAP values as input and them adds them up. As with the CCL example it starts with a definition of the output stream (in this example it is called `VWAPAggregator@day`) and its schema `#1`. The window definition `#2` means that the operator takes input from the `TradeFilter@day` stream, and calculates its aggregate every time that a trade occurs, using the last 15 trades (this is similar to our sliding event temporal context). The operator also includes a segmentation context `#3` that means that the calculation is performed separately (and in parallel) for every distinct value

of the incoming events' ticker attribute. The last lines #5 tell the system where to locate this processing for optimal performance in a multi-processor system.

To summarize, stream oriented languages are one of the common event processing language styles. While several of these languages extend SQL, different languages extend it in different ways[5]. We now look at another style: rule-oriented languages.

10.1.2 Rule oriented languages

The other dominant style of event processing languages is the style we call *rule-oriented*. The *rules* word is overloaded, as there are several distinct types of rules: production rules, active (Event-Condition-Action) rules and logic programming based rules. We briefly survey each of these styles.

PRODUCTION RULES

Production rules are rules of the type: if condition then action. They operate in a forward chaining way, when the condition is satisfied the action is performed. Production rules are rooted in expert systems; the operational processing of production rules may be either declarative or procedural:

- Declarative production rule execution is typically based upon some variation of the RETE[6] algorithm which matches facts against the patterns contained in the rules to determine which rule conditions are satisfied. Information about the antecedents (conditions) of each rule is stored in an internal state, and in every execution cycle changes to these states are evaluated.
- Procedural production rule execution is based on sequential execution of compiled rules.
- Production rules are based on state changes and not on events; however, there are some event processing languages that extended RETE based production rules to support event processing. This is done by making events an explicit part of the model, so that event occurrences can be used as part of the conditions for invoking an inference rule. Thus the event processing is done through an inference process.

Figure 10.3 shows the OMG Production Rule Representation classes.

-

---

[5] This article presented by owners of different languages discuss the semantic differences among stream SQL extensions: Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, Stanley B. Zdonik: Towards a streaming SQL standard. PVLDB 1(2): 1379-1390 (2008). http://www.vldb.org/pvldb/1/1454179.pdf

[6] The RETE algorithm has been introduced in:
Charles Forgy: Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. Artif. Intell. 19(1): 17-37 (1982)

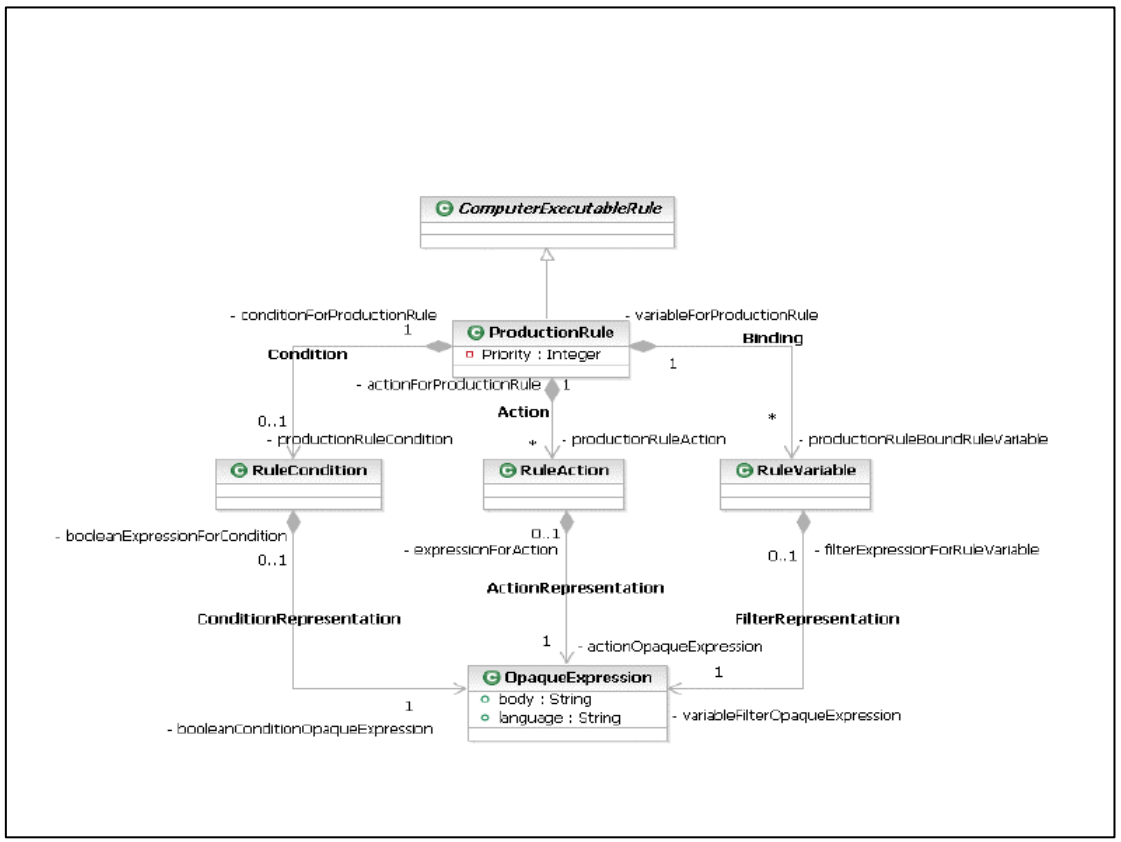


Figure 10.3 OMG Production Rules Representation

This figure comes from part of an OMG standard for modeling production rules in UML. As noted, events are modeled as part of the rule conditions.

ACTIVE RULES

Active rules, also known as Event-Condition-Action (ECA) rules, are descended from work on active databases[7]. Active rules operate according to the following execution pattern:

When event occurs, evaluate conditions and if satisfied, trigger an action.

---

[7] Norman Paton, Active Rules in Database Systems. Springer, 1998, http://www.amazon.com/Database-Systems-Monographs-Computer-Science/dp/0387985298/ref=sr_1_11?ie=UTF8&s=books&qid=1259266096&sr=8-11

The event may be primitive or composite. Each active rule can be mapped to an EPA in our model, since it has input events that trigger it, has some filtering conditions, and executes some action that may derive additional events. The Rulecore example in chapter 9 is an example of specific active rule language, and in listing 10.2 we show the general structure for active rules.

Listing 10.2 General structure for active rules

```
<Rule style="active" eval="strong">
      <on>
                    <!-- event -->
      </on>

      <if>
                    <!-- condition -->
      </if>

      <do>
                   <!--  action -->
      </do>

     <ifPost>
                  <!-- postcondition -->
     </ifPost>

     <doAlternative>
                   <!-- alternative/else action -->
     </doAlternative>
</Rule>
```

This is a general structure for active rules; particular rule languages are variations of this structure.

The third kind of the event processing rule language is the logic programming rule style.

LOGIC PROGRAMMING BASED RULES

Logic programming is a programming style based on logical assertions, the most well-known example of a logic programming language being Prolog. The application of the logic programming style to event processing stems from work done in the deductive database area[8].

Listing 10.3 shows an example of logic programming based event processing, taken from the ETALIS implementation of the Fast Flowers Delivery application. More information about the exact syntax and semantics of the language exists on the website.

---

[8] A good source of knowledge about deductive databases is: Stefano Ceri, Georg Gottlob, Letizia Tanca: Logic Programming and Databases. Springer-Verlag, 1990. http://www.amazon.com/Programming-Databases-Surveys-Computer-Science/dp/0387517286/ref=sr_1_7?ie=UTF8&s=books&qid=1259274738&sr=8-7

Listing 10.3 ETALIS logic programming based example

```
no_bid_alert(DeliveryRequestId-:(                                    #1
     start_automaticAssignment
(DeliveryRequestId,
            StoreId,
            ToCoordinates,
            DeliveryTime) fnot
delivery_bid
(DeliveryRequestId,
_DriverId,
_CurrentCoordinates,
_PossiblePickupTime.(
no_bid_alert(DeliveryRequestId-:(                                    #2
     start_manualAssignment
(DeliveryRequestId,
     StoreId,
     ToCoordinates,
     DeliveryTime) fnot
      delivery_bid
(DeliveryRequestId,
_DriverId,
_CurrentCoordinates,
_PossiblePickupTime.(
print_trigger(no_bid_alert/1.(
```

This assertion is intended to derive the derived event `no_bid_alert`. It identifies two cases the automatic assignment case #1, and the manual assignment case#2.

10.1.3 Development environments

There are two types of development environment, text based and graphical based environments. These two are not mutually exclusive, and in some cases some of the development functions done graphically, and some are text oriented. Text can be either full text mode, or "fill a form" type of text. The various environments reflect different assumptions about the developers' preferences. In some cases developers prefer a more familiar text-based interface, while there are also those who prefer more visual style of development. Figure 10.4 shows an example of a text based development environment taken from Apama's Eclipsed based Integrated Development Environment, which is called Apama Studio.

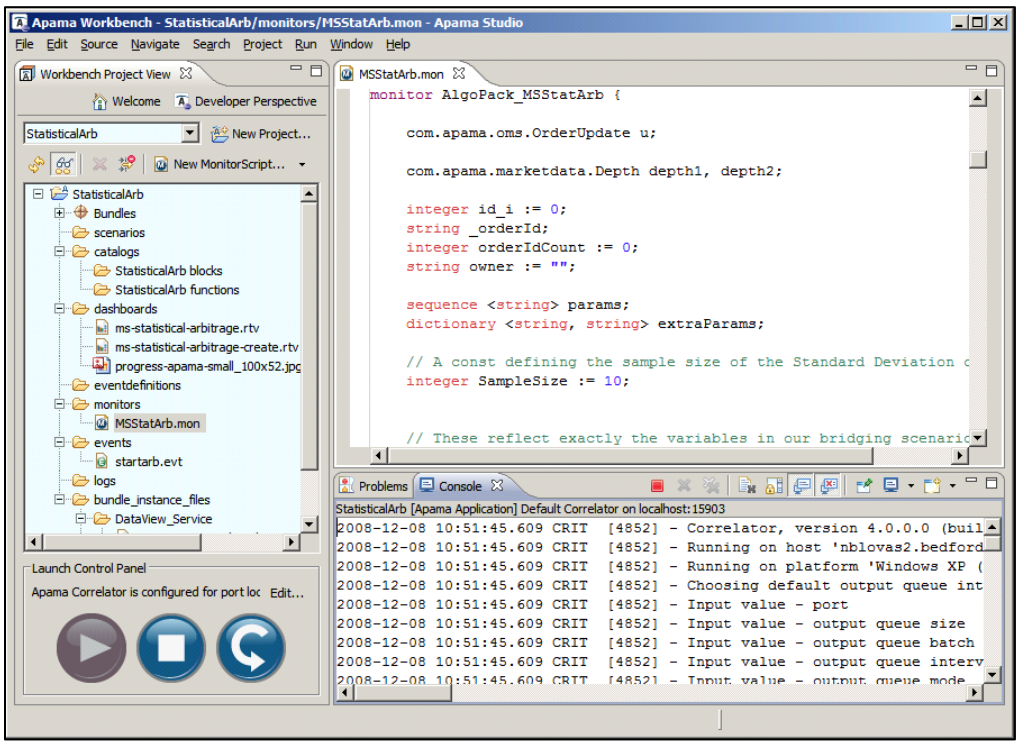


Figure 10.4 Text based development environment (Apama Studio)

Streambase also has an Eclipse-based IDE, but as you can see from Figure 10.5, this has a graphical based development environment, with some functions being provided in a textual manner. In this tool (Streambase Studio) the EPN is constructed graphically, while event types and individual functions are then built using form-oriented text.

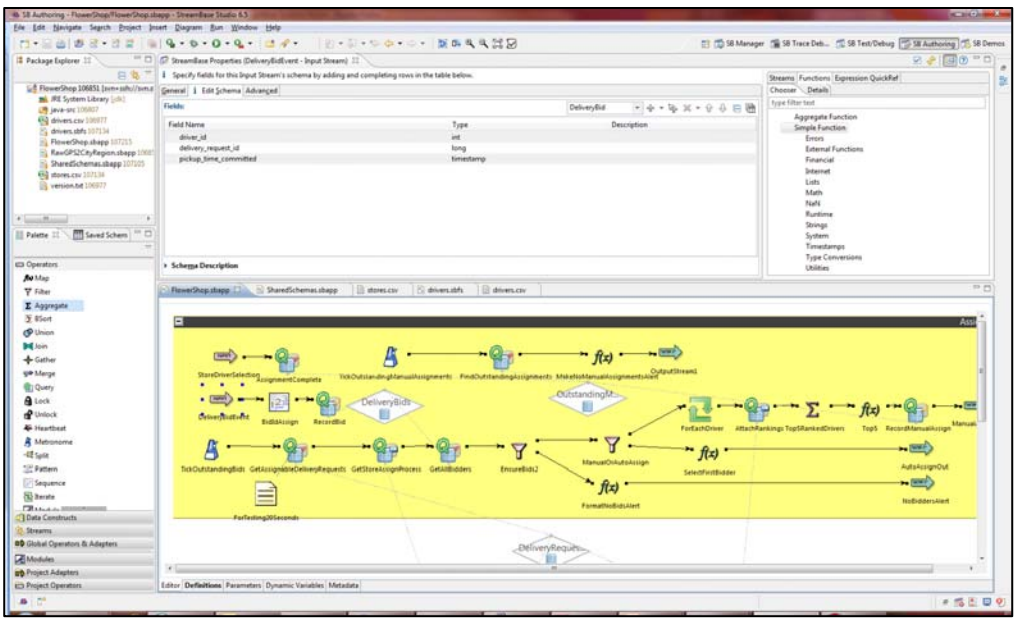


Figure 10.5 Combined graphical and form based development environment

The environments we have looked at here are geared mostly to technical developers. In Chapter 12 we discuss the trend towards having semi-technical event processing developers.

The language and development environment is just one facet of the event processing implementation, next we discuss the non-functional properties of event processing systems.

10.2 Non-functional properties

An important aspect of the engineering and implementation considerations in any system is the non-functional aspect. Non-functional requirements are concerned not with WHAT a system does but HOW WELL it does it. It is often the non functional properties that make or break a specific application. In this section we briefly survey the main non-functional aspects of event processing systems and explain the particular requirements imposed by event processing systems that the system designer should be aware of. Not all of these requirements apply equally to all application, so when designing an event processing application one needs to consider which of them are important for the case in hand. In the next section we deal with various optimizations and relate them back to these requirements.

The non-functional requirements that we discuss in this section are scalability, availability and security. There are some further non-functional properties such as reliability and

usability. We will touch on reliability in Chapter 11 when discussing inexact event processing, and usability requirements are closely related to the programming styles and development environments discussed earlier in this chapter.

10.2.1 Scalability

We start with a definition of what we mean by scalability:

Definition

Scalability is the capability of a system to adapt readily to a greater or lesser intensity of use, volume, or demand while still meeting its business objectives.

There are several dimensions of scalability, the dimensions relevant to us here are: the volume of events, the number of agents, producers, consumers and contexts, the complexity of computation, and the processor environment.

SCALABILITY IN THE VOLUME OF PROCESSED EVENTS

High event throughput is considered as one of the characteristics and main motivations for the use of event processing software. This is certainly true in some application segments, however experience shows that the primary reason for employing event processing software is its effectiveness: to increase agility and reduce total costs of ownership. So the range of applications that are likely to employ generic event processing software is much wider than just those requiring high event throughput.

The scalability requirement in event processing systems is the ability to handle variable event loads efficiently; the quantity of events may go up and down over time. Extremely high volumes of input events may require some special treatment and optimization, examples of systems that require high event throughput are: some financial market applications, weather-related event processing, and telephony call tracking. There are some systems that have been specifically designed with high event throughput in mind, and in section 10.3 we discuss performance measures and optimizations.

SCALABILITY IN THE QUANTITY OF AGENTS

In some applications of event processing the major scalability issue is the ability of the EPN to grow substantially and have a very large number of EPAs. An example is a banking system that enables each customer to create his or her own sophisticated alerts. Each customer could end up with a unique EPA and this could result in the dynamic creation of a very large and complex EPN. When designing an event processing system, estimates about the number of EPAs and their growth curve may impact the way the system is implemented and deployed. Some related optimizations are discussed in section 10.3.

SCALABILITY IN THE QUANTITY OF PRODUCERS

In some cases the number of event producers can grow substantially. Consider a Web book store that tracks events related to all the customers that browse and buy books so as to determine patterns of use. If we view every customer as a separate event producer then the number of event producers can grow very large. Even though the number of events from each customer may be small, the total number of customers can be high and the system should account of it. This can also lead to a high number of context partitions, which we discuss later.

SCALABILITY IN THE QUANTITY OF CONSUMERS

In some cases the amount of event consumers may become very high, this may happen in popular subscription systems. In some cases an event emitted from the event processing system may be routed to many consumers. This may require optimizations in the routing level such as the use of multicasting[9], some related optimizations are discussed in section 10.3.

SCALABILITY IN THE QUANTITY OF CONTEXT PARTITIONS

The number of context partitions that are concurrently active may grow very fast in some event processing applications. Consider an Internet retail store, which has an open context for each order from the time it opened until delivered. The number of such orders may be very large, and if we assume that each context partition has an internal state this requires the event processing system to store a large amount of state information. If each context partition is implemented by a distinct run-time artifact, this also leads to a scale-up in number of agents.

SCALABILITY IN CONTEXT STATE SIZE

Another context-related scalability issue is the ability of a single context partition to accumulate big states, especially if it is a long-running context partition. For example a trend pattern running over a 24 hour period might need to accumulate and retain a large quantity of events each day.

SCALABILITY IN THE COMPLEXITY OF COMPUTATION

The complexity of the EPAs themselves may have substantial impact on the overall performance of the system. Cases where the EPAs implement highly complex logic may

---

[9] Multicasting is the ability to transmit a single stream to multiple subscribers at the same time; for more information refer to http://www.tcpipguide.com/free/t_IPMulticasting.htm.

require different types of optimization than the other scalability aspects that we have mentioned. We return to this point in Section 10.3

SCALABILITY IN THE PROCESSOR ENVIRONMENT

Event processing systems may run in heterogeneous environment, on one extreme they may run on multi-processor supercomputers, on the other extreme small devices that have footprint limitations. Both ends, as well as those in the middle, require specific optimizations and an implementation that works well at point in this spectrum may need significant redesign to work well on a different size processor.

A system designer should be aware of all these scalability issues when designing an application, as well as the corresponding optimizations discussed in the next section.

10.2.2 Availability

Availability is one of the notable Quality of Service requirements in current systems, and we start by defining this term.

> **Definition**
>
> Availability is the percentage of the time in which a certain system is perceived as functioning by its users.

Event processing systems can use standard high availability practices like logging, failover and disaster recovery practices and we don't need any event-processing specific approaches here. The designer of event processing system has, however, to make some design decisions related to high availability. These considerations relate to whether it is cost-effective to employ high availability practices as they don't come for free, and may not be fully required in some applications. An example of such a consideration is the issue of recoverability, which we discuss now.

Some of the event processing agents that perform aggregation, composition and pattern detection are stateful. The internal state of such an agent has to be kept as long as the particular EPA instance is active, meaning as long as its context partition is valid. For example an EPA that calculates the `always` pattern over a period of 24 hours has to retain all the relevant events that occurred during that period. This brings us to the issue of recoverability.

> **Definition**
>
> Recoverability is the ability to restore the state to its exact value before a failure occurred.

There are well known techniques to achieve recoverability; the interested reader is referred to the additional reading list at the end of the chapter for sources. Achieving recoverability requires some overhead on the processing; changes in states need to be logged and the entire state needs to be written to persistent store, at least periodically[10]. This overhead may have a toll on the processing latency and total throughput of processed events.

In some applications recoverability is a must. If the event processing is part of a mission-critical application, and decisions are made using the results of this processing, losing some of the system's state may have critical implications such as: an order being ignored, missing a pattern in a specific customer's behavior, losing the location of a consignment of goods, taking a wrong decision due to not knowing about a new trend and more.

For other applications, it might not be cost-effective to apply recoverability. Consider a network management system that receives events about observable faults in the system and attempts to find the root cause. Since the events are symptoms of some underlying problem, they will occur again, unless the problem is resolved, so recoverability may help identify a problem faster, but it is not vital and might not be cost effective. Likewise, there are systems which look for statistical trends, these systems may be based on sampling or on analysis of large amount of events; in these cases recoverability may not be required at all.

As a conclusion, event processing systems should support recoverability as optional property with various tuning alternatives (for example full persistence of state, checkpointing) and the designers of each application should consider the cost-effectiveness of recoverability to their own application and decide whether recoverability is required.

From availability we move to discuss security in event processing systems.

10.2.3 Security

Security requirements relate both to ensuring that all operations are performed by authorized parties, and that privacy considerations are met. Specifically this means:

- Ensuring only authorized parties are allowed to be event producers of event consumers
- Ensuring that incoming events are filtered so that authorized producers cannot introduce invalid events, or events that they are not entitled to publish
- Ensuring that consumers only receive information to which they are entitled. In some cases a consumer might be entitled to see some of the attributes of an event but not others.
- Ensuring that unauthorized parties cannot add new EPAs to the system, or make modifications to the EPN itself (in systems where dynamic EPN modification is supported)
- Keeping auditable logs of events received and processed, or other activities performed by the system.

---

[10] Typically states are persisted in checkpoints, and logs are kept between checkpoints.

- Ensuring that all databases and data communications links used by the system are secure.

Some people[11] view security and privacy issues as barriers for the trust and thus utilization of event processing systems. Authorization issues are a concern since attacks that may send false events can be devastating to systems such air traffic control, medical devices, finance and the electricity grid. Privacy issues are also a big concern for people; there are people who won't install electronic vehicle toll payment devices, such as the EZPASS[12] system that exists in some of the USA states, since they are sensitive to their privacy and don't wish to have anyone recording information about their movements. Privacy is also a concern in healthcare applications, and in many jurisdictions there is legislation requiring organizations to safeguard the privacy of personal data in all application domains. Trust is particularly significant in applications where sensitive data is passed between different organizations.

Event processing systems may have various levels of sensitivity to security and privacy issues. If the collection of event producers is a closed set in which security practices are trusted then the problem is reduced, on the other hand if anybody can be a producer (for example when events are based on Twitter feeds) then the security issues may be pervasive.

There have been some studies on related security issues[13], but specific event processing security and privacy issues are mainly dealt in ad-hoc way. In the additional reading section at the end of this chapter, the reader is referred to additional reading on database security, which has many common issues.

In conclusion, designers of an event processing application should be aware of their non-functional requirements and make the choices appropriate to their own particular applications. We now discuss optimization techniques to address some of these non-functional requirements.

10.3 Performance objectives

Some of non-functional requirements can be translated to performance objectives which can then be the subject of various optimization approaches. In this section we discuss some of the major performance objectives for event processing relating to throughput, latency and time-constraint objectives. We summarize these objectives in Table 10.1 and will discuss each later in this section.

Table 10.1 Performance objectives and their associated metrics

---

[11] The claim about security and privacy as barriers is taken from Chandy and Schulte's book that is cited in the additional reading section at the end of the book.

[12] http://www.ezpass.com/

[13] An early article about security issues in pub/sub system is: Mudhakar Srivatsa, Ling Liu: Securing publish-subscribe overlay services with EventGuard. ACM Conference on Computer and Communications Security 2005: 289-298 http://portal.acm.org/citation.cfm?doid=1102120.1102158

Objective Number	Objective Name	Objective metrics
1	MAX input throughput	Maximize the quantity of input events processed by a certain system or sub-system within a given time period
2	MAX output throughput	Maximize the quantity of derived events produced by a certain system or sub-system within a given time period
3	MIN average latency	Minimize the average time it takes to process an event and all its consequences in a certain system or sub-system
4	Min maximal latency	Minimize the maximal time it takes to process an event and all its consequences in a certain system or sub-system
5	Latency leveling	Minimize the variance of processing times for a single event or a collection of events in a certain system or sub-system
6	Real-time constraints	Minimize the deviation in latency, from a given value, for the processing of an event and all its consequences in a certain system or sub-system.

All of these objectives are intended to address scaling issues, but each of them addresses it using different assumptions and may be served by different optimizations. As you can see from table 10.1, each of the objectives may apply to an entire system, or to any part of it. In some systems there is a single performance objective for all the processing in the system, for example: latency leveling for each event type in that system. In other systems there may be mix of performance objectives, some of the events may have real-time constraints associated with them while others may have another metric. Performance objectives may also be composed out of several separate metrics. We now briefly discuss each of the six performance objectives defined in Table 10.1, followed by a discussion of metric composition.

MAX INPUT THROUGHPUT

This is the performance metric most mentioned as a motivation for high performance stream processing systems[14]. This metric is strongly related to the requirement for scalability in the quantity of events. This metric measures the number of input events that the system can accept within a given timeframe while still functioning correctly. It is sometimes referred to as Events Per Second (EPS). Note that while this metric asserts that the system can absorb

[14] An example of an article showing an optimization related to this performance metric is: Joel L. Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, Lisa Fleischer: SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. Middleware 2008: 306-325 http://www.springerlink.com/content/9h772844u5875757/

events, it does not say anything about the latency of processing. To specify a required latency, this metric has to be composed with a latency-oriented metric.

MAX OUTPUT THROUGHPUT

This is a performance metric that refers to the output throughput rather than the input throughput. It is also measured in events per second, but in this case the measure relates to the derived events that the system generates and not to the input events.

MIN AVERAGE LATENCY

This is the first of the latency metrics. It is a statistical metric that refers to the average latency of all events, and it is measured as a time unit (for example 10 milliseconds). Since different event types may have different processing complexity, it's sometimes useful to measure the latency of a single event type, rather than the overall metric, which is the average of all the average event type latencies.

MIN MAXIMAL LATENCY

This metric relates to the maximal latency for a certain event type or collection of event types. Note that this is a different objective than the previous one, and there are optimizations that improve one of these metrics, and make the other one worse.

LATENCY LEVELING

This metric is also known as deterministic performance metric, and is sometimes identified with real-time processing. The motivation of this metric is to have predictable and low variance performance processing for each event type or collection of event types.

REAL-TIME CONSTRAINTS

While latency leveling is identified with real-time systems, these systems may also need to impose particular performance upper limits for either processing of a certain event type, or a certain EPA. This can be achieved through a real-time constraints metric which specifies just such an objective. Note that real-time constraints may be hard real time, in which compliance with these constraints is a must, since lack of compliance may have disastrous consequences, or soft real-time constraints that are considered as a Quality of Service measurements.

COMPOSING METRICS

In some cases there is a need to form a performance objective that includes more than one metric. This composition may be related to the same part of the system, for example there might be an event type which has both throughput and latency related metrics. Alternatively, there could be different performance metrics for different parts of the system. An

optimization plan might have to take into account different objectives, with some weighting among them, creating an objective function.

This takes us now to look at the various types of optimization available to help meet such objective functions.

10.4 Optimization types

In this section we discuss various types of optimization that have either been used or have been proposed for use with event processing systems. These can serve as building blocks for an optimization plan that is particular to a specific performance function. We discuss optimizations in the following areas:

- Optimizations related to EPA assignment: partition, parallelism, distribution and load shedding.
- Optimizations related to the coding of specific EPAs: code optimization, state management.
- Optimization related to the execution process: scheduling, and routing optimizations.

It should be noted that the optimization considerations are quite complex, and this area is still in need of more established methods and practices. The purpose of this section is to make applications designers aware of optimization opportunities, rather than to provide a recipe to optimize a specific application.

10.4.1 EPA assignment optimizations:

In chapter 6 we stated that EPA represents a logical function, and that there are various ways to map the logical functions to physical run-time artifacts. This is the basis for the EPA assignment optimizations, as the choice of assignment can influence the performance metrics that we listed earlier.

These optimizations are also known as "black box optimizations", since the EPA's implementation is assumed to be fixed. They deal with factors external to the EPA such as its location and relative scheduling. We survey now the most common assignment optimizations:

PARTITIONING OF EPA INSTANCES TO RUN TIME ARTIFACTS

The decision of how EPA instances should be mapped to run-time artifacts can have a major effect on the various performance metrics. We refer to this as *partitioning* the EPAs, and the idea is to group EPA instances so that they execute together for better performance. The two extremes are a single centralized run-time artifact that embeds all the EPA instances, and a separate run-time artifact for each EPA instance. The centralized solution has benefits for cases where the volume of events is not an important measure, since it saves the overhead of communication between the different EPAs. Partitioning is the key both to parallel

execution and to distributed execution, both of which we will discuss later. It is an enabler for many of the scalability dimensions, since independence of sub-systems may lead to better scalability.

Partitioning decisions can be driven by the EPN topology as this determines the dependencies among EPAs, although as we saw in listing 10.4 some languages, such as SPADE, let the programmer make partition decisions. One approach to partitioning is based on assigning EPAs to strata, where the EPAs in each stratum are independent of one another and can run in parallel. If EPA1 produces events that are consumed by EPA2, then EPA2 is placed in a higher stratum. We show an example of stratification in figure 10.6, where the EPN is partitioned to three strata, each of which contain independent EPAs. Note that this is a very simple example, and for EPNs in which there are many interdependencies among EPAs, the stratification process is more complex. We reference an article describing stratification based optimization at the end of this chapter.

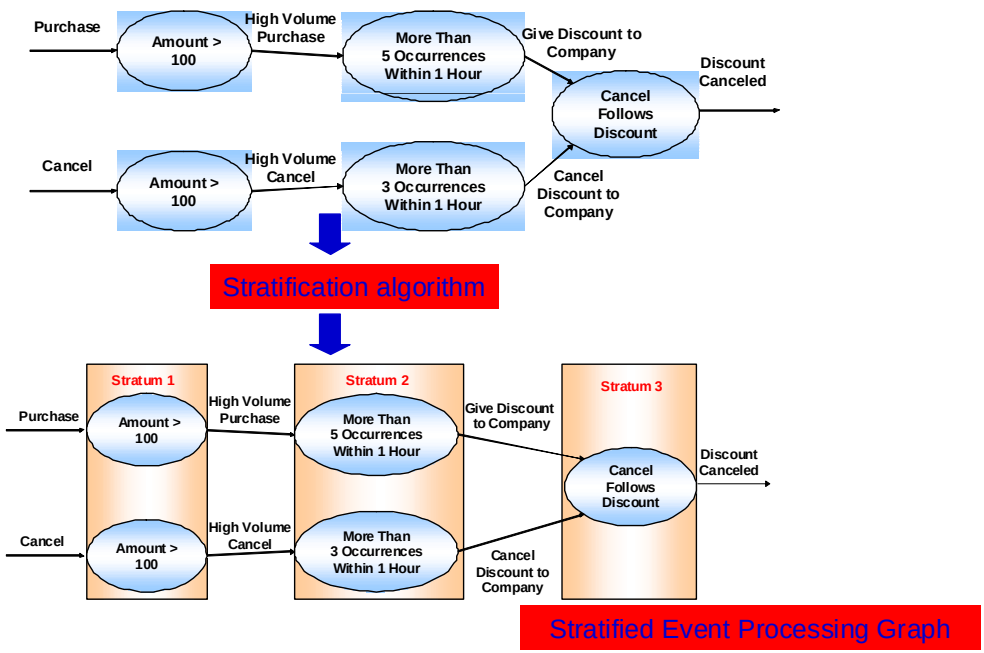


Figure 10.6 Stratification of an EPN to three strata.

This stratification process tells us which EPAs can run in parallel, but to decide which of them should be grouped together in the same runtime artifact we have to consider more things

such as the number of available core/processors, the level of distribution, the communication overhead, and of course the performance objective function. We discuss some of these aspects next.

PARALLEL PROCESSING

One of the major ways to achieve various performance metrics is parallel processing. There are three levels of parallelism: parallelism inside a single core using multi-threading, and parallelism by partitioning the work within a multi-core machine where the threads have access to shared memory, and partitioning the work to multiple machines within a cluster. Decisions about what activities should be run in parallel are difficult, and are usually taken automatically by a system optimizer, rather than being performed manually. There has been some research work performed into such parallel processing[15] .

DISTRIBUTED PROCESSING

Another optimization is to distribute the processing and makes the processing close to the producers and consumers where applicable. For example, if there are multiple sensors within the same location, and the processing starts with the aggregation of events that are emitted by the sensors, then placing the aggregation EPA close to the sensors can eliminate a substantial amount of network traffic. Likewise if at the leaf of the EPN, there is an EPA that creates many events that are all consumed by a certain consumer, or a set of consumers that are located in a certain location, it might be useful to locate this EPA close to the consumer. This optimization approach can also complement the parallel processing approach. If instead of a physical cluster or multi-core machines that are co-located, the parallel processing is executed over a grid of machines within various geographic locations, the assignment optimization may assist in the decision of which agents should be co-located, since they have substantial amount of communication among them.

LOAD SHEDDING

Static optimization techniques, such as stratification, involve analysis of the EPN dependencies, making some assumptions about the traffic load and available resources. However these assumptions, as well as the topology of the EPN, may change in time. The introduction of more resources, the temporary unavailability of computing resources, as well as unexpected changes in the distribution and load of events, are all reasons for re-evaluation of the partitioning scheme, since these changes may change the assumption they are based upon. Several works about load shedding in the context of data stream processing

---

[15] An example of such optimization for stream processing is in the referenced article: Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel L. Wolf, Kun-Lung Wu, Henrique Andrade, Bugra Gedik: COLA: Optimizing Stream Processing Applications via Graph Partitioning. Middleware 2009: 308-327 http://www.springerlink.com/content/aw817m13m4536001/

have been published in recent years[16]. We list a book that provides survey of load shedding techniques in stream processing at the end of this chapter. The more general issue of load shedding in EPA assignment still requires more work, currently there are some ad-hoc solutions.

Some performance objectives require us to go further than the "Black Box" approaches we have discussed so far, and optimize the actual EPA code itself. We discuss a couple of such "white box" optimizations next.

10.4.2 EPA code optimizations

White box optimizations are optimizations that modify the internal execution of EPAs. This area is less developed then the black box optimizations. We briefly discuss some of the possibilities in this area. We start by looking at code generation and then move on to the more developed area of state management.

OPTIMIZED CODE GENERATION

Query optimization is a vital part of relational database execution; substantial research and development has been invested over the years in this area. The core idea behind query optimization is that while queries may look similar, different queries have different optimized execution plans, and thus an optimizer might generate totally different code.

The equivalent of this idea is also valid for event processing, and you might hope that if you have a language that is an extension of SQL extension then you could adjust the SQL query optimization to include continuous queries. However it turns out that these adjustments are not trivial.

To see why this might be not trivial, consider the EPA shown in figure 10.7 which is detecting a sequence pattern.

---

[16] For example, refer to the following article: Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik: Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. VLDB 2007:159-170 http://www.vldb.org/conf/2007/papers/research/p159-tatbul.pdf

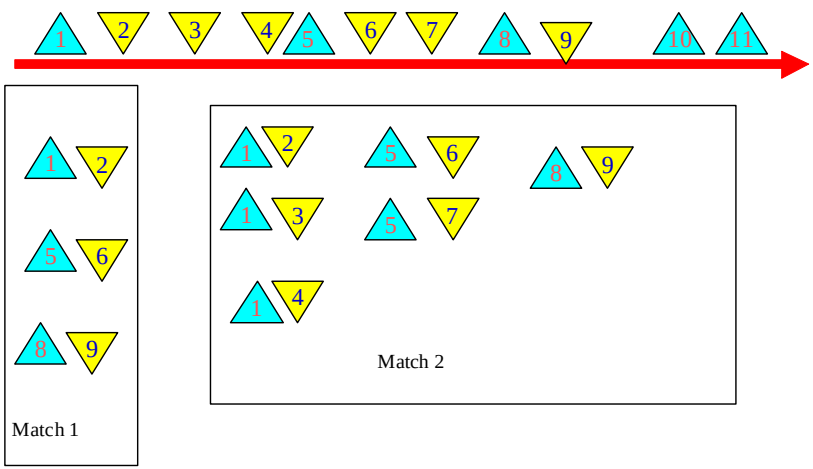


Figure 10.7 A sequence pattern matching example with different policies.

In this example, there are two types of event, the first type (E1) shown by a triangle with point upwards, has five instances: 1, 5, 8, 10, 11; while the second type (E2) has six instances: 2, 3, 4, 6, 7, 9. These instances arrive in the order shown at the top of the diagram, and the pattern is looking for the sequence of (E1, E2). As we saw in chapter 9, the output of a pattern EPA depends on the matching policies being used. Using a policy that matches each event in E1 to the next occurring E2 event gives us three matches: <1, 2>; <5, 6>; <8,9>. However a policy that matches against multiple subsequent E2 instances would creates six matching sets. It is conceivable that each of these interpretations has different optimal data structure, and different optimal code. A code optimization should account for this fact.

Another optimization that is being used by some event processing implementations is the use of Real-time Java[17], which allows for thread priorities and smoothes the memory management.

STATE MANAGEMENT

State management optimization relates to the way that internal state is held by EPAs, and in some cases also to global state elements. The basic trade-off is one between performance and recoverability. A memory-based state provides better performance, but recoverability requires some overhead and implementation complexity. A persistence-based state (for example one where state is held in a database) provides better recoverability, but may

---

[17] http://www.rtsj.org/

conflict with performance goals. The implementation of state management is a function of the requirements, both for performance and recoverability. Use of in-memory state can also be problematic when there is a need for scalability in the number of context partitions or quantity of events accumulated within a context partition.

Some possible optimizations in this area are:

- Using persistent states. This resolve space scalability issues and recoverability, however, it may harm performance goals.
- Using in-memory databases that provide caching capabilities, while guaranteeing recoverability. This is a way to balance between the two sides of the trade-off, with various tuning possibilities that relate to different assumptions about MTBF (mean time between failures) an MTTR (mean time to repair), in this case time for recovery.
- Using grid memory instead of persistence: The idea here is to replicate the state in memory held on multiple machines, so as to get recoverability without having to use disk-based persistence. This solution has an overhead of network traffic, and the complexity of synchronization among the different replicas.
- Using a mixture of approaches: Use persistent storage for states that have space scalability issues, and in-memory for others. You can also allow different levels of recoverability for different EPAs.

We complete our survey of optimization techniques with a discussion of execution optimization

10.4.3 Execution optimizations

There are some additional optimizations that can be performed at execution time.

SCHEDULING

Scheduling optimization deals with the planning of which EPA to run first in cases where there is no natural order of precedence, but the EPAs concerned compete with each other for computing resources. Scheduling optimization can be done when there are different performance requirements for different EPAs, for example where one EPA has real time constraints while the other does not. In such cases you might use a preemptive schedule that delays the execution of a run-time artifact that has already started in order to execute another run-time artifact that needs to run so as to comply with its real time constraints.

Scheduling can also be done by analysis of the EPN topology, giving priority to EPAs that are in a critical path to achieve some performance criteria[18].

ROUTING

There are various optimizations that relate to the transport layer which deal with the manner and physical implementation of routing between the various components of the systems (producers, EPAs and consumers). This relates to the way that event channels are implemented and the routing method used (anycast, broadcast, multicast and unicast). Routing optimizations are typically assumed to be the role of the transport infrastructure.

There is no comprehensive methodology yet for event processing optimization, so performance tuning of event processing applications is still an art rather than a precise science. This section provided a quick look at some of techniques that a system designer can use in order to optimize an application. Some of these optimizations are provided today, to some extent, by the event processing middleware that supports event processing applications. However this is still an active area of research and development and we expect that more optimization tools and methodologies will be provided in the future. This is also a good point to summarize this chapter.

10.5 Summary

In this chapter we have discussed some of the engineering aspects of event processing. The first aspect we looked at was software engineering, and here we reviewed various programming styles and development environments. We then talked about non-functional aspects of event processing, followed by a discussion of performance objectives and optimization techniques. The current engineering practices provide solid foundations for many existing applications. As the area of event processing is evolving, the engineering practices both in the software engineering aspects and in the optimization aspects. The next chapter discusses some challenges within the current state of the practice.

Additional reading

Klaus Schmidt: High Availability and Disaster Recovery: Concepts, Design, Implementation, Springer, 2006. http://www.amazon.com/High-Availability-Disaster-Recovery-Implementation/dp/3540244603/ref=sr_1_2?ie=UTF8&s=books&qid=1259392209&sr=8-2

---

[18] An example of an article showing an optimization related to scheduling is: Joel L. Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, Lisa Fleischer: SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. Middleware 2008: 306-325 http://www.springerlink.com/content/9h772844u5875757/

This book is recommended if you want to get deep understanding of high availability techniques.

Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman:
Concurrency Control and Recovery in Database Systems, Addison Wesley, 1987.
The book can be freely downloaded from Phil Bernstein's homepage: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

This is a classic book and an excellent source for anybody who wants to understand the principles of recoverability, and the techniques needed to implement it.

Ron Ben Natan, Implementing Database Security and Auditing: Includes Examples for Oracle, SQL Server, DB2 UDB, Sybase, Digital Press, 2005.
http://www.amazon.com/Implementing-Database-Security-Auditing-Examples/dp/1555583342/ref=sr_1_2?ie=UTF8&s=books&qid=1259399581&sr=1-2

This book deals with the state of the practice in database security, and can give some insights about approach to security issues.

K M Chandy, W R Schulte: Event Processing: Designing IT Systems for Agile Companies McGraw-Hill Osborne Media; 1 edition (September 24, 2009)
http://www.amazon.com/Event-Processing-Designing-Systems-Companies/dp/0071633502/ref=sr_1_1?ie=UTF8&s=books&qid=1258816511&sr=8-1

This book, which we have mentioned before, is included here as it mentions security and privacy issues as barriers for the adoption of event processing (chapter 12: The future of event processing).

Jane W. S. Liu: Real-Time Systems, Prentice Hall, 2000
http://www.amazon.com/Real-Time-Systems-Jane-W-Liu/dp/0130996513/ref=sr_1_5?ie=UTF8&s=books&qid=1259439073&sr=1-5

This book explains the notion of basic concepts of real-time systems.

Geetika T. Lakshmanan, Yuri G. Rabinovich, Opher Etzion: A stratified approach for supporting high throughput event processing applications. DEBS 2009
http://portal.acm.org/citation.cfm?doid=1619258.1619265

This article described EPA partition using the stratification approach.

Sharma Chakravarthy, Qingchun Jiang: Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing, Springer 2009.

http://www.amazon.com/Stream-Data-Processing-Perspective-Scheduling/dp/0387710027/ref=sr_1_1?ie=UTF8&s=books&qid=1259477906&sr=1-1

This book provides an introduction to stream processing, and discusses several load shedding and scheduling optimizations.

Exercises

10.1. How do Continuous Queries, and Rules relate to the concept of an EPA?

10.2 In the EPN model we present in this book, we associate processing functions with the EPN nodes. Some stream processing models associate functions with edges. Can you describe an alternative EPN representation where the functions are associated with edges? Can the functionality be spread between nodes and edges?

10.3 What are the pros and cons of graphical versus text oriented development environments?

10.4 State the non functional requirements, performance metrics and optimizations for the Fast Flowers Delivery application used in this book.

10.5 Devise guidelines for using the various performance metrics that we listed.

10.6 Which of the optimizations mentioned can be controlled by an application designer, and which depend on capabilities provided by event processing middleware?

10.7 Are the various event-processing programming styles and non-functional requirements related or totally orthogonal to each other? Provide some examples to justify your answer.

11

Focal points on major challenging topics

"Challenges are gifts that force us to search for a new center of gravity. Don't fight them. Just find a different way to stand."

- Oprah Winfrey

Up to this point we have focused on what might be called state-of-the-art event processing practice. Event processing applications are being developed successfully using the patterns and approaches that we have discussed, but the application developers should be aware that there are some challenging topics that are not fully resolved within the current-state-of-the-practice. These topics might not have any bearing on your particular application, but you should consider their implications to see if they raise any issues that you need to avoid.

The purpose of this chapter is to make the reader aware of these challenges. In this chapter we discuss three topics: temporal semantics of event processing, inexact event processing, and event processing causality and retraction. The thing that they all have in common is that current state-of-the-practice typically treats them in a simplified manner. We will take you through each topic, explaining the problem and outlining some possible solutions and their implications.

11.1. The temporal semantics of event processing

In part II of the book, we saw that time plays a major role in event processing. In Chapter 3, while discussing event types and structures, we noted that an event instance can have two temporal attributes: the `detection time`, which is the time that the event processing system detected that the event occurred, and the `occurrence time` which is the time, often provided by the event producer, at which it is thought that the event occurred in

reality. In Chapter 7, while discussing context dimensions, we saw that the temporal dimension is often the dominant context dimension, and also plays a role in most composite contexts. In Chapter 8 we saw that the temporal dimension had an important influence on stateful filtering and transformations and in our examination of event processing patterns in Chapter 9 we noticed that many patterns rely on the temporal order of events, and that furthermore these patterns are often used within a temporal context.

We can summarize this by saying that, in many cases, the outcome of a piece of event processing is affected by the timestamps associated with the input messages, and the order in which these messages are processed relative to one another. In this section we go deeper into the temporal dimension, and discuss three major issues:

- Occurrence time: time points versus intervals;
- The temporal properties of a derived event;
- Issues related to event ordering.

We'll start with occurrence time.

11.1.1 Occurrence time: Time point versus interval

When we defined the `occurrence time` attribute we defined it as a single point in time, the only ambiguity being the precision with which it is recorded, and we set a bound on this imprecision through the chronon (time granularity) attribute. This definition is in keeping with a view of events as being transitions between states of an external system; in models that handle transition between states, a transition is typically considered as instantaneous, that is something that occurs at a specific point in time, and so has "zero duration". Now is the time to review this assumption, and raise the question of whether events do indeed follow this definition and are instantaneous, or whether instead they really occur over a time interval. Here are three examples that demonstrate the ambiguity of `occurrence time`, and suggest that events can really have non-zero *event duration*.

1. The event `Flight BA0238 landed`. Landing is a process that starts with the descent of the aircraft, and ends when the aircraft parks at the gate[1]. This is clearly an interval. One could argue that "landing" is a state and not a transition, and so there are two point-in-time events, one when the aircraft starts landing, and one (the `landed` event) when it arrives at the gate. However the way that the `landed` event understood by people is still ambiguous, some people might understand it to mean the time when the first wheel makes contact, or all the wheels make contact, or when the aircraft leaves the runway.
2. A medical application uses a derived event called `call physician` produced by the pattern: `blood pressure is constantly raised during a 2 hour period and fever > 39° during this period`. When does this event occur? One interpretation could be that the event occurs during the entire two hour interval, while

[1] Some might say it doesn't end until the aircraft door is open.

another answer is that the `occurrence time` should be taken to be the time when the derived event is detected (which will typically be some time after the two hour interval has ended).

3. The event "the financial crisis of 2008/2009" is made up of many atomic events, and spans an interval that had not really ended when this book was written.

In reality many events take place over a period of time, contrary to the view of an event as an instantaneous state transition; such events really have an occurrence *time interval* with a start time and an end time. Why does this matter? The answer is that for computational purposes it is often easier to deal with events that occur at a single time point, and so many systems today assign a single timestamp to the event, rather than giving it separate start and end times. For example many event processing operations depend on knowing the order in which events occur, and it is much more obvious how to define an order if each event has a single `occurrence time`. This gap between reality and computational convenience is sometimes bridged by selecting a relatively coarse temporal granularity so that an event time interval can be approximated by a single time point.

This approach of approximating time-intervals to time-points is good enough for some applications, particularly when the event time-intervals are short relative to other timescales in the application, but in other cases it would be better to have an explicit representation of event time intervals. To do this, the following additions to the model are required:

1. Support for `time interval` as an explicit data type; a time interval is designated by the two time points that serve as its start and end point.

Definition

A *time interval is a data type that designates a continuous segment in time, starting at a time point (Ts) and ending at a time point (Te).*

A time point t is part of a time interval (TI) if: $Ts(TI) \leq t < Te(TI)$[2]

2. It may be useful to support another data type named *temporal element*, taken from temporal databases, as defined below.

Definition

A *temporal element is a non overlapping collection of time intervals.*

Temporal elements are useful when representing repeating intervals, for example working hours (every day that is a working day between 9:00am and 5:00pm).

---

[2] We define is as a "half-open interval" where the ending boundary is not included in the interval.

3. A temporal context, as defined in Chapter 7, establishes a set of (possibly overlapping) time intervals, called partitions. If an event instance is associated with a single time-point then it is clear whether it is to be included with a given context partition or not. When the event itself occurs within an interval and not a time-point, then the relationship becomes an interval-to-interval relationship and not a point-to-interval relationship. Figure 11.1 shows some of the basic relationships of an event interval to a context partition.
4. Since intervals are partially-ordered, then all patterns that are based on order (for example the `sequence` pattern, or the trend-oriented patterns) cannot be used as currently defined with events that have a time interval, and so their definitions have to be adjusted. Interval oriented patterns are outside the scope of this book, but we draw the interested reader's attention to the exercises at the end of this chapter.

Figure 11.1 Relationships between the context interval and the occurrence interval of an event

Figure 11.1 shows some of the possible relationships between the event's occurrence interval and a temporal context interval (for each of the first six there is also the converse relation,

for example context partition before event interval). When using a temporal context to process events with time intervals you need to say what relationship is to be used. For example a travel expenses application might define a context to be the year 2009, but with a relationship of `Finishes`, meaning that all business travel events that end in 2009, even if they started earlier, are to be included in this context[3].

11.1.2 Putting derived events in order

In an Event Processing Network we do not make a fundamental distinction between derived events (events generated by EPAs in the network) and raw events (events introduced by an event producer). An EPA may take either or both kinds of event as input and the way it processes an event depends on the event's type and content, not on whether it is raw or derived.

As we have seen, the `occurrence time` or `detection time` event attributes are important when determining whether an event instance falls within a certain context partition; in some systems these temporal event attributes are also used to determine the order of an event relative to other events, something that is important when using a pattern whose semantics is dependent upon that order. In this section we discuss the semantics of issues related to the ordering of derived events.

Recall that the order of events may be determined according to the time in which the event occurred in reality (`occurrence time`), or the order in which the event arrived at the system (`detection time`). The issue before us is how we should assign these timestamps to a derived event, and how we should order its processing time relative to other events, both raw and derived.

Let's look at some examples. In the Fast Flower Delivery example, things start when a store sends a `Delivery Request` event (a raw event) to the application. The application then creates the derived event `Bid Request` which is sent to the drivers. Let's assume that `Delivery Request 1` has occurred, and that a second Delivery Request occurs before the application has completed the calculation to issue the first `Bid Request`. The implementation now has a choice, it could either queue up the second `Delivery Request`, so that `Bid Request` 1 is issued before `Bid Request` 2, or it could suspend processing of `Delivery Request` 1 until after it has dealt with `Delivery Request` 2 (so the `Bid Requests` are issued in the opposite order), or it could use separate processing threads to handle both `Delivery Requests` in parallel. In this last case the two `Bid Requests` could be issued in either order, depending on processor loads or other conditions out of the direct control of the application. In this particular example, the relative order of the two `Bid Requests` is not that important, since they are independent and can themselves be dealt with in parallel, although, as we have seen, the drivers can get `Bid Requests` that are not

---

[3] The interval-to-interval relationships were introduced in Allen's seminal paper:
James F. Allen: Maintaining Knowledge about Temporal Intervals. Commun. ACM 26(11): 832-843 (1983)

in the same order as the original `Delivery Requests`. You might try and experiment yourself with this example using the various implementations on the book's website in order to see whether the order is being kept. To remind you the book's website is:

http://www.ep-ts.com/content/view/74/108/

While in the Fast Flower Delivery example the order may not matter, in other examples it may influence the result that is produced by an EPA further downstream in the EPN.

To see how this can happen, let's consider an event processing system that processes auctions. The auction's rules are that bidders can place bids within an auction interval, and at the end of this interval the highest bid wins. If there are multiple bidders that issued the highest bid, the first bidder to have issued this bid wins the auction. Let's assume that each bid starts as a raw event, but there is a validation process that involves checking the bidder's history and credit and an enrichment process which provides more information on the bidder. These two processes produce a derived event that inserts the bid into the auction system.

Now let's suppose that we take the most intuitive approach and assign the `detection time` of each derived event to be the point in time at which the derived event was emitted by the enrichment process, similar to the way that a raw event's `detection time` is the point in time at which it was entered into the system by an event producer. Doing this we may encounter two anomalies, both illustrated in figure 11.2:

1. Suppose that a bid event was issued on time, within the auction interval, but that the interval ends before the validation and enrichment has completed. An auction system using `detection time` would reject the bid, even though it was validly submitted and might actually have been the highest bid.
2. Suppose now that two equal highest bids are submitted one shortly after the other, both well within the deadline. They both get processed before the auction interval has ended but, as we saw earlier, the processing of the second bid might complete first, resulting in it getting assigned an earlier detection time. This means that it wins the auction, whereas according to the rules it should not have.

The conclusion from this discussion is that the `detection time` of derived events does not necessarily follow the semantics of `detection time` in raw events, and should itself be treated as a derived value. In our example, if we assign the `detection time` of the auction entry event to have the same value as the `detection time` of the corresponding bid event then these two anomalies disappear. There is a further issue to be tackled in the first anomaly case, as it will result in an "out-of-order" event condition; the event belongs to the context partition, but occurs after the context partition ends. We discuss out-of-order conditions in section 11.1.3. In this case it would have been better to use occurrence time ordering and not detection time ordering, but one of the practical problems is that some of the current event processing languages don't support occurrence time ordering.

Of course, this is not always the desired solution, in some cases we may explicitly want to impose the order based on the time in which the system derived the events. There are other reasons for allowing derivation in the detection time attribute, for example a transform EPA might want to set a `detection time` that is some time-offset from the `occurrence time` of one of its input events, or even make the `detection time` an interval instead of a time-point.

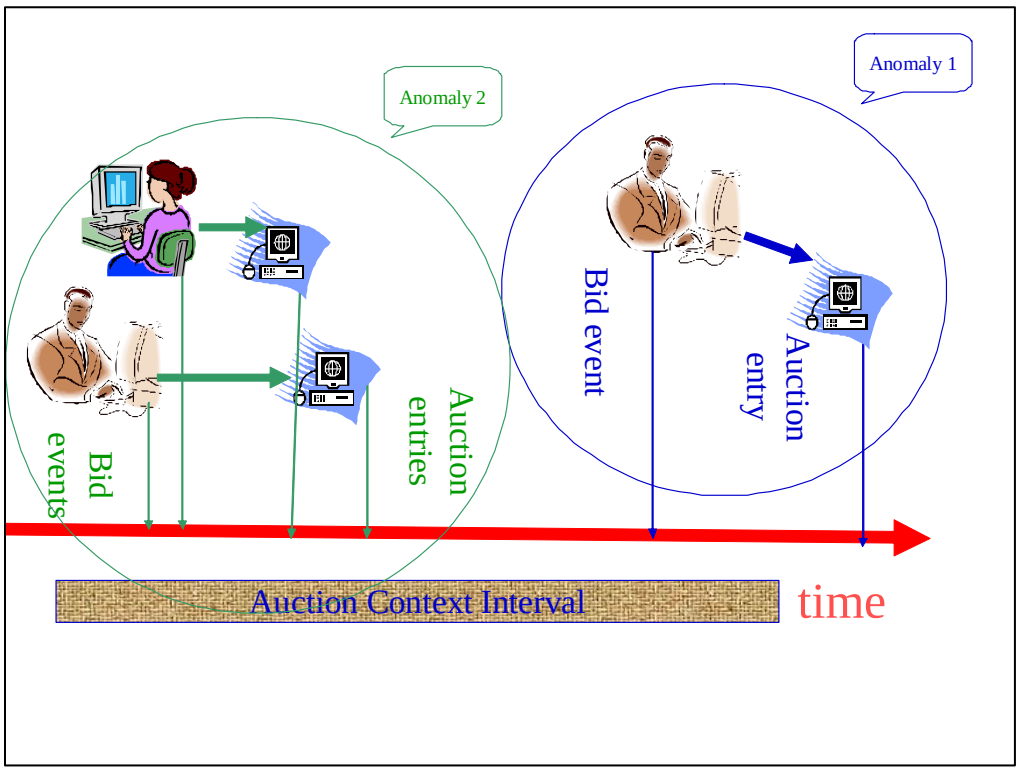


Figure 11.2 Two anomalies that may stem from using detection time for derived events in a naïve way.

We said that in some cases it is better to look at the `occurrence time` in order to determine the correct ordering, which brings us to the fundamental question of deciding when a derived event actually occurs. Let's look at a couple of pattern detection examples. Our first example is the sequence pattern example shown in Figure 9.7. In this example we are looking for patients who are discharged from hospital and then re-admitted, for the same reason, within 48 hours. We show such a case in table 11.1.

Table 11.1 An example of the use of the sequence pattern. This is to illustrate possible

alternatives for the semantics of occurrence time for derived events.

Event Identity	Event Type	Patient	Hospitalization Reason	Detection Time	Occurrence Time
E232243	Patient-Discharge	Pierre Werner	High fever	September 14, 2009 15:04	September 14, 2009 14:50
E291126	Patient-Admission	Pierre Warner	High Fever	September 16, 2009 08:20	September 16, 2009 08:18
E291244	Patient-Readmission (derived)	Pierre Warner	High Fever	September 16, 2009 08:21	?

In this example the `detection time` is the time at which the event arrived in the system after being created by the EPA, the question is what value should its `Occurrence Time` have? We can think of three possible values:

1. The `occurrence time` = `detection time` = September 16, 2009, 08:21.
   Rationale: since this event is a virtual one, and does not occur in reality, its occurrence time is identical to the time it is detected.
2. The `occurrence time` = `occurrence time` of the last event that completed the pattern, in this case, the `Patient-Admission` event, which we see had an `occurrence time` of September 16, 2009, 08:18.
   Rationale: the Patient-Admission event was the one which completed the pattern, thus it was the direct reason for the event derivation.
3. The `occurrence time` occurs over the interval of all events participated in the derivation = [September 14, 2009 14:50, September 16, 2009 08:18].
   Rationale: since all three events were responsible for the event derivation, then the derived event must have occurred within the interval bounded by the `occurrence times` of these three events.

We cannot say that one of the interpretations is more valid than the other two, so the choice as to which value to set for `occurrence time` should be left to the system designer as a policy decision, similar to the policies that we have seen before.

We will validate these three policy options by looking at another example. Let's take an absence pattern from the Fast Flower Delivery example, shown in figure 11.2.

Table 11.2 An example using the Absence pattern. We are using this to illustrate possible alternatives for the semantics of occurrence time for derived events.

Event Identity	Request Id	Detection Time	Occurrence Time
Bid Request 91	R3022	October 8, 2009 11:30	October 8, 2009 11:30
No Bidders	R3100 12	October 8, 2009 11:36	?

This pattern detection process is looking for cases where no drivers respond to a bid request. The pattern detection EPA generates a `No Bidders` derived event when the `Bid Interval` temporal context partition terminates and no event of type `Delivery Bid` has been detected. The bid interval terminates 5 minutes after it starts, which is at 11:35. We are showing the detection time of the derived event being set to the time when that event is created, at 11:36. The question again is: what value should be set as the `occurrence time` of the No Bidders derived event? Let's examine the three policies discussed in the previous example to determine whether they make sense:

1. The `occurrence time` = `detection time` = October 8, 2009, 11:36.
   The rationale here is similar to the previous example. `No Bidders` is a virtual event, as such it occurs when its derivation takes place.
2. The `occurrence time` = end of the interval time = October 8, 2009, 11:35.
   The rationale here is that the time at which we know that there aren't going to be any bidders is the time in which the context interval expired. This is similar to the case of the last event completing the pattern in the previous example.
3. The `occurrence time` is the interval of all events participated in the derivation = [October 8, 2009 11:30, October 8, 2009 11:35].
   The rationale here is that the `No Bidders` event relates to the entire interval, the event did not occur at any one of the time-points during this interval.

To conclude, putting derived events in order may not be trivial and the system designer should be aware of possible semantic anomalies here and take the appropriate policies carefully.

Some of the examples we have looked at were able to create events that need to be processed out-of-order. The next section deals with the issue of event ordering.

11.1.3 Event order and out-of-order semantics

This section deals with the issue of guaranteeing that events are processed in the correct order. There are many cases where the ordering of events is significant and it is important that they are processed in the correct order. Here are just three examples:

1. An event is part of a time-series event-stream, and we are looking for a trend pattern that is based on the order of events as occurred in reality (for example we might be checking to see if the value of a particular attribute is increasing). A disturbance to the order of the event stream will affect whether this pattern is detected or not.
2. A public library has a limited number of workstations available to library users. When all the workstations are occupied, the library institutes a time-out protocol terminating user's sessions so that they hand over the workstation to he person waiting next in line to use it. This is implemented by an event processing system that records events of the following types: work-station becoming free, work-station becoming occupied, person entering the queue, and person leaving the queue. The decision to start and stop the time-out protocol is determined upon the order of events.
3. In the auction example shown in Figure 11.2, the winner depends on the order in which the events are received by the application.

In this section we discuss the difficulties associated with determining the correct order, keeping the events in that correct order, and processing events if they are not in the correct order. These difficulties in ordering stem from several causes: the occurrence time synchronization problem, synchronization of the processing order with occurrence time ordering, and keeping detection time order in a distributed environment.

OCCURRENCE TIME SYNCHRONIZATION

The problem of time synchronization relates to the fact that there may be many event producers in the application, using their own internal clocks to generate event occurrence times. These internal clocks might not be synchronized with each other or with the servers that are running the event processing logic or other analytics. In some cases a producer's clock might be wildly incorrect, but even if it is not there could still be enough inaccuracy to yield wrong results for any order-sensitive processing that uses these occurrence times as the basis for determining the order of the events. There are two approaches that can be used to mitigate this problem: clock synchronization and the use of a time server.

- Clock synchronization is a well researched topic in distributed computing, starting with Lamport's 1978 paper[4], and progressing over the years. The methods developed are

[4] Leslie Lamport: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7): 558-565 (1978)

aimed to ensure that the clocks of all sources will be synchronized. The major problem with this approach is that it requires a degree of central control and also co-operation between event producers. This might be feasible in a bounded environment, where all the relevant event producers can be controlled, and synchronized, but gets hard if there is a large number of producers, or if they are owned and managed independently from one another.

- Time Server: An alternative solution is that all event producers set the occurrence time time-stamp from the same time server[5], and not from their own internal clocks. One obvious example is a GPS device which gets a time-stamp from the GPS satellites at the same time that it is getting a fix on its position. There are various such time servers available through the Internet, and some organizations provide time servers in their intranets. This solution may have some latency, and may not be applicable when the chronon granularity is small; the "time-server" solution is typically considered good enough, and is indeed being used by several event processing implementations. This approach requires that producers of order-sensitive events work under an agreement that they all use the same time server used by all of them, which, again, requires some level of control over the producers' logic.

In cases where it is not feasible to apply either of these approaches, the occurrence time order may be inexact. Section 11.2 discusses this and other sorts of inexact event processing.

ORDERING IN A DISTRIBUTED ENVIRONMENT

In a distributed environment, the order in which events occur may not be identical to the order in which they arrive in the system. This could be because the time taken to propagate event messages through the system varies from message to message, or because the system uses multiple threads to process events. Furthermore, if the entry point to the event processing system is provided by distributed channels, then we can't rely on event `detection times` to give us an accurate ordering, since we have the same synchronization issue that we encountered with `occurrence time`. This can lead to the following anomalies:

1. The `occurrence time` of an event is accurate, but the event arrives out-of-order and processing that should have included the event might already been executed.
2. Neither the `occurrence time` nor detection time can be trusted, so the order of events cannot be accurately determined.

---

[5] NIST time server is an example of such time server, see: http://tf.nist.gov/service/its.htm

There have been some attempts in various systems to cope with the out-of-order issue; the most common solution uses a time-out buffering technique, it is based on the following assumptions:

- Events are reported by the producers as soon as they occur;
- The delay in reporting events to the system is relatively small, and can be bounded by a time-out offset;
- Events arriving after this time-out can be ignored.

Based on these assumptions, time-out reordering is performed by putting the incoming events into a buffer prior to assigning them a `detection time` and sending them to be processed. Let τ be the time-out offset, according to the assumption it is safe to assume that at any time-point t, all events whose `occurrence time` is earlier than t - τ have already arrived. Each event whose `occurrence time` is To is then kept in the buffer until To+τ, at which time the buffer can be sorted by `occurrence time`, and then events can be processed in this sorted order. The main benefit of this technique is that it guarantees that the processing order will be the same as the occurrence time order. In some cases systems assign `detection time` attributes to the events after the sorting has been performed, so that `detection time` can also be used as an accurate metric for temporal order.

This method has two main deficiencies that may or may not be important for various applications:

- Each event has to be delayed by time τ , thus increasing the end to end latency of the processing system;
- Events may be ignored due to late arrival, which may impact the quality of the processing results.
- The assumptions behind this method may also be invalid for some realistic cases: there are cases in which the producers do not themselves sense or instrument the events, but instead simply forward events from their original sources to the EPN, so the delay in reporting may not be negligible; also it may not be acceptable to ignore events that arrive after the time-out.

Another similar method, used by some messaging systems is to assign sequence numbers to events when they are produced, so that gaps in an incoming event stream can be detected and events buffered until the gaps have been filled. This approach has benefits of not having to unnecessarily delay everything by τ, however the delay may not be bounded (unless a time-out is imposed again). It can also be difficult to assign sequence numbers if there are multiple producers, though in some applications, such as those processing time series data, there might be a natural sequence number in the event data itself.

RETROSPECTIVE COMPENSATION

Another way to deal with the case where events miss the processing that they should have participated in because they arrived late, is to compensate for this retrospectively, providing the effect of undo and redo. So rather than delaying the processing of all events just in case some happen to be late, we can go ahead with processing the events that have arrived, and then use a compensation approach to deal with any latecomers there might be. The idea of compensation is used in transaction processing, when dealing with long running business transactions (for example processing of an insurance claim which might takes several days). It's not practical to have a database transaction running for the entire period as you cannot afford to leave data items locked for that long a time. Instead, applications take an optimistic approach and release data locks when some sub-transaction concludes, and if it then turns out that the entire transaction needs to abort, it compensates by generating transactions that undo the original transaction and redo dependent transactions. A similar principle, known as eventual consistency, applies in distributed systems, in which consistency inside the system (for example among replicas of the same data-item) may be sacrificed temporarily inconsistent, but the system is eventually brought back to consistency.

We can borrow from these ideas by arranging for the following actions to occur for each time an "out-of-order" event is detected:

1. Find out all EPAs that have already sent derived events which would have been affected by the "out-of-order" event if it had arrived at the right time.
2. Retract all the derived events that should not have been emitted in their current form[6].
3. Replay the original events with the late one inserted in its correct place in the sequence so that the correct derived events are generated.

This logic is similar to the logic of truth maintenance systems in Artificial Intelligence systems. In practice there are several potential problems with applying it to the "out-of-order" issue:

1. Event processing operations may result in actions done by event consumers, those actions are not part of the event processing system, and so the event consumers need to be able to accept event retractions and perform the appropriate compensation actions. This might not be feasible either because some actions are not undoable, or because a consumer does not have a compensation system implemented.
2. The execution of retrospective processing requires the system to maintain past states. If an EPA is to be able to redo a function then it needs to have access to all relevant information, both the historical events that it was processing, and the past state of any global state elements (e.g. reference data, global variables) that is was using.

---

[6] See further discussion on event retraction in section 11.3

This requires both keeping the history and finding the right events and data-items (temporal databases can help with these issues, but they are not in general commercial use today).

3. A compensation process may have cascading effect in the sense that a single compensation for out-of-order event can trigger a large number of compensating actions, putting a high burden on the system's performance.

For these reasons this solution, while theoretically appealing, is difficult to implement. It might however still be worth considering as an option in some cases, especially when the undo and redo of all consequences are feasible, the information is still available, and the compensation process is bounded.

To conclude, those who develop order-sensitive applications should be aware of the possible anomalies that can occur, their solutions and the possible problems associated with these solutions. The cases where it is not possible to determine the order of events accurately may be handled using inexact event processing tools, as discussed in the next section.

11.2 Inexact Event Processing

Developers and users of event processing systems should be aware of points where event processing may become inexact. The cases that we'll discuss are:

- uncertainty whether an event actually occurred
- inexact content in the event payload
- inexact matching between derived events and the situations they purport to describe

We first explain each of these three issues, and then discuss possible solutions. This whole issue is handled in ad-hoc way, or not handled at all in current systems.

11.2.1 Uncertain events and inexact event content

While you may be certain that a particular event either occurred or did not occur in the real world, there can still be uncertainty in the event objects that report on it in a computer system. Events that occur in the real world may not get reported, while events that have been reported might not have occurred. There are several reasons that may induce this kind of uncertainty:

- An unreliable or imprecise source: An event producer (such as a sensor) may malfunction and indicate that an event has occurred even if it has not. Similarly, an event producer may fail to signal the occurrence of an event which has in fact occurred. In the case of derived events problems in the design or the implementation of the EPA deriving this event can cause it to create false derived events or not create logically valid ones.

- A malicious source: An event could be the direct or indirect result of act that is intended to sabotage the system.
- Projection of temporal anomalies: In the previous section we discussed a number of anomalies which can cause order-sensitive EPAs to process events in an order that is not consistent with the true order of event occurrence. This can cause an EPA to create derived events that should not have been created, or to skip events that should have been created.

Inexact event content occurs when the content of an event object's header or payload is not consistent with one or more of the characteristics the event that happened in reality. The reasons for inexact event content are similar to the reasons for uncertain events.

- The source may be imprecise, for example a badly calibrated thermometer being used to measure someone's fever could yield an incorrect result.
- Temporal anomalies can also lead to incorrect event content in derived events.
- Raw events may contain estimated or sampled data, which are inherently inaccurate.

When an EPA derives further events from inexact input events, this can cause it to propagate uncertainty or inexact content to subsequent phases of the EPN.

Figure 11.3 illustrates the reasons for inexactness and uncertainty in events.

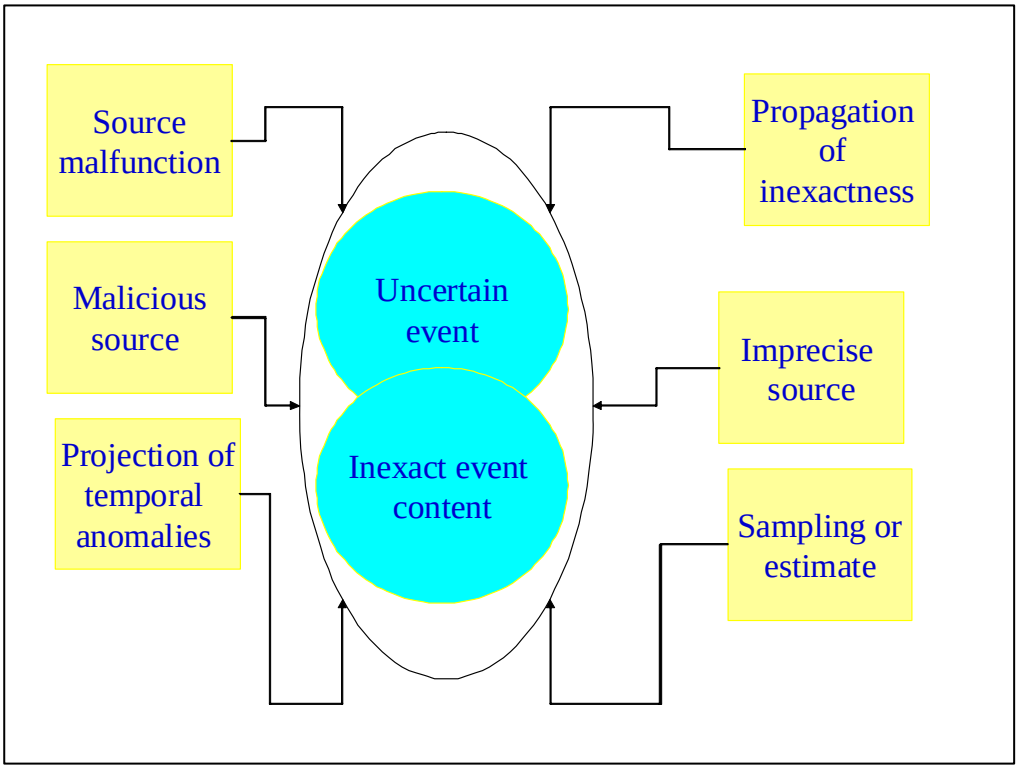


Figure 11.3 Reasons for uncertainty and inexactness in events

Before talking about possible solutions, we need to talk about the other aspect of inexact event processing, which is inexact matching between events and situations.

11.2.2 Inexact matching between events and situations

In chapter 1 we defined a *situation* to be an event that might require a reaction. This definition sits squarely in the domain of users, rather than in the computer domain. In the user domain consideration of what should trigger a reaction depends upon the user's perspective; this is rather different from the computer domain in which everything is determined according to computational processes.

In most cases you might assume that an event in the computer domain, either a raw event detected by an event processing system, or a derived event produced by an event processing system, is exactly what you should use to trigger the reaction, however, in reality this may not be accurate. Consider the case in which an event processing system generates a derived event signaling that a network Denial of Service (DoS) attack has occurred. In this

case the "network DoS attack" is the situation we are looking for, since it obviously requires some reaction. However the computational process that created this derived event is not an exact science, and can have two phenomena associated with it: false positive and false negative situation detection.

Definition

False positive situation detection refers to cases in which an event representing a situation was emitted by an event processing system, but the situation did not occur in reality.

False positives may lead to reactions that should not be performed, and may be damaging. Shutting down activities because of false DoS attacks may be costly; other cases may be even more damaging: false detection of a missile attack might trigger a counter-attack. The converse to false positive situation detection is false negative situation detection.

Definition

False negative situation detection refers to cases in which a situation occurred in reality, but the event representing this situation was not emitted by an event processing system.

False negatives may also be damaging, in the DoS attack example, failure to detect this situation may inflict even more damage than the false positive in this area, and the same is true for the other example of failing to detect a missile attack.

False positives and false negatives may result from uncertain and inexact events. In cases where situations are detected via derived events, false positives and false negatives may be the result of the fact that the computational process used in the derivation cannot guarantee that the situation occurred, but just approximate it. In the DoS attack example, the derived event that detects this situation may be the result of matching one or more patterns known to indicate likely attacks; however these patterns may just approximate the situation and do not indicate it with complete certainty.

Now we have completed our discussion of the issues, we will briefly touch on ways that they can be mitigated.

11.2.3 Handling inexact event processing

The way that inexact event processing is dealt with in systems is often based on assumptions about its frequency and the importance of its implications. There is a spectrum of views, with two extreme positions:

- The first extreme position, which is quite common in current systems, assumes that

these are rare cases, which can be considered as exceptions and can be handled manually. Based on this assumption, the event processing system does not include any feature to handle inexact event processing.

- The other extreme is based on the assumption that inexact event processing is frequent and important enough that every part of the system should behave as if it is inexact, so inexactness is an integral part of the event processing infrastructure.
- Obviously, systems can be somewhere in the middle of this spectrum. For example they could assume that inexact event processing is always required for some event types or event patterns, while for others it can be ignored. In some cases this decision can depend on the context in which the processing occurs.

Techniques to handle inexact event processing may be based on known uncertainty handling frameworks, such as probability based methods (e.g. Bayesian Networks), evidential reasoning (Dampster-Shafer), and fuzzy logic. Probability based methods are the most common ones, they work by associating a probability both to the occurrence of an event and to the accuracy of its content.

Table 11.2 shows some examples of the sorts of probability that can be attached to an event instance.

Table 11.2 Probability indicators associated with an event instance.

Inexact indicator	Probability
Event did not occur	0.4
Event occurred before T1	0.1
Event occurred in [T1, T2]	0.45
Event occurred after T2	0.05

You can see that we can track not only the likelihood of the event occurrence, but also estimate the accuracy of some of its temporal characteristics. One can also assign probabilities to patterns and to derived events, for example assessing the occurrence probability of events produced by a given pattern detection operation, as a function of the certainty or exactness of its input events.

It should be noted that existing tools like Bayesian nets can cope with probabilistic networks and the propagation of such probabilities through an EPN. However in practice, assigning the probabilities is not an easy task to do manually. There are some projects that use machine learning to derive these probabilities, but at present there is no readily available solution that can be applied to the general case.

To summarize, developers and users of event processing systems should be aware of the different cases of inexact event processing and the fact that many current systems view them as rare exceptions that can be dealt with by ad-hoc solutions.

Our last challenging topic relates to relationships among events: retraction and causality.

11.3 Retraction and causality.

In this section we discuss two challenging issues related to the relationships between events. The first of these is the issue of retraction and the second is the issue of causality that can provide traceability in event processing systems.

11.3.1 Event retraction

The Fast Flower Delivery example application includes a facility to allow a customer to cancel an order; we will use this example to demonstrate the challenges involved in retracting events. First let's consider what we want to happen when an order is cancelled. The customer who sent the flower delivery order is not interested in pursuing the order anymore, so you might think that we should treat this like a database transaction and roll back all the operations that have been performed thus far. In practice, however some actions are undoable and some are not, furthermore there may be different ways to perform an undo, depending on the state we happen to be when the cancellation event is received. Figure 11.4 illustrates the retraction possibilities along the life-cycle of the delivery.

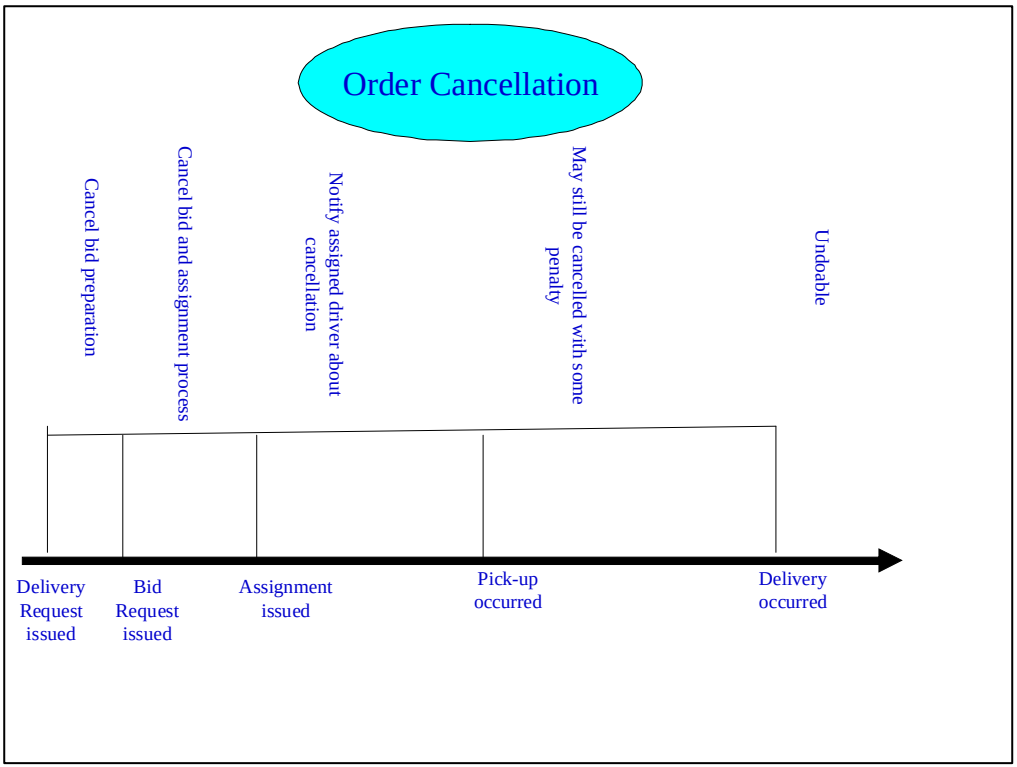


Figure 11.4 Order cancellation implications over the delivery life-cycle.

The implications of cancellation are contingent upon the phase in the delivery life-cycle:

- After the `Delivery Request` has been issued, but before the `bid request` has been issued, the cancellation is realized through aborting the `bid request` preparation process.
- Between the time that the `bid request` is issued, and assignment performed, the cancellation is realized by cancelling the bid and assignment process. In this case, as with the previous one, no action is required to change any plans in the real world.
- Between the time that the assignment is issued and pick-up occurred, the cancellation may be realized by notifying the assigned driver about the cancellation.
- Between the pick-up time and delivery time, the driver can be called to return and not

finish the delivery, however this depends on the store's policy. Some stores might allow it, and some might allow it but with some penalty.

- After the delivery already occurred, it cannot be cancelled of course.
-
- While this analysis is quite reasonable, and in principle might be inferred by the system, in current systems there is no support for retraction as an automated process. Furthermore, since at some points in the delivery life-cycle we needed to go beyond the borders of the EPN, it is also not trivial to understand how all these events are logically related, a topic which we discuss next.

To conclude this topic: developers and users of event processing systems should be aware that retraction needs to be hard-coded, and carefully determine the exact cases of retraction.

11.3.2 Event causality

Event causality is described as a key term in David Luckham's book "The Power of Events". Its definition is quite simple, as can be seen below:

Definition

Event Causality is a relation between two events e1 and e2, designating the fact that the occurrence of the event e1 caused the occurrence of event e2.

The practical importance of this concept is that through causality relations it is possible to trace back the events and computing elements that led to the execution of some action, or the detection of some situation. Looking deeper at the notion of causality we can observe three types of event causality:

- **Type I: predetermined causality**. Take to raw events, e1 and e2 where we know that event e2 always occurs as a result the occurrence of e1. We may thus assume that if e1 has been reported, e2 occurred whether reported or not. This occurrence may also be conditioned, for example some time offset or interval may be attached to this causality.
- **Type II: Induced causality.** The event e1 is an input to an EPA a1, and the derived event e2 is the output of a1.
- **Type III: Potential causality.** The event e1 is an event that is sent from an EPN to a consumer c1. The actions of c1 are beyond the borders of the event processing system, but c1 also acts as an event producer and can produce events of type e2. The event processing system cannot know, without further knowledge, whether there is indeed causality among events e1 and e2, but cannot rule out this possibility.
-

Note that only type II can be automatically inferred from knowledge of the event processing network. We need additional information in order to be able to detect the other two types. This can be done by adding event-to-event relationship information to the event type definition. Such information can either by entered as domain knowledge, or in some cases determined using machine learning techniques, which can learn statistical correlations between events that may approximate causality.

Developers and users of event processing systems should determine whether causality tracing is important to their system, and if so they need to establish the right causality relations so as to be able finding the lineage of events in the system.

11.4 Summary

Event processing can be used for many purposes and provide various benefits due to its abstractions. This chapter provided developers and users of these systems with some insights about challenges they may encounter when using current state-or-the-art tools and techniques. The challenges discussed in this chapter included various types of temporal issues, issues related to inaccuracy of events and event processing, and issues of traceability and retraction of events. System designers should check whether any of these issues apply, and determine how to resolve or mitigate any specific issues they have. The next chapter, which concludes this book, deals with event processing of the future and a summary of what this book is and isn't.

Exercises

11.1 Give an example to an event whose occurrence time is best represented by a temporal element.

11.2 Take each of the relationships between context interval and occurrence time intervals described in Figure 11.1, and provide an example in which this relationship occurs.

11.3 Determine, for each of the derived events in the Fast Flower Delivery example, what the appropriate detection times and occurrence times are.

11.4 Which of the various pattern detection EPAs in the Fast Flower Delivery Example are order-sensitive? Suggest a way to handle out-of-order anomalies in each of them.

11.5 Devise a scenario in which it is practical to use retrospective processing to handle out-of-order events

11.6 Devise a scenario in which all types of inexact processing exist, and show how a probabilistic method can be applied.

11.7 Provide examples of false positives and false negatives in the Fast Flower Delivery Example, and explain how you would mitigate for these anomalies.

11.8 Devise a retraction scenario (not the Fast Flower Delivery example), and explain the different retraction steps and how you would handle them

11.9 Can you find examples of each of the three types of event causality in the Fast Flower Delivery application? if yes, show what they are. If any of them cannot be found in the application devise a scenario in which this causality type does occur.

12

Emerging directions of event processing

"You cannot escape the responsibility of tomorrow by evading it today"
- Abraham Lincoln

We started this book by stating that event processing is an emerging area, and by using the book's website you can experience the current state-of-the-practice. One of the properties of emerging technologies is that they are keep moving. This chapter reflects the personal opinions of the authors about the emerging directions of event processing. We start with a review of some current trends, continuing with a discussion of several areas which are emerging in the technology front. The last section of this chapter serves as epilogue to this book.

12.1 Event processing trends

In this section we discuss several trends that we anticipate will have the most major impact over the direction that event processing will take. The trends we discuss are: going from narrow to wide, going from monolithic to diversified, going from proprietary to standards-based, going from programmer centric development to semi-technical centric development, going from stand-alone to embedded and going from reactive to proactive.

12.1.1 Going from narrow to wide

Every new area starts with early adopters, often centered on one or two specific industries or application types. Event processing is no different; the early adopters of this type of technology were capital market trading applications. Event processing is now spreading to

other industries and is being employed by many different types of application. This has wide implications to the technology. These implications start with the languages; additional use cases require the extension of event processing languages to include more primitives. In addition there are architectural implications, driving the shift from centralized architectures to distributed architectures and from monolithic to diversified architectures. Generalization can also be a driver for standards. The trend of going from narrow to wide is a trigger for some of the additional trends that we discuss.

12.1.2 Going from monolithic to diversified

"One Size Fits All: An Idea Whose Time Has Come and Gone", this is a title of a famous paper by Michael Stonebreaker and Uğur Çetintemel, talking about relational databases and explaining why the authors' opinion is that the single solution approach is not valid anymore. In the event processing area we are still mostly in the "one size fits all" era. As a consequence of "going from narrow to wide", the range of new applications to be supported will require some diversity in implementation technology. This diversity includes:

- Variety of functions: Particular application segments will require particular functions, such as: specific types of transformation and aggregation or trend patterns that are based on advanced statistical functions.
- Variety of Quality of Service requirements: Different EPAs may have different QoS requirements, which require different implementations. Some examples: A certain EPA may require that its internal state be fully recoverable, while other EPAs do not; a certain part of the event processing network may have hard real-time constraints, while the rest of it does not.
- Variety of platforms: Different EPAs may reside on different platforms so the EPN is spread across platforms. For example in an RFID application, one of the EPAs might be embedded inside an RFID reader, while others run on a server.
- This diversity will lead to the development of heterogeneous EPA implementations, and lead to a component-based approach architecture in which an EPN can be built from a collection of EPAs selected from a library of components. Some of these components are generic and some specific to a particular industry or application and provided by niche suppliers. Obviously, standards are vital for achieving diversity, and we discuss them next.

12.1.3 Going from proprietary to standards-based

Some level of standardization will be required if we are to get the position where event processing applications can be assembled out of diverse sets of components instead of being developed for specific monolithic event processing engines. Standardization in the event processing area is a challenge because of the different starting points and approaches that

have been taken so far; this will be evident to readers who have already experimented with the various implementations on this book's website. However standards often emerge when an area of technology starts to mature, and while currently it seems that there is no strong pressure towards standardization, we anticipate that there will be shift towards standardization in the event processing of tomorrow.

There are several avenues for standardization related to event processing:

- Event structure and meta-data representation: As seen from the examples in Chapter 3, there are various ways to represent event meta-data, and differences between products when it comes to header attributes and the kind of data that can be included in an event payload. Standards covering exchange of event type meta-data and runtime event instances would enable interoperability between various components.
- Domain specific event meta-data: By this we mean standardization of actual event types for specific applications or subject areas, for example: system management symptoms, insurance claims, workflow state, and medical devices. Many of these areas have standards today however, each has had to choose its own way to represent event type meta-data and make its own decisions about event structures, in the absence of the standards mentioned in our previous point.
- EPA component model: These standards would define the terminal interfaces used by an EPA to emit and receive events, as well as other runtime interfaces used by an EPA during its lifecycle. It would let someone produce an event processing component that could be hosted in any software environment that supported the model.
- EPA assembly model. This would standardize the language used to express how Producers, EPAs and Consumers are linked together to form EPNs. We have presented a basic assembly model in this book.
- Event distribution standards: These are standards for transporting events between systems (including producers and consumers), and for exchanging meta-data about events and event processing. They include publish/subscribe protocol specifications. There are already some evolving standards in this area like WS-Notification and WS-Eventing.
- Event processing specification meta-modeling: These are standards that will allow modeling event processing functionality. These standards may be extension of existing standards such as UML or BPMN.
- Event processing language: Standardization of the language used to express what an EPA does. This is the toughest area for getting agreement on. Standardization here might be achieved in a phased approach, with the first phase being standardization at the meta-language level, such as the building blocks that have been presented in this book. We think it likely that a standard language will be adopted at some stage, but it might have to wait till the event processing of the day after tomorrow.

Next we discuss the trend related to user types.

12.1.4 Going from programming-centered to semi-technical-centered

The first generation of event processing application development tools is mostly programming-centered in the sense that you must possess programming skills in order to develop event processing applications. We are observing an increasing trend towards allowing business users and business analysts, who might not have deep programming skills, to compose all or part of an event processing application. Figure 12.1 shows a part of a customer survey conducted by ebizQ that indicates the majority of customers surveyed would like to have "event rules" defined by business analysts and business specialists.

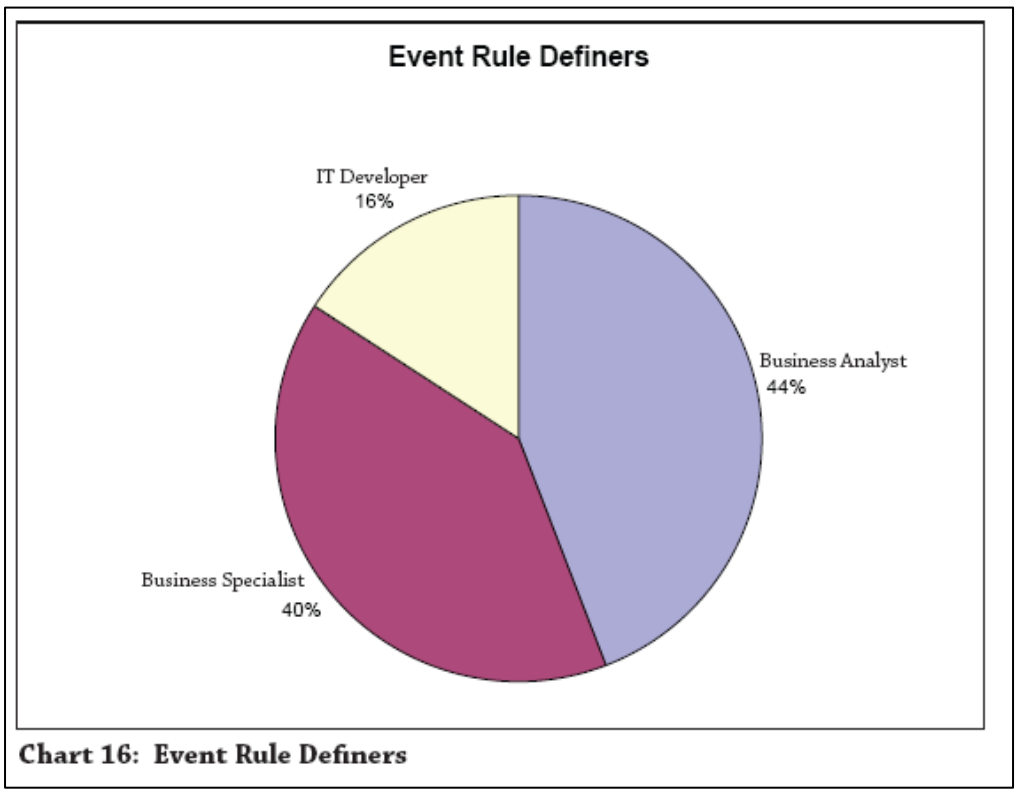


Figure 12.1 A chart taken from ebizQ customers' survey about who are the desired event rule definers.

This trend implies the need for user interfaces and abstraction levels that fit this population. Figure 12.2, showing the interface of IBM Websphere Business Events, is an early example of this trend.

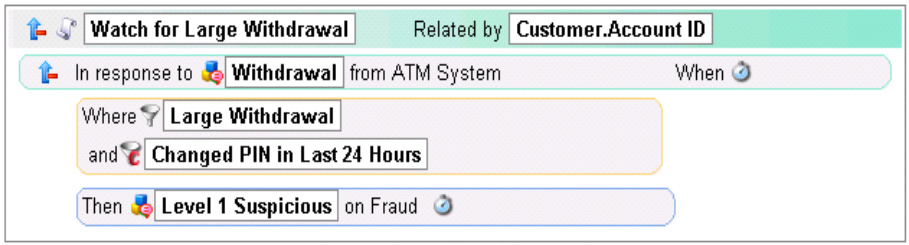


Figure 12.2 An example of a user interface geared for semi-technical developers

We believe that the level of abstraction will go even higher, and future business oriented languages will be based on assertions, intentions and goals. We expect that more work will be dedicated to this topic in the next few years, as it will continue to grow in importance Next we discuss how event processing fits into part of a bigger picture.

12.1.5 Going from stand-alone to embedded

Event processing technology today is delivered in two different ways. The first of these is as a specialist event processing platform, whose primary goal is to support event processing applications. The second way is one where event processing functionality is embedded inside some other piece of software that needs event processing capabilities, either middleware or a packaged application. Some analysts predict that in the future up to 80% of the event processing market will be embedded. We discussed some connected technologies in Chapter 2, and here provide some examples of this trend.

BUSINESS PROCESS MANAGEMENT (BPM)

- The combination of event processing and Business Process Management is sometimes called edBPM (event-driven Business Process Management). The idea is that event processing logic is used to analyze events and detect situations and the BPM part of the system can then react by triggering a new BPM workflow instance, or by stopping or modifying an existing workflow instance. BPM systems can also act as producers of events as we saw in chapter 4.

BUSINESS ACTIVITY MONITORING (BAM)

- Business activity monitoring has emerged in recent years as a category of software in its own right. While early BAM products were batch oriented, the newer generation monitors and analyses information in real time, so as to be able to give up to date information about the state of the business that is being monitored. Many modern BAM products, therefore, are event-driven and embed Event Processing capabilities.

BUSINESS INTELLIGENCE (BI)

- In the Business Intelligence area we have also seen some movement from batch-

oriented to online-oriented analytics, so that businesses can detect and react to fast changing situations in a timely fashion. We expect that BI products will follow the example of BAM products and start to include event processing functionality, there are signs that this is already starting to happen.

MESSAGING ORIENTED MIDDLEWARE (MOM) AND ENTEPRISE SERVICE BUS (ESB)

- Message oriented middleware may be embedded in event processing platforms so as to provide an event transport layer, but the converse could also be the case, with event processing being embedded inside a messaging system to detect patterns in message traffic, and to provide efficient routing decisions based on such patterns (for example a messaging system that saw three messages from a customer to a service center within 2 hours could, route the third message to a supervisor).

Enterprise service buses (ESBs) typically perform filtering and transformation functions, among other Enterprise Integration patterns, and as we saw in chapter 8 there is some partial overlap between them and event processing platforms, although the primary role of enterprise service bus is to provide communication among services. Event processing can assist in some ESB functions, such as selection of services, routing decisions, and validation decisions, as these can be based on event processing patterns.

PACKAGED APPLICATIONS

Event processing functions may also be embedded inside packaged applications; notable examples being the use of event processing in network and system management applications and its use In trading platforms found in capital markets. More examples are emerging,

12.1.6 Going from reactive to proactive

Event processing today is used largely in a reactive manner, where a system needs to take some action as a result of an event or a series of events that have already happened. Event processing is used in such applications to analyze events and detect situations that need to be handled.

In *proactive computing*, on the other hand, the emphasis is on detecting undesirable states so that they can be eliminated, or at least mitigated, before they give rise to unwanted (usually bad) consequences. A good example of proactive computing is its use in predicting traffic congestion before traffic actually comes to a standstill so that steps can be taken to manage the traffic flow to prevent a jam. This can be done at a city level, but in Figure 12.3 we show a smaller scale example where proactive computing is used to set traffic light policies (the timings for red, yellow and green traffic lights) within an area of a city.

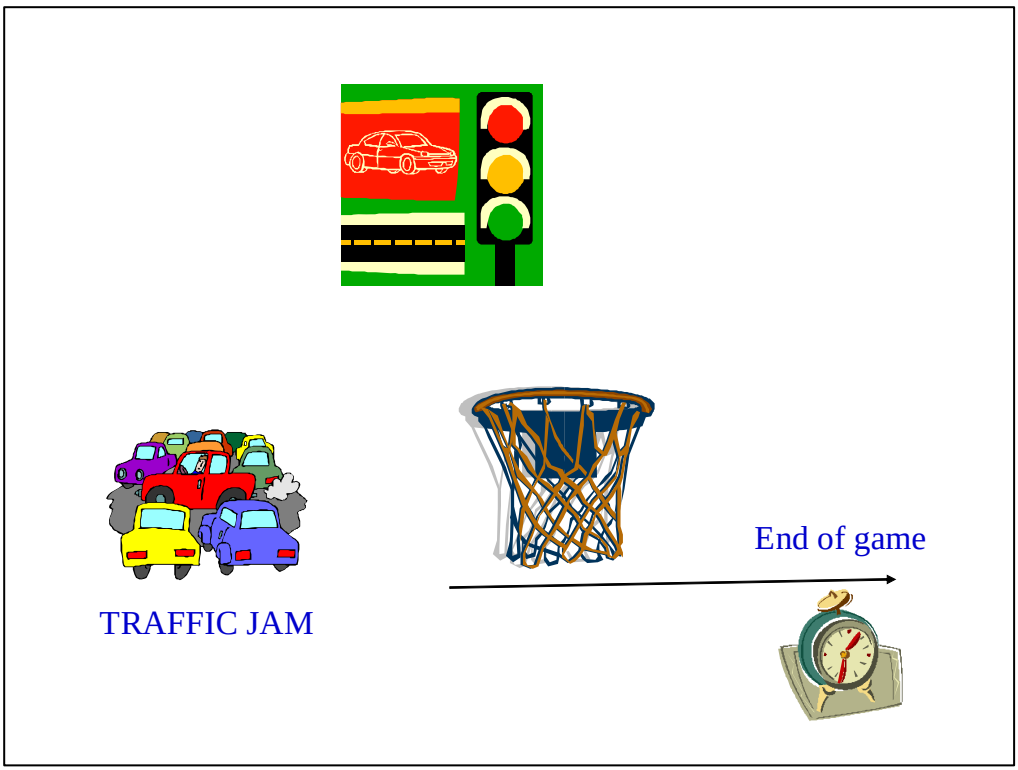


Figure 12.3 Use of proactive computing to control traffic by resetting the traffic light policies based on expected events such as the end of a basketball game

Traffic management like this can be performed in a simple manner just by reacting to observed traffic conditions. For example if the system sees that there is a heavy build-up of traffic in one direction, while other directions have relatively sparse traffic, then it can adjust the timings to give more time to the busy traffic stream. Traffic light policies can also be based on other information. In the example shown in the figure, a specific traffic light policy is set when a basketball game is due to end, so as to limit congestion caused by spectators leaving the game. If the game then goes into extra time, then the system might switch back to the original policy until the extension is over.

Proactive computing involves other technologies besides event processing, such as predictive analytics to identify possible future outcomes and to select between appropriate courses of action.

Now we have looked at these trends we survey some of the developments in technology that we anticipate will affect event processing platforms and products

12.2 Future directions in event processing technology

In this section we discuss some of the emerging technological directions that we observe in the event processing area. We discuss event processing virtual platforms, event processing optimization, event processing software engineering and intelligent event processing.

12.2.1 Event processing virtual platforms

Figure 12.4 shows some of the many different kinds of platform that are used to run today's event processing applications.

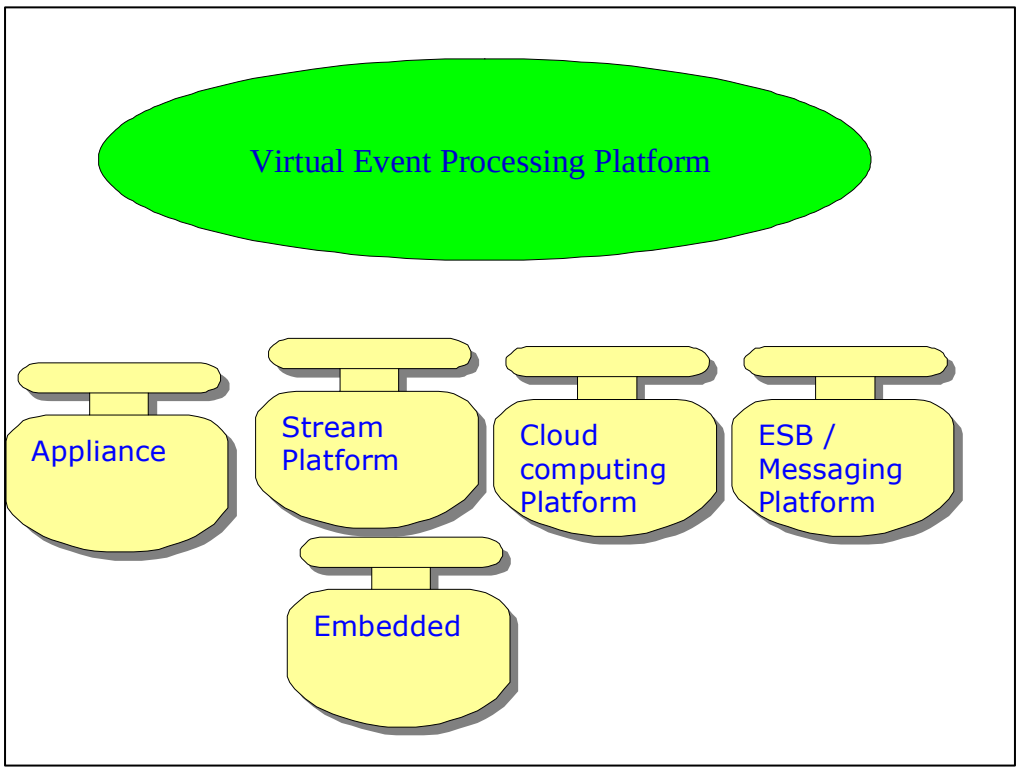


Figure 12.4 A Virtual event processing platform can replace today's multiple event processing platforms.

These platforms include:

- Hardware appliance platforms: These are specialist hardware platforms, often with many processor cores, dedicated to running event processing. The event processing

software that runs on them is usually specially tuned to the hardware capabilities. In the multi-core machine cases, the system can perform automatic parallelization of the event processing logic.

- Stream platforms: A stream platform is a platform for processing streams of information, such as video, audio or news information. The applications that run on these platforms, for example surveillance or traffic offence detection applications, frequently make use of event processing.
- Cloud computing: Cloud computing platforms are growing in use. It is expected that many event processing applications will run on grid platforms. ESB/messaging platform: As event processing becomes part of SOA applications, the connection between events and services are done by ESB platforms.
- Embedded platforms: There are many specialized platforms in which event processing may be embedded. For example robotic platforms, RFID readers, home appliance gateways and more. As we saw in the previous section, there is a trend towards having more embedded event processing.

It is clear that it is not cost effective to build different event processing software for each of these platforms, so the alternative is to construct virtual event processing platform that can be mapped to each of the hosting platforms in an efficient way. There are already some indications that this is beginning to happen.

12.2.2 Event processing optimization

- Relational databases became pervasive after the introduction of query optimization. The trend of "growing from narrow to wide" will necessitate more work on optimization issues in event processing. An optimization decision is relative to an objective function (the metric that we are trying to optimize); we have discussed such optimization metrics in chapter 10.

Note that each of these objective functions entails a different type of optimization. For example with a Java implementation you might minimize maximal latency by smoothing the Java garbage collection process, making it continuous rather than discrete. However this raises the average latency, so if the objective function is "minimize average latency" other methods should be used.

The most common optimization approach being used at present is "black box optimization". In this approach you take the implementation of the EPAs as given, and optimize the partition of EPAs to threads, cores and machines, and their relative scheduling. We anticipate a trend towards "white box" optimization, where the optimization process has the ability to vary the code used to implement the EPAs themselves in order to obtain an implementation that best meets the requirements of the objective function given the particular circumstances of the application.

12.2.3 Event processing software engineering

Event processing requires a slightly different type of thinking than traditional computing. We have covered some of this thinking in this book, starting from the decoupling principle that we discussed in Chapter 2, and moving through the other concepts that we introduced; there is a need to devise software engineering methodologies and tools to support this kind of thinking. This will be realized through methodologies and supported tools, design patterns and best practices collections, and also be assisted by modeling and meta-modeling standards.

12.2.4 Intelligent event processing

Intelligent event processing is area subject that brings together a number of extensions to the event processing technologies we have covered in this book. These include: pattern acquisition, handling inexact and uncertain events, handling predicted events.

- The pattern detection EPAs that we discussed in chapter 9 all have to be programmed in some way with details of the specific pattern that they are to detect. We have been assuming that the application designer is aware of what these patterns are when developing the application, and that the patterns just form part of the application specification. In many cases, for example patterns used to monitor compliance with regulations, this is a fair assumption. However there are other cases, for example fraud detection or the traffic congestion prediction example we mentioned earlier, where you may not know exactly what you are looking for when first designing the application. Acquisition of event processing patterns in these cases is not always straightforward. There are several techniques that are originated from the Artificial Intelligence area that might be used. These are knowledge acquisition techniques that have been used in expert systems, and machine learning techniques can be used to examine historical events and learn the patterns from them. This machine learning technique can be assisted by data mining tools, neural networks and other such techniques.
- In Chapter 11 we discussed inexact and uncertain event processing issues. Handling inexact and uncertain event processing can be achieved using techniques devised to handle uncertain reasoning, such as Bayesian Networks.
- The "from reactive to proactive" trend requires the prediction of events, and the handling of such predicted events. Another branch of intelligent event processing is the handling of causality networks, which consist of semantic relations between events and entities. These causality relations have to be acquired (in similar methods to pattern acquisition) and processed. Causality relations also have a temporal dimension that represents the delay between cause and effect. Extensions in this area will take event processing further towards the support of proactive computing.

These trends and technology advancements are some of what we anticipate in the event processing of the future. Of course, as the time passes the future will become present, and

new trends and features will be seen in the horizon. With this glance to the future, it is now time to summarize this book.

12.3 Epilogue to the book

This is the end of our journey through this book, but the event processing journey is only starting. The event processing area is still a young subject, and will most certainly evolve in the future. We hope that this book has provided a solid foundation for the understanding of the concepts and facilities of event processing. The building block approach we have used is intended as a way to understand the concepts, as well as the gate to future programming of event processing systems, which we believe will be done using a level of abstraction similar to this. If you wish to get hands-on experience with some of the various approaches to event processing you are welcome to use the different languages through our website. In fact, this website will also help to keep this book as a living entity with updates, a live forum, and contact with other readers.

Appendix A
Definitions

This appendix provides lexicographic glossary that contains all definitions done in the book.

A

The **absence pattern** is satisfied when there are no relevant events (9.2.1)

An **aggregate EPA** is a transform EPA that takes as input a collection of events and creates a single derived event that apply some aggregation function over the input events (6.2.4)

The **all pattern** is satisfied when the relevant event set contains at least one instance of each event type in the participant set (9.2.1)

The **always pattern** is satisfied when all the relevant events satisfy the always pattern assertion (9.2.4)

The **any pattern** is satisfied if the relevant event set contains an instance of any of the event types in the participant set (9.2.1)

Availability is the percentage of the time in which a certain system is perceived as functioning by its users (10.2.2)

The **average distance pattern** is satisfied when the average distance of event locations for all the relevant events, satisfies the average distance threshold assertion (9.3.3)

C

A cardinality policy is a semantic abstraction that controls how many matching sets are created and also determines the time when they are created. The possible policies are: single, single deferred, unrestricted and bounded (9.4.3)

A **common attribute** is an event attribute whose semantics are defined by the attribute name, so within the application domain all attributes with the same name are considered to be semantically equivalent (3.3.2)

A **compose EPA** is a transform EPA that takes groups of events from different input terminals, looks for matches using some matching criterion and then creates derived events based on these matched events (6.2.4)

A **composite context** is a context that is composed from two or more contexts. The set of context instances for the composite context is the Cartesian product of the instance sets of its constituent contexts (7.7)

A **context** is a named specification of conditions that groups together event instances for the purpose of processing them together. A context may have one or more *context dimensions* and consist of one or more *context instances* (7.1)

A **context initiator policy** is a semantic abstraction that defines the behavior required when a window has been opened and a subsequent initiator event is detected. The possible policies are: open another window, ignore the new initiator event, refresh the window or extend the window (7.6)

A consumption policy is a semantic abstraction that defines whether an event instance is consumed as soon as it is included in a matching set, or whether it can be included in subsequent matching sets. Possible consumption policies are: consume, reuse and bounded reuse (9.4.4)

The **count pattern** is satisfied when the number of instances of all relevant events satisfies the count threshold assertion *(9.2.2)*

D

The decreasing pattern is satisfied for a value of an attribute A if for all the relevant events, e1 << e2 ⇒ e1.A > e2.A (9.3.2)

A **Derived Event** is an event that is generated as a result of event processing that takes place inside the Event Processing Network (1.2.1)

The **Detection Time** attribute is a time stamp (in the event type's temporal granularity) that records the time in which the event became known to the event processing system (3.2.2)

E

An **enrich EPA** is a subtype of the translate EPA that takes a single input event, matches it against a global store element, and creates a derived event which includes the original event, with possible modified attributes, and an additional collection of attributes {A1,…,An} copied or calculated as a result of using the global state (6.2.4)

An **Entity distance location context** assigns events to context instances based on their distance from an entity location that is either specified by an event attribute or is a fixed entity (7.3.2)

An event is an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain. The word *event* is also used to mean a programming entity that represents such an occurrence in a computing system (1.1.1)

The **Event annotation** attribute provides a free-text explanation of what happened in this particular event (3.2.2)

The **Event certainty** attribute denotes an estimate of the certainty of this particular event (3.2.2)

Event Causality *is a relation between two events e1 and e2, designating the fact that the occurrence of the event e1 caused the occurrence of event e2 (11.3.2)*

An event channel is a processing element that receives events from one or more source processing elements, makes routing decisions, and sends the input events unchanged to one or more target processing elements in accordance with these routing decisions (6.3.1)

The **Event composition** attribute is a Boolean attribute that denotes whether the specific event instance consists of composition of several events or not (3.2.1)

An Event Consumer is an entity that receives events (1.2.1)

An Event distance location context assigns events to context instances based on their distance from the location of another event (7.3.3)

An **Event Entity Reference** is an event attribute whose value is a reference to a particular entity external to the event (3.3.2)

The Event Generalization and Specialization relationships indicate that an event type is a generalization or specialization of another event type, possibly conditioned by a predicate (3.4).

The Event Identity attribute is a system generated unique id for each individual event instance (3.2.2)

An **Event Initiator synonym condition** occurs when there is more than one event instance that can be assigned to an event parameter (7.6)

An **Event Initiator policy** is a semantic abstraction that defines the behavior when event Initiator synonym is detected. The possible policies are: add another interval, ignore the new initiator and refresh the event interval (7.6)

In an **event interval context** each window is an interval that starts with the occurrence of an event that satisfies some predicate and terminates with an occurrence of another event that satisfies a predicate, or when a given period has elapsed. (7.2.2)

An Event synonym condition occurs when there is more than one event instance of the same event type within the relevant event set of a certain pattern matching (9.5.1)

An Event synonym policy is a semantic abstraction that defines the behavior when event synonyms condition is detected for pattern. The possible policies are: override, every, first, last, with largest value, with smallest value (9.5.1)

Event Processing is computing that performs operations on events. Common Event Processing operations include reading, creating, transforming and deleting events (1.2.1)

An Event Processing Agent is a software module that processes events (1.2.1)

An **Event Processing Network** is a collection of event processing agents, producers, consumers and global states, connected by a collection of channels (1.2.3)

An Event Producer is an entity that emits events (1.2.1)

The **Event Source** attribute is the name of the entity that originated this event. This can be either an event producer or an event processing agent (3.2.2)

An **event stream** is a set of associated events. It is often a temporally totally ordered set (that is to say that there is a well-defined timestamp-based order to the events in the stream). A stream in which all the events must be of the same type is called a *homogeneous* event stream; a stream in which the events may be of different types is referred to as a heterogeneous event stream (1.2.5)

An **event type** is a specification for a set of event objects that have the same semantic intent and same structure; every event is considered as an instance of an event type (3.1)

The **Event type identifier** attribute identifies the event type definition that describes the event instance (3.2.1)

F

False negative situation detection *refers to cases* in which a situation occurred in reality, but the event representing this situation was not emitted by an event processing system (11.2.2)

False positive situation detection *refers to cases* in which an event representing a situation was emitted by an event processing system, but the situation did not occur in reality (11.2.2)

A **filter EPA** is an EPA that performs filtering only and has no matching or derivation steps, so it does not transform the input event (6.2.3)

In a **fixed interval context** each window is an interval that has a fixed time length; there may be just one window or a periodically repeating sequence of windows. (7.2.1)

A **fixed location context** has a single predefined context instance based on event location. The event location is either determined directly by the value of a location attribute or by mapping of a location attribute to another spatial entity (7.3.1)

I

The **increasing pattern** is satisfied for a value of an attribute A if for all the relevant events, e1 << e2 $\Rightarrow$ e1.A < e2.A (9.3.2).

L

A **location data type** is used to designate the location in which an event occurred in the "real world"; it can refer to domain-specific geo-spatial terms, e.g. lines and areas that are defined in this domain (3.3.1)

M

The **max distance pattern** is satisfied when the maximal distance of event locations for all relevant events satisfies the min threshold assertion (9.3.3)

The **min distance pattern** is satisfied when the minimal distance of event locations for all the relevant events, satisfies the min distance threshold assertion (9.3.3)

The mixed pattern is satisfied for a value of an attribute A, if within the relevant events exist e1, e2, e3, e4 such that: e1 << e2 and e1.A < e2.A and e3 << e4 and e3.A > e4.A (9.3.2)

The Moving in a consistent direction pattern is satisfied if there exists a direction from the set {north, south, east, west, northeast, northwest, southeast, southwest} such that for any pair of relevant events e1, e2 we have e1 << e2 $\Rightarrow$ e2 lies in that direction relative to e1 (9.3.4)

The Moving in a mixed direction pattern is satisfied if none of the eight **moving in a consistent direction pattern** is satisfied (9.3.4)

The **moving toward** pattern is satisfied when for any pair of relevant events e1, e2 we have e1 << e2 $\Rightarrow$ the location of e2 is closer to a certain object then the location of e1 (9.3.4)

N

The **non decreasing pattern is** satisfied for a value of an attribute A if for all instances of the participant event type within the context that satisfy the assertions, if e1 << e2 $\Rightarrow$ e1.A $\leq$ e2.A (9.3.2)

The non increasing pattern is satisfied for a value of an attribute A if for all relevant events e1 << e2 $\Rightarrow$ e1.A $\geq$ e2.A (9.3.2)

The **not selected pattern** is satisfied when there is a relevant event which is not a member of any matching set of the patterns specified in the not selected assertion (9.2.4)

O

The **Occurrence Time** attribute is a time stamp with a precision given by the event type's temporal granularity (Chronon). It records the time at which the event occurred in the external system (3.2.2)

An order policy is a semantic abstraction that defines the meaning of the << temporal order of the event instances in the relevant event set. The possible policies are: by occurrence time, by detection time, by user-defined attribute, or by stream position (9.4.2)

P

The **Participants set** is a predefined set of event types that form part of the pattern matching function. The order of these event types has importance for some pattern functions (9.1.2)

A **Pattern** is a function that takes a collection of input event instances and produces a matching set that consists of zero or more of those input events (9.1.2)

A **Pattern assertion** is an assertion that is used as part of the matching process (9.1.2)

A **pattern context** is a single context associated with the pattern (9.1.2)

A **pattern detection EPA** is an EPA that performs a pattern matching function on one or more input streams. It emits one or more derived events if it detects an occurrence of the specified pattern in the input events (6.2.6)

A **pattern policy** is a named parameter that disambiguates the semantics of the pattern and the pattern matching process (9.1.2)

A Pattern matching set is the output of the pattern matching process; it is a subset of the relevant events (9.1.2)

A **Pattern type** is a label that determines the meaning and intention of the pattern and specifies the particular kind of matching function to be used (9.1.2)

A **project EPA** is a subtype of the translate EPA that takes an input event, and creates a single derived event that contains a subset of the input event attributes (6.2.4)

R

A **Raw Event** is an event that is introduced into an event processing network by an event producer (1.2.1)

Recoverability is the ability to restore the state to its exact value before a failure occurred (10.2.2)

The relative average distance pattern is satisfied when the average distance among the distances of any two relevant events satisfies the relative average threshold assertion (9.3.3)

The relative max distance pattern is satisfied when the maximal distance among the distances of any two relevant events satisfies the max threshold assertion (9.3.3)

The relative max pattern is satisfied by the event which has the maximal value of a specific attribute over all the relevant events (9.2.3)

The **relative min distance pattern** is satisfied when the minimal distance among the distances of any two relevant events satisfies the min distance threshold assertion (9.3.3)

The relative min pattern is satisfied by the event which has the minimal value of a specific attribute over all the relevant events (9.2.3)

The Relevant events for a specific pattern are those event instances that occur within the pattern's context, which are instances of the event types listed in the participant set list and which satisfy the relevance assertion (9.1.2)

A **repeated type condition** occurs when the relevant event set contains more than one event instance of the same event type (9.4.1)

A Repeated type policy is a semantic abstraction that defines the behavior when a repeated type condition occurs in a pattern's relevant event set. The possible policies are: override, every, first, last, with largest value, with smallest value (9.4.1)

A **Retraction event** relationship is a property of an event type referencing a second event type that is the logical retraction of the referencing event type (3.4)

A **routing scheme** denotes the type of information used by the channel to make a routing decision. The possible routing schemes are: fixed, subscription-based, itinerary-based, type-based and content-based (6.3.2)

A **run-time event processing network** is a directed graph, whose nodes are platform-specific run-time software artifacts that implement EPAs, channels, producers, consumers, global states, and whose edges denote the flow of specific event instances (6.1.1).

S

Scalability is the capability of a system to adapt readily to a greater or lesser intensity of use, volume, or demand while still meeting its business objectives (10.2.1)

A **Segmentation oriented context** assigns events to context instances based on the values of one or event attributes, either using the value of these attribute(s) or using predicate expressions to define context instance membership (7.5)

The sequence pattern is satisfied when the relevant event set contains at least one event instance for each event type in the participant set, and the order of the event instances is identical to the order of the event types in the participant set (9.3.1)

Set-at-a-time processing scheme denotes that the EPA is being invoked when a collection of (zero or more) events have arrived (1.2.5)

A sliding event interval is an interval of fixed number of event instances that continuously slides on the time axis (7.2.4)

In **a sliding fixed interval context** each window is an interval with fixed temporal size or fixed number of events. New windows are opened at regular intervals relative to one another (7.2.3)

A **situation** is an event occurrence that might require a reaction (1.1.1)

The **sometimes pattern** is satisfied when there is at least one relevant event that satisfies the sometimes pattern assertion (9.2.4)

A **split EPA** is a transform EPA that takes as an input a single event and creates a collection of events, each of them can clone the original event, or project the event, e.g. taking a subset of its attributes (6.2.5)

The stable pattern is satisfied for a value of an attribute A if for all the relevant events, $e1 << e2 \Rightarrow e1.A = e2.A$ (9.3.2)

In **State oriented context** events are grouped of based on a state of an external entity that is in effect when the event occurs or is detected (according to the temporal order of this context) (7.4)

Stateful event processing agent. An event processing agent is said to be *stateful* if it can generate derived events whose content is influenced by more than one input event (1.2.5)

The stationary pattern is satisfied if the location of all relevant events is identical (9.3.4)

T

A **temporal element** *is a non overlapping collection of time intervals (11.1.1)*

The Temporal Granularity (or Chronon) attribute denotes the "atom of time" from a particular application's point of view. It stands for the unit in which time-stamps in the application are being measured, examples: second, minute, hour, or day (3.2.1)

A time interval *is a data type that designates a continuous segment in time, starting at a time point (Ts) and ending at a time point (Te). A time point t is part of a time interval (TI) if:* $Ts(TI) < t <$ Te (TI) (11.1.1)

A **time stamp is** a data type that denotes a certain point in time, its granularity is based on the chronon that applies to the event type (3.3.1)

A **transform EPA** is an EPA that performs the derivation function, and optionally also the filtering function. (6.2.4)

A **translate EPA** is a transform EPA that takes as an input a single event, and generates a single derived event which is a function of the input event, using a translation formula (6.2.4)

V

The **value average** pattern is satisfied when the average value of a specific attribute over all the relevant events satisfies the value average threshold assertion (9.2.2)

The **value max** pattern is satisfied when the maximal value of a specific attribute over all the relevant events satisfies the value max threshold assertion (9.2.2)

The **value min** pattern is satisfied when the minimal value of a specific attribute over all the relevant events satisfies the value min threshold assertion (9.2.2)

Appendix B

The Fast Flower Delivery Example

This appendix shows the complete description of the Fast Flower Delivery example using the definition elements we provided in this book. It consists of definitions of the various parts of this example, taken from the various parts of the book: Event types, contexts, event processing network: consumers, event processing agents, producers, and channels.

Figure B.1 shows the event processing network in this example (excluding channels). The event processing network has been drawn using the EPDL editor that can be downloaded from: http://code.google.com/p/epdleditor/

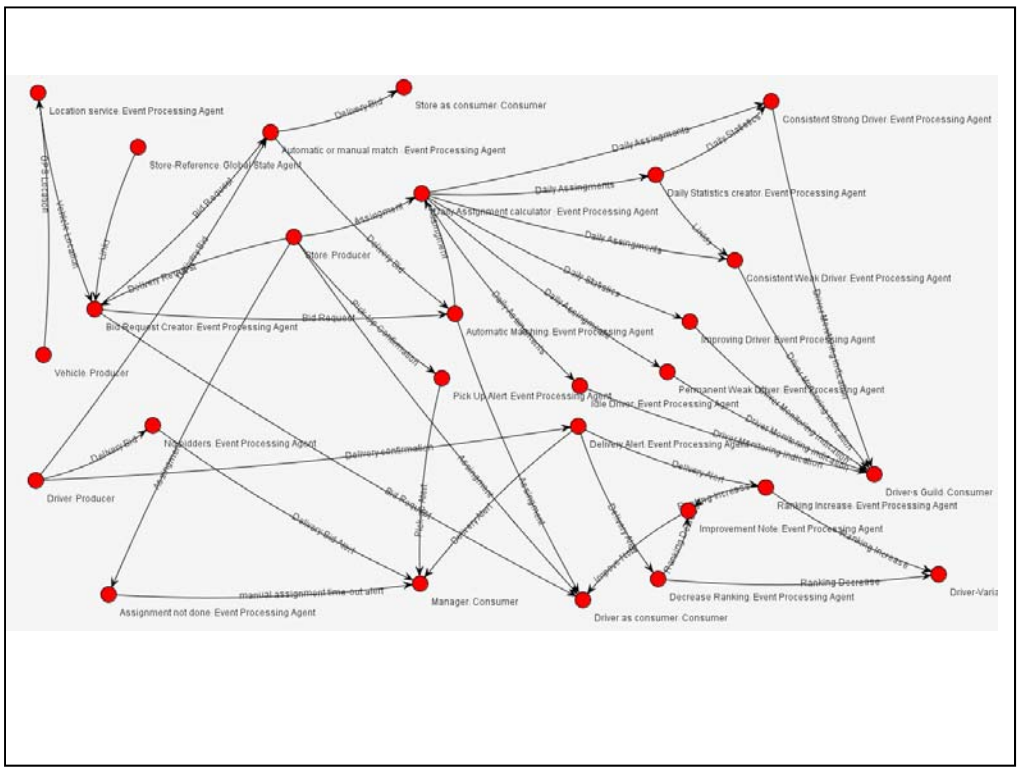


Figure B.1. Drawing of the event processing network

This figure shows the entire event processing network (excluding channels) on one chart, however, defining sub EPNs and show them separately may make sense when the EPN is large enough. We now recall all the details of this example.

B.1. Specification of the Fast Flower Delivery example

This specification is taken from Chapter 1, Section 1.3.

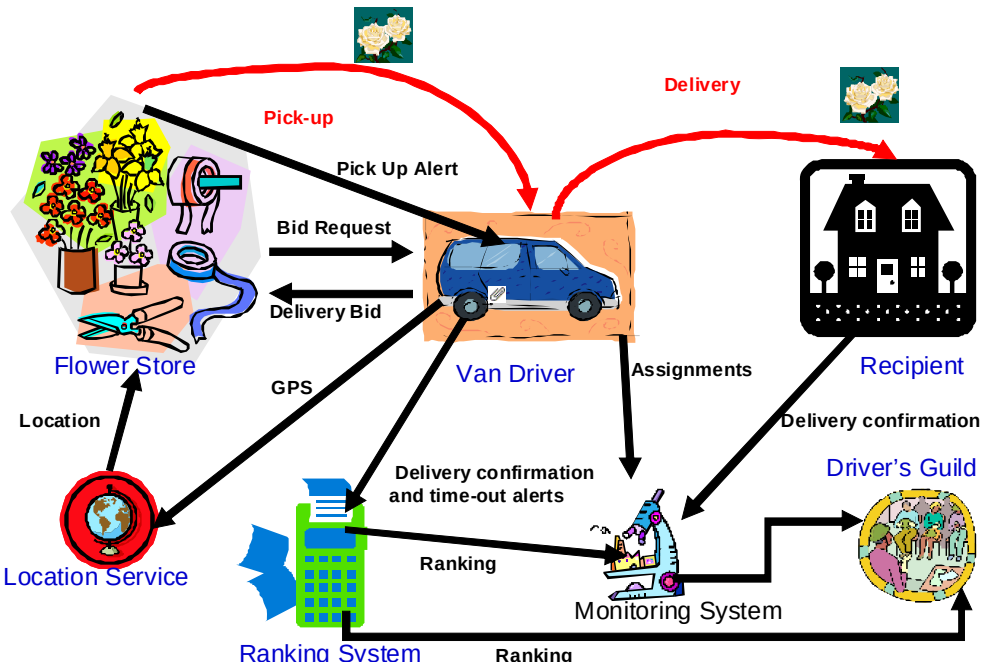


Figure B.2 An illustration showing the various parts of the "Fast Flower Delivery" example

In Figure B.2 the black arrows represent event flows, the pictures represent the various entities, labeled in blue, and the red curved arrows represent an actual driver's journey from a flower store to a recipient. Next we describe the example.

B.1.1. General description

The flower stores association in a large city has established an agreement with local independent van drivers to deliver flowers from the city's flower stores to their destinations. When a store gets a flower delivery order it creates a request which is broadcast to relevant drivers within a certain distance from the store, with the time for pick up (typically now) and the required delivery time if it is an urgent delivery. A driver is then assigned and the customer is notified that a delivery has been scheduled. The driver picks up the delivery and delivers it, and the person receiving the flowers confirms the delivery time by signing for it on the driver's mobile device. The system maintains a ranking of each individual driver based on his or her ability to deliver flowers on time. Each store has a profile that can

include a constraint on the ranking of its drivers, for example a store can require its drivers to have a ranking greater than 10. The profile also indicates whether the store wants the system to assign drivers automatically, or whether it wants to receive several applications and then make its own choice.

B.1.2. Skeleton Specification

These are the five phases of the skeleton specification:

PHASE 1: BID PHASE

The communication between the store and the person who makes the order is outside the scope of the system, so as far as we are concerned a delivery's life-cycle starts when a store places a `Delivery Request` event into the system. The system *enriches* the `Delivery Request` event by adding to it the minimum ranking that the store is prepared to accept (each store has different level of tolerance for service quality). Each van is equipped with a GPS modem which periodically transmits a `GPS Location` event. The system *translates* these events, which contain raw latitude and longitude values, into events which indicate which region of the city the driver is currently in. When it receives a `Delivery Request` event the system matches it to its list of drivers. A filter is applied to this list to select only those authorized drivers who satisfy the ranking requirements and who are currently in nearby regions. A `Bid Request` event is then broadcast to all drivers that pass this filter.

PHASE 2: ASSIGNMENT PHASE

A driver responds to the `Bid Request`[1] by sending a `Delivery Bid` event designating his or her current location and committing to a pick up time. Two minutes after the broadcast the system starts the assignment process. This is either an automatic or a manual process, depending on the store's preference. If the process is manual the system collects the `Delivery Bid` events that match the original `Bid Request` and sends the five highest-ranked of these to the store. If the process is manual, the store makes the assignment and creates an `Assignment` event that is sent to the system; if the process is automatic then the first bidder among the selected drivers wins the bid, and the `Assignment` event is created by the processing system. The pickup time and delivery time are set and the `Assignment` is sent to the driver.

There are also some alerts associated with this process: If there are no bidders an alert is sent both to the store and to the system manager; if the store has not performed its manual assignment within one minute of receiving its `Delivery Bid` events then both the store and system manager receive an alert.

---

[1] Note that the term "Request" here means a message that requests drivers to bid; it should not be confused with a service request issued in the "request-response" protocol.

PHASE 3: DELIVERY PROCESS

When the driver arrives to pick up the flowers from the store, the store sends a `Pick up Confirmation` event; when the driver delivers the flowers, the person receiving them confirms by signing the driver's mobile device, and this generates a `Delivery Confirmation` event. Both `Pick-Up Confirmation` and `Delivery Confirmation` events have time-stamps associated with them, and this allows the system to generate alert events. A `Pick-Up Alert` is generated if a `Pick-Up Confirmation` was not reported within five minutes of the committed pick up time. A `Delivery Alert` is generated if a `Delivery Confirmation` was not reported within ten minutes of the committed delivery time.

PHASE 4: RANKING EVALUATION

The system performs an evaluation of each driver's ranking every time that that driver completes 20 deliveries. If the driver did not have any `Delivery Alerts` during that period then the system generates a `Ranking Increase` event indicating that the driver's ranking has increased by one point. Conversely if the driver has had more than five delivery alerts during that time then the system generates a `Ranking Decrease` to reduce the ranking by one point. If the system generates a `Ranking Increase` ***for a driver whose previous evaluation*** had been a ***Ranking*** `Decrease` then it generates an `Improvement Note`.

PHASE 5: ACTIVITY MONITORING

The system aggregates assignment and other events and counts the number of assignments per day for each driver for each day on which the driver has been active. Once a month the system creates reports on drivers' performance, assessing the drivers according to the following criteria:

- A *permanent weak driver* is a driver with fewer than five ***assignments*** on all the days on which the driver has been active.
- An *idle driver* is a driver with at least one day of activity which had no ***assignments.***
- A *consistent weak driver* is a driver, whose ***daily assignments*** are at least two standard deviations lower than the average assignment per driver on each day in question.
- A *consistent strong driver* is a driver, whose daily ***assignments*** are at least two standard deviations higher than the average assignment per driver on each day in question.
- An *improving driver* is a driver whose ***assignments*** increase or stay the same day by day.
-

As we have said, this use case accompanies us throughout the book, and provides us with a good view into the various functions performed by an event processing system.

Next we explain the notion of a building block and show how we use building blocks to specify Event Processing applications.

B.2 Event Type definitions

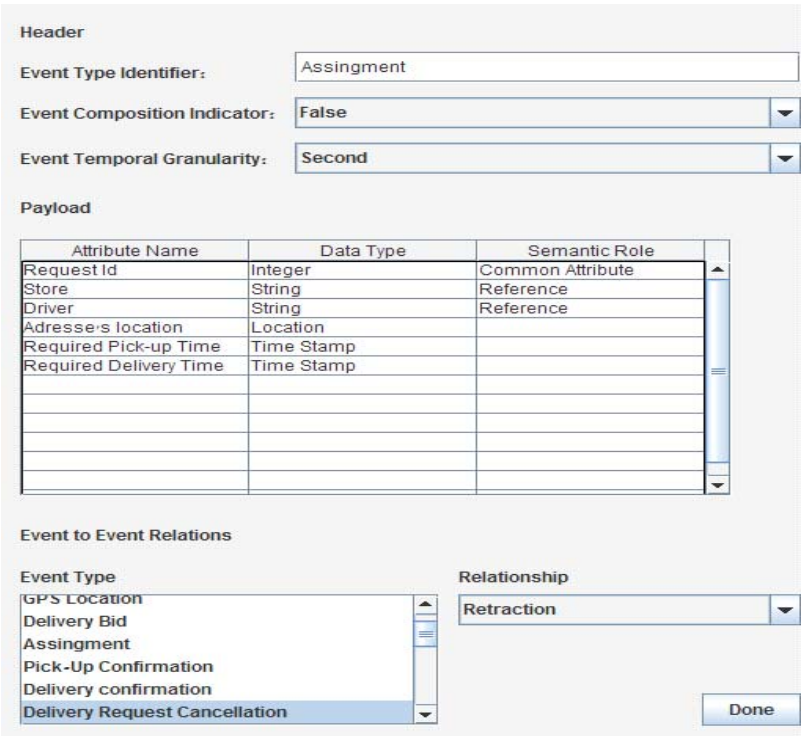
Table B.1 lists of all event types in this example, in alphabetical order.

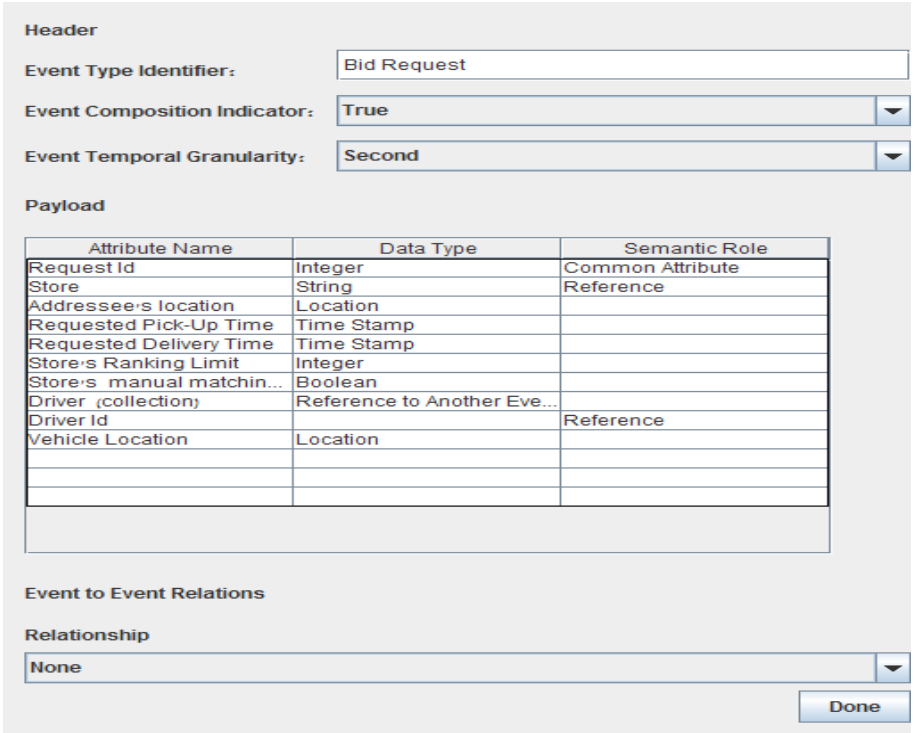
Table B.1 Event Types

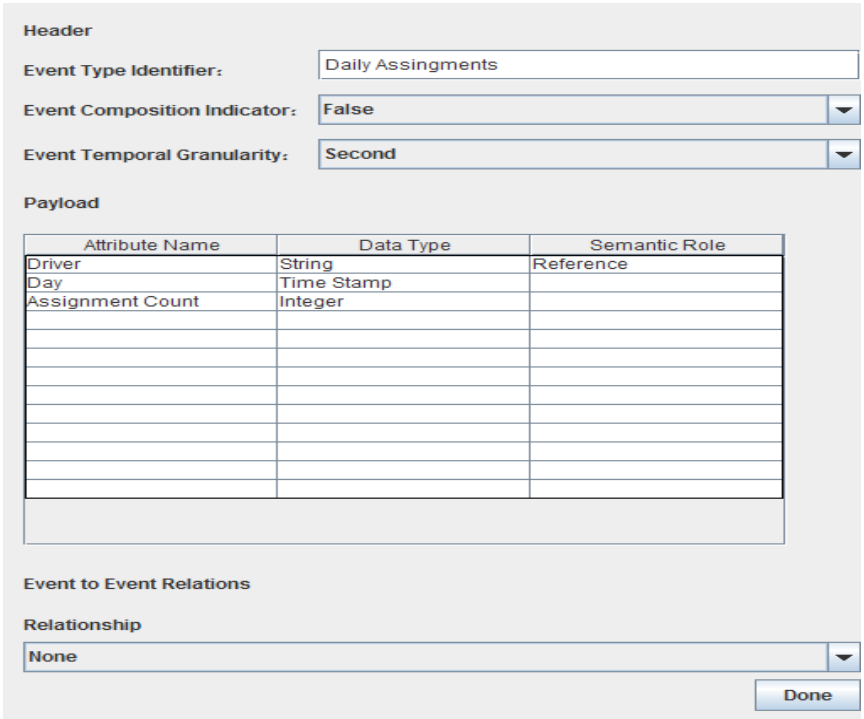
Event Type	Raw	Derived
Assignment	*	*
Bid Request		*
Daily Assignments		*
Daily Statistics		*
Delivery Alert		*
Delivery Bid	*	
Delivery Bid Alert		*
Delivery Confirmation	*	
Delivery Request	*	
Delivery Request Cancellation	*	
Driver Monitoring Indication		*
GPS Location	*	
Improve Note		*
Manual Assignment		*
Time-out Alert		
Pick-up Alert		*
Pick-up Confirmation	*	
Ranking Decrease		*
Ranking Increase		*

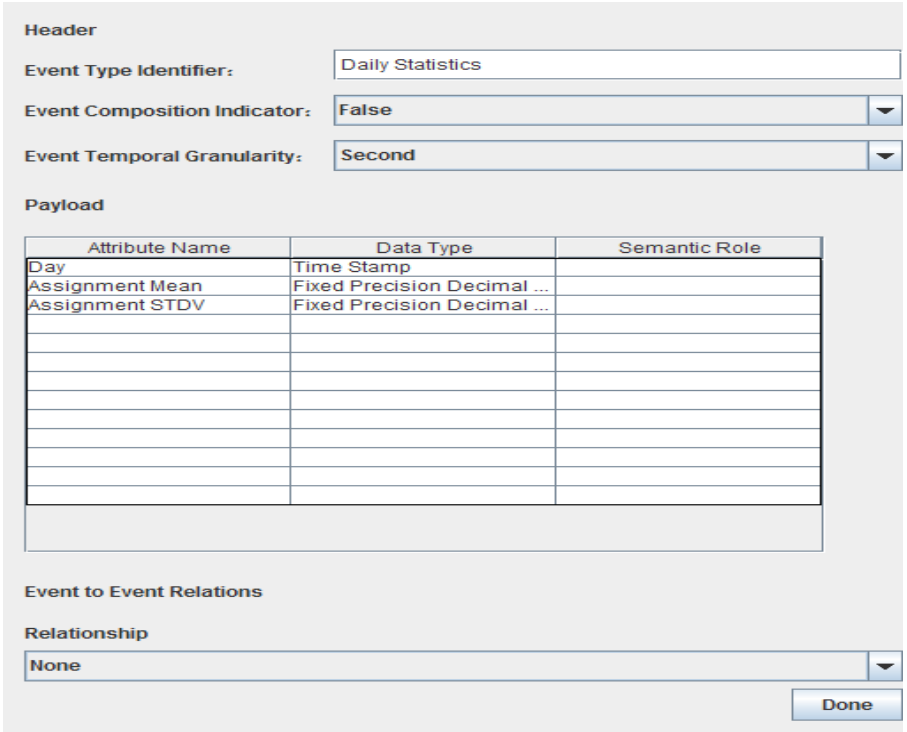
Vehicle Location *

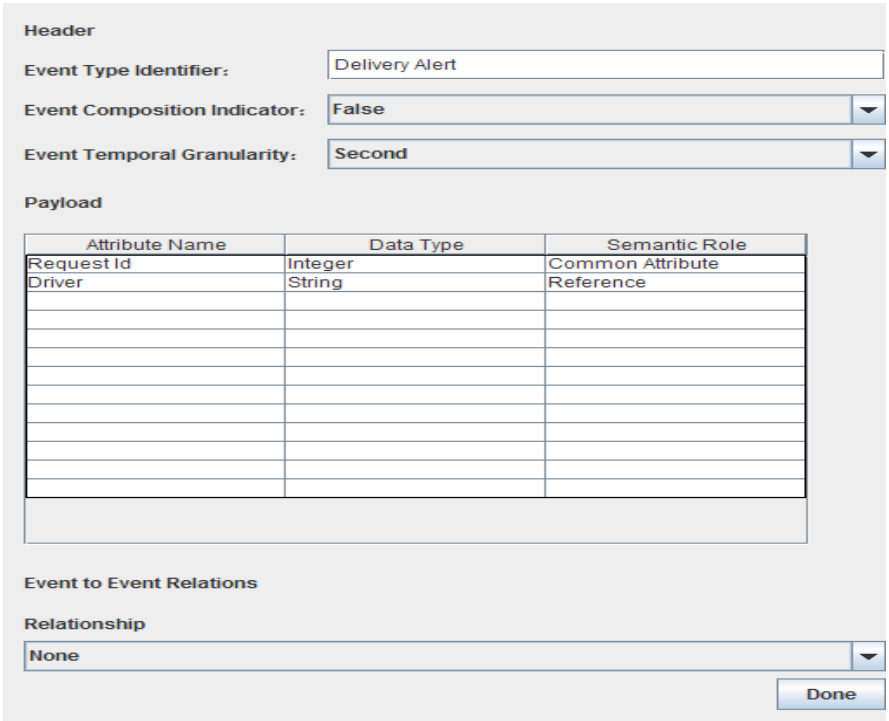
The following figures show the definitions of these events.

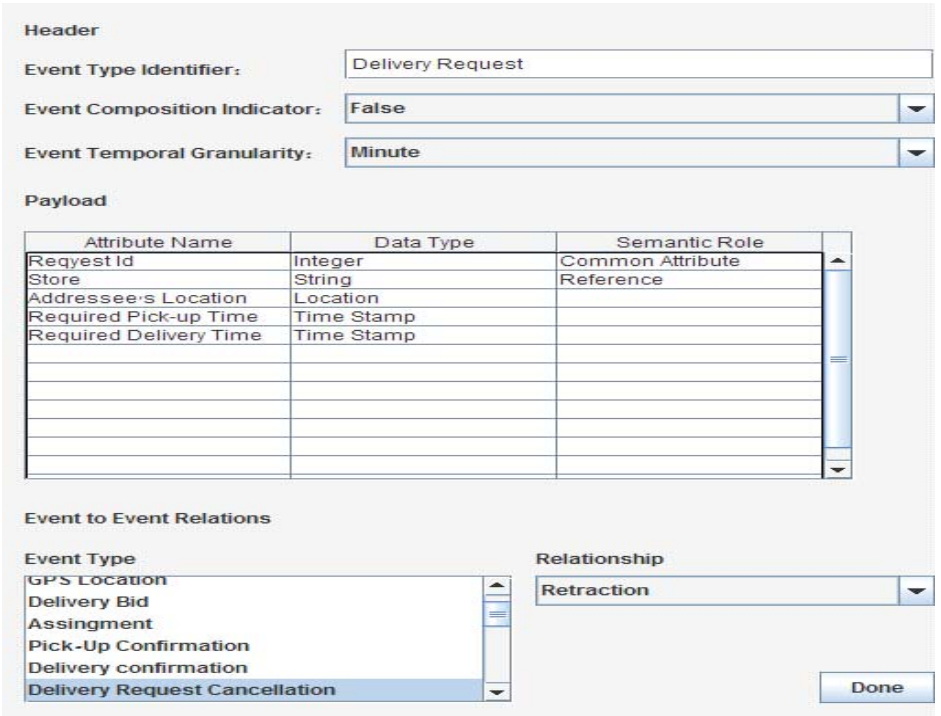


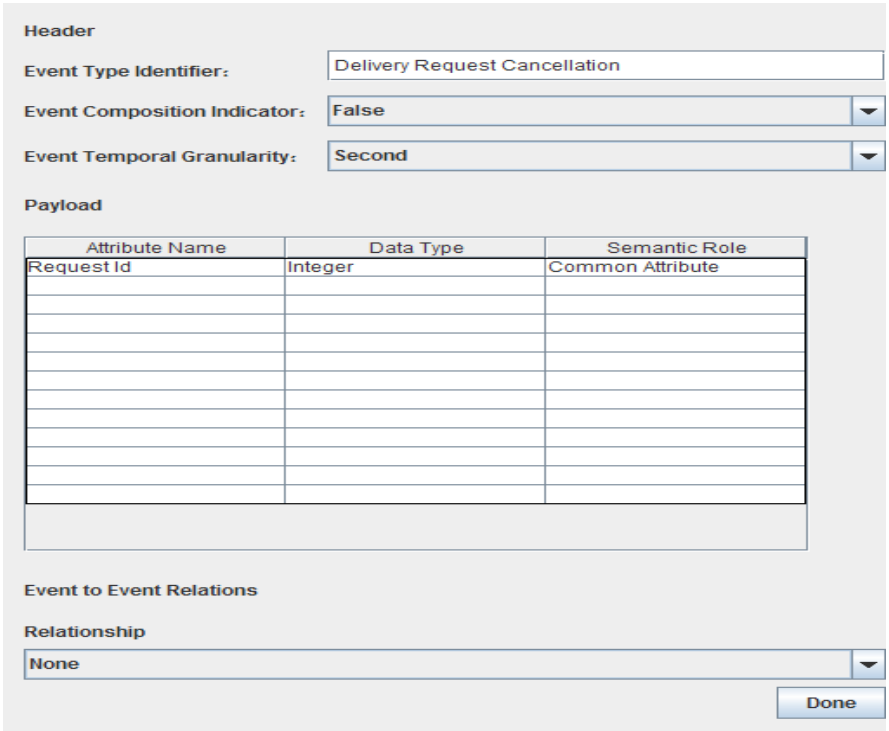


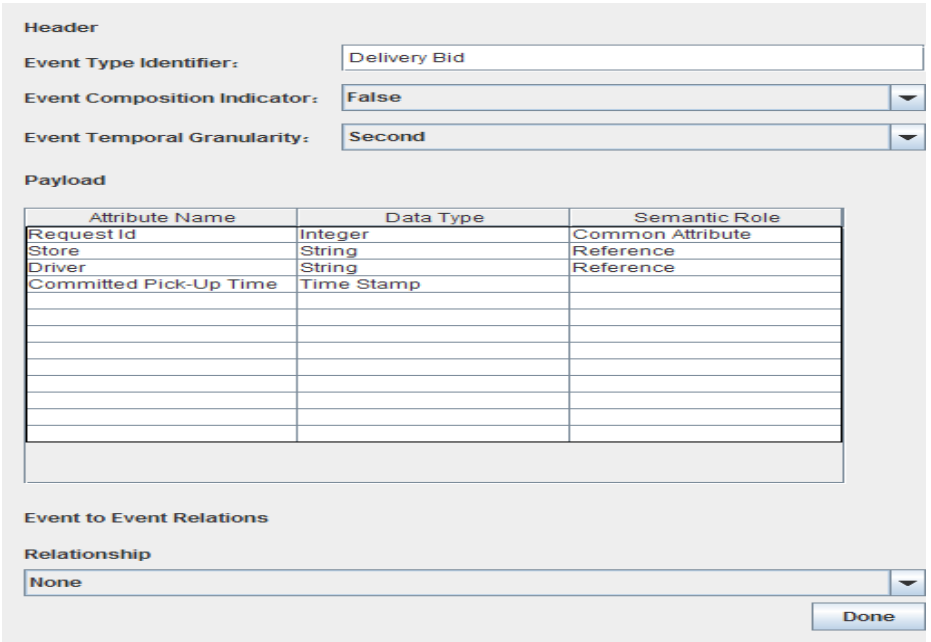


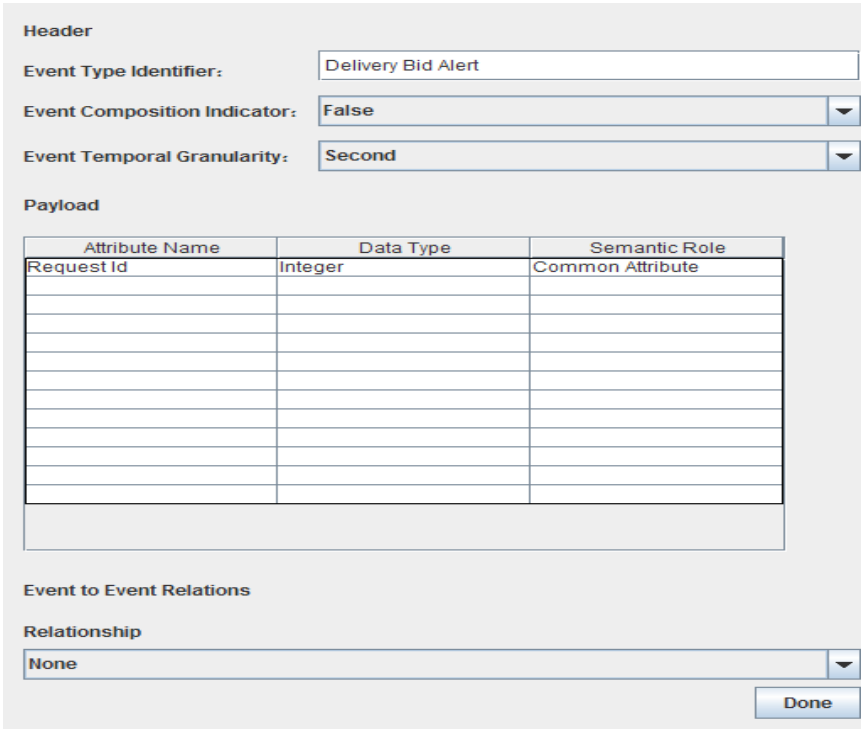


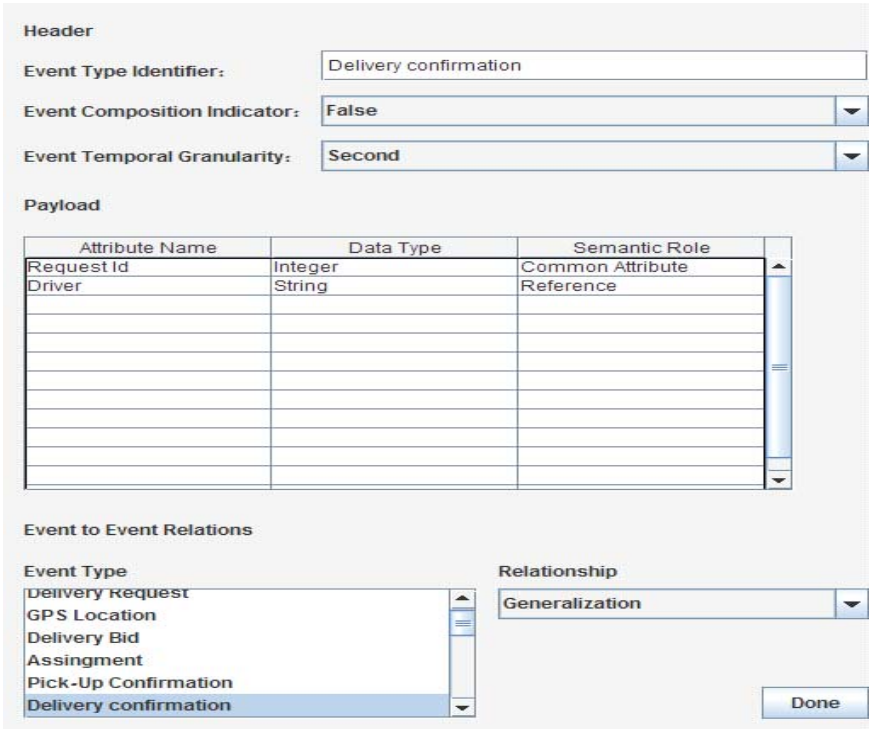


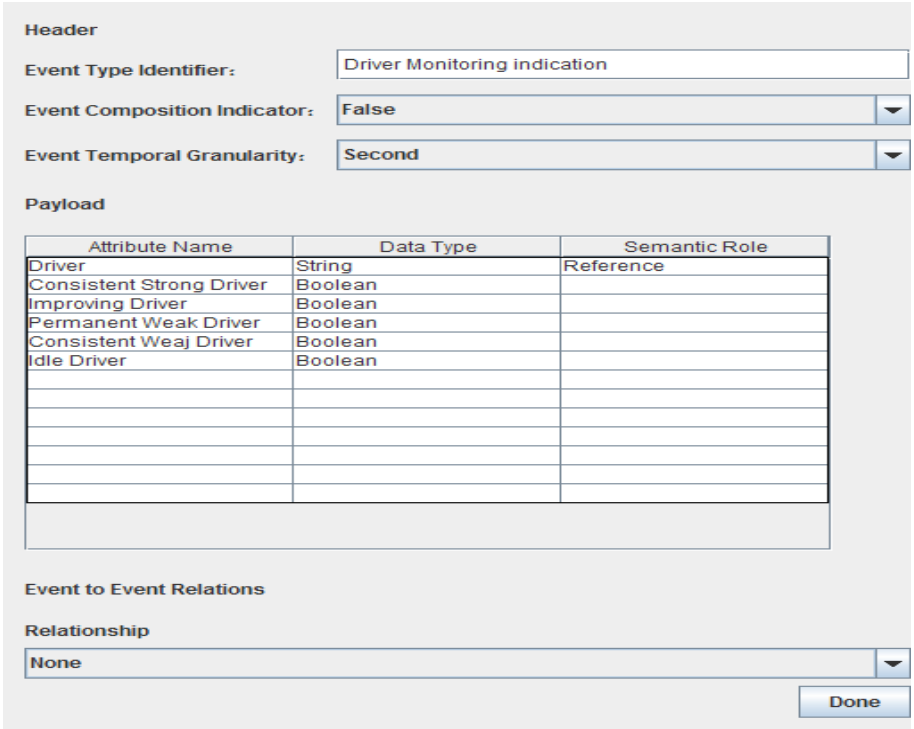


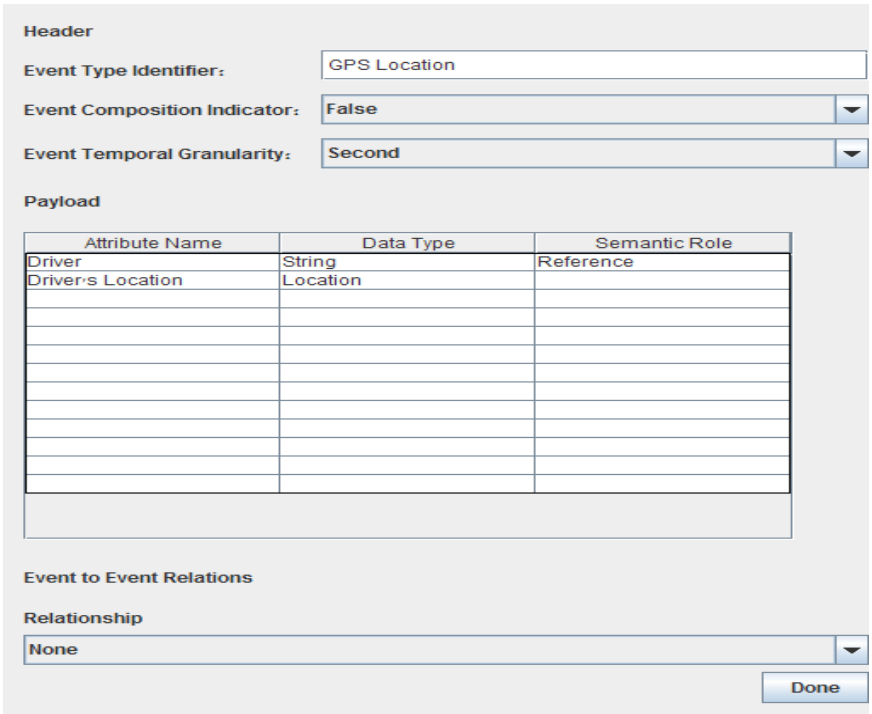


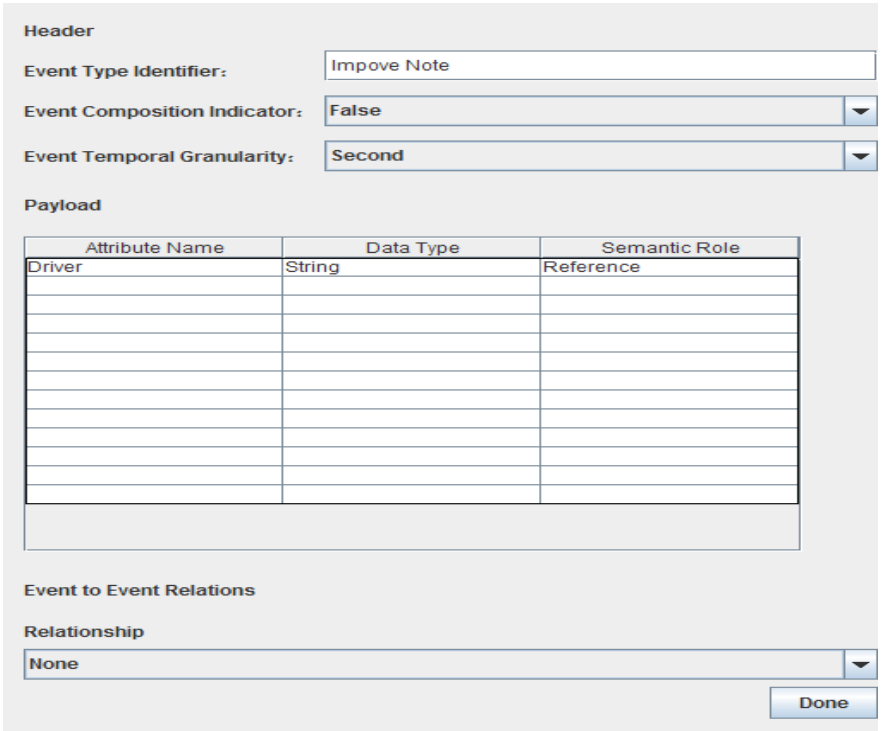


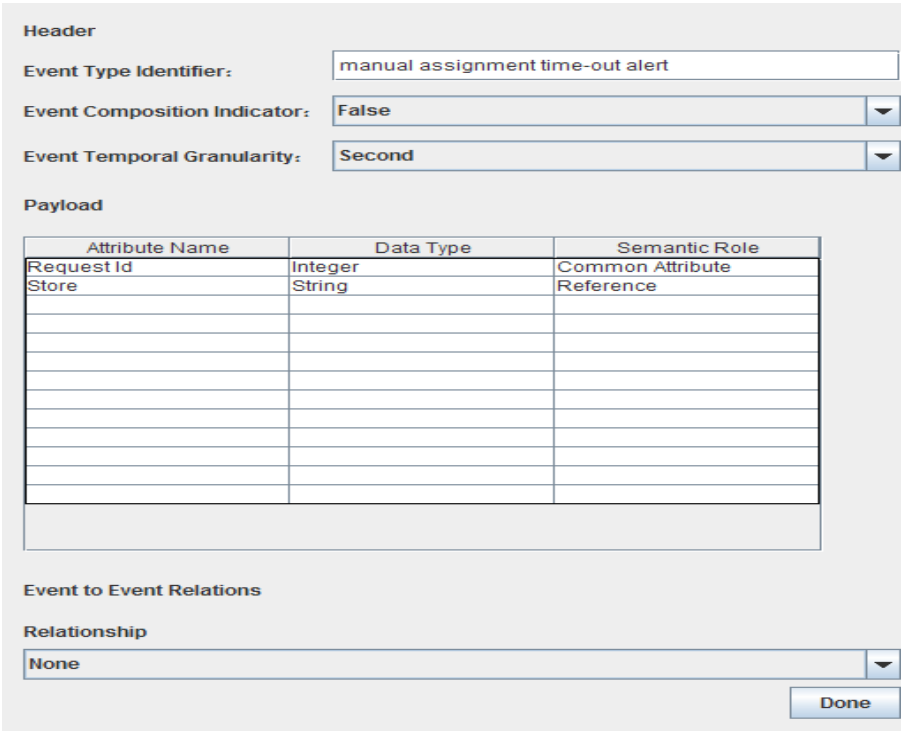


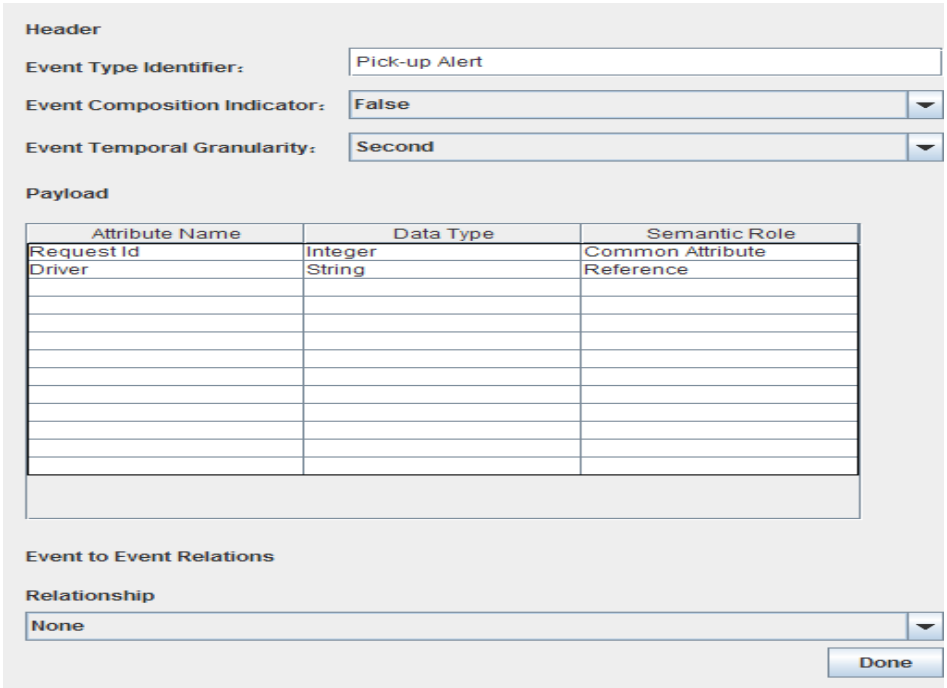


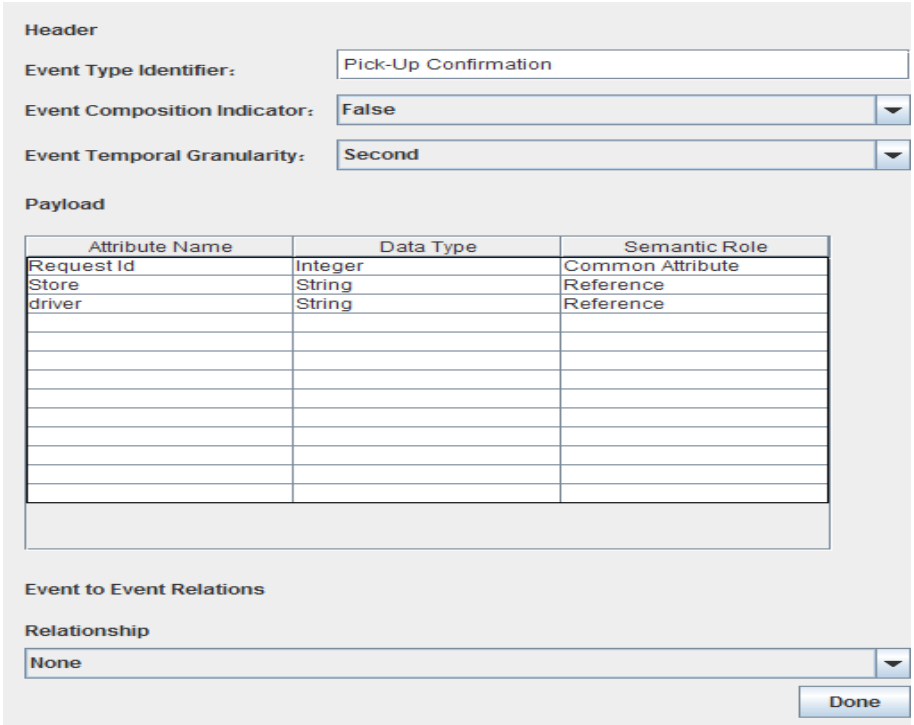


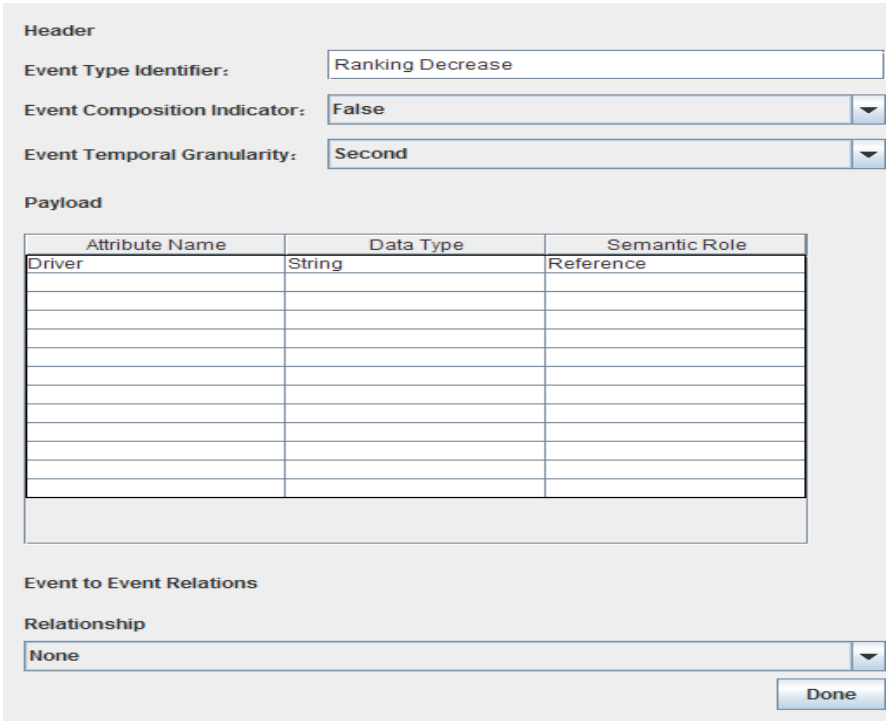


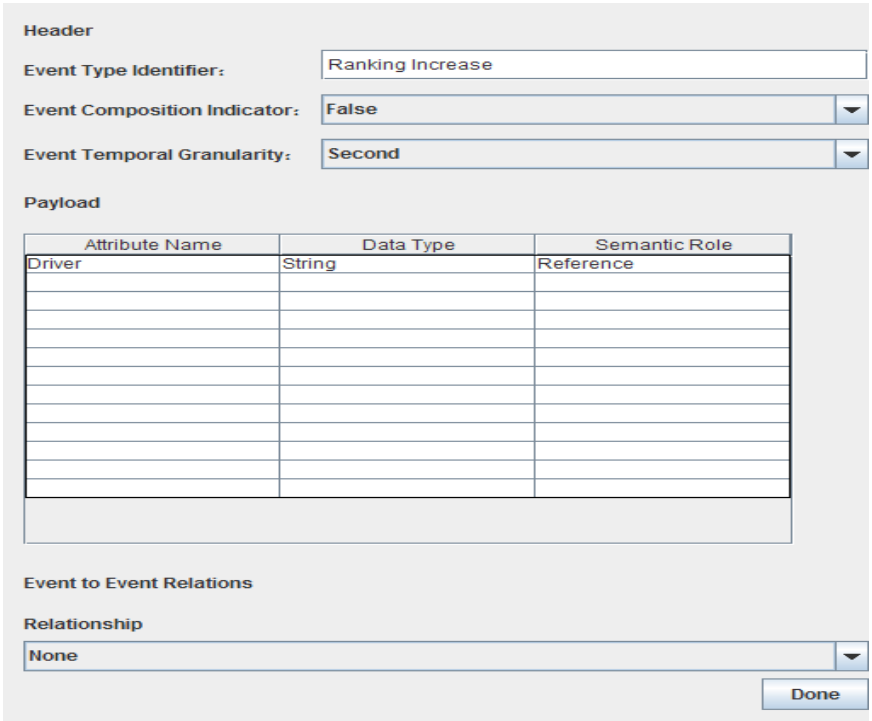


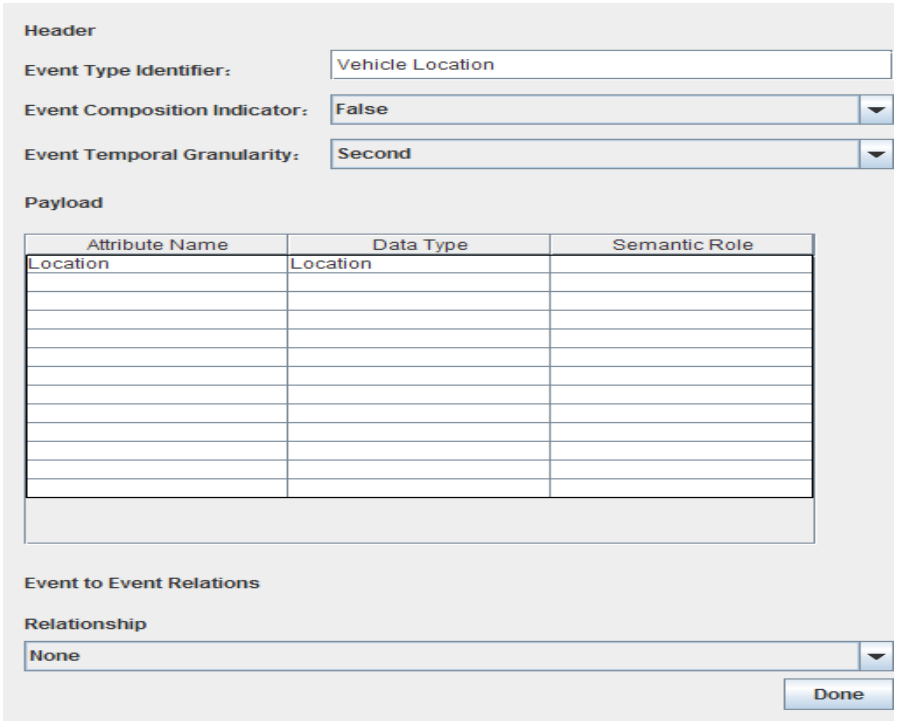












B.3 Contexts

```
Context              Context Type     Context sub-       Composition
Identifier                            type               operator
Driver context       Segmentation

Partition Parameter     Parameter Value
Attribute               Driver
```

This context partitions the relevant events according to driver

```
Context              Context Type     Context sub-       Composition
Identifier                            type               operator
Request context      Segmentation

Partition Parameter     Parameter Value
Attribute               Request Id
```

This context partitions the relevant events according to driver

Context Identifier	Context Type	Context sub-type	Composition operator
Monthly	Temporal	Sliding Fixed	

Partition Parameter	Parameter Value
Interval Period	Month
Interval Duration	Month
Overlapping	False

This context partitions the relevant events according to the month.

Context Identifier	Context Type	Context sub-type	Composition operator
Driver monthly	Composite		Multi-dimensional Segment = Driver, Temporal = Month

This context partitions the relevant events according to the combination of month and driver

Context Identifier	Context Type	Context sub-type	Composition operator
Bid Interval	Temporal	Event Interval	

Partition Parameter	Parameter Value
Initiator	Bid Request
Terminator	+ 2 Minutes
Matched-by	Request Id
Synonym policy	ignore

This context opens a partition each time that a bid request is issued; this is a temporal context that ends after 2 minutes. Notes that since Request Id should have a unique Request-Id any synonym should be ignored.

Context Identifier	Context Type	Context sub-type	Composition operator
Response Interval	Temporal	Event Interval	

Partition Parameter	Parameter Value
Initiator	Bid Interval Termination
Terminator	+ 1 Minutes
Matched-by	Request Id
Synonym policy	ignore

This context opens a partition each time that a bid request is terminated; this is a temporal context that ends after 1 minutes. Notes that since Request Id should have a unique Request-Id any synonym should be ignored.

Context Identifier	Context Type	Context sub-type	Composition operator
Pick up interval	Temporal	Event Interval	

Partition Parameter	Parameter Value
Initiator	Assignment
Terminator	Pick-Up Confirmation
Expiration offset	Assignment. Pick Up + 5 Mins.
Matched-by	Request Id
Synonym policy	Ignore

This context denotes the time interval in which pick up is expected. It is expired when the time-out arrives.

Context Identifier	Context Type	Context sub-type	Composition operator
Delivery interval	Temporal	Event Interval	

Partition Parameter	Parameter Value
Initiator	Assignment
Terminator	Delivery Confirmation
Expiration offset	Assignment. Delivery + 5 Mins.
Matched-by	Request Id
Synonym policy	ignore

This context denotes the time interval in which delivery is expected. It is expired when the time-out arrives.

Context Identifier	Context Type	Context sub-type	Composition operator
Driver Evaluation	Temporal	Sliding event	

Partition Parameter	Parameter Value
Event Type	Delivery Confirmation
Event Count	20
Matched-by	Driver

This context partitions the Delivery Confirmation for each driver in groups of 20 events.

Context Identifier	Context Type	Context sub-type	Composition operator
Monthly	Temporal	Fixed Sliding	

Partition Parameter	Parameter Value
	Month
Interval Period	
Interval Duration	Month
Overlapping	False

This is a long-term context that lasts for a month,

Context Identifier By Driver	Context Type Segmentation	Context sub-type	Composition operator
Partition Parameter Attribute	Parameter Value Driver		

Context Identifier Monthly Driver	Context Type Composite	Context sub-type	Composition operator ∩
Context Monthly By Driver			

B.4 Event Producers

Definition element of Producer type Store

Producer Category	Definition Element Type	Producer Identifier	Output Terminal	Event Type	Targets
Human	Producer Class	Store	Send Delivery Request	Delivery Request	Delivery Request Channel
			Report Manual Assignment	Manual Assignment	Assignment Channel
			Confirm Pick-up	Pick-up Confirmation	Pick-up confirmation channel
			Request Cancellation	Delivery Request cancellation	Delivery Cancellation channel

Note that in our example each of the stores' output terminals is wired to a separate channel. We have chosen to use separate channels for each event type, with the exception of the Assignment channel which is used for both Manual and Automatic assignments,

Definition element of Producer type GPS Sensor

Producer Category	Definition Element Type	Producer Identifier	Output Terminal	Event Type	Targets
Sensor	Abstract Type	GPS Sensor			

Definition element of Producer type Vehicle

Producer Category	Definition Element Type	Producer Identifier	Output Terminal	Event Type	Targets
GPS Sensor	Producer Class	Vehicle	Report Location	GPS Location	GPS Channel

This producer belongs to the category of GPS sensor, which has been defined as an abstract producer type in the previous definition element.

Definition element of Producer type Driver

Producer Category	Definition Element Type	Producer Identifier	Output Terminal	Event Type	Targets
Human (via handheld device)	Producer Class	Driver	Bid for Delivery	Delivery Bid	Delivery Bid Channel
			Confirm Delivery	Delivery Confirmation	Delivery Confirmation Channel

Delivery confirmation is produced by the Driver's handheld device, but requires signature of the delivery recipient

B.5 Event Consumers

Consumer Category	Definition Element Type	Consumer Identifier	Input Terminal	Event Type
Human	Consumer Class	Driver	Bids	Bid Request
			Assignments	Assignment

Consumer Category	Definition Element Type	Consumer Identifier	Input Terminal	Event Type
Human	Consumer Class	Store	Bids	Delivery Bid
			Alerts	No Bidder
			Alerts	No Assignment
			Alerts	Delivery

				alert

Consumer Category	Definition Element Type	Consumer Identifier	Input Terminal	Event Type
Human	Consumer Instance	Manager	Alerts	No Bidder
			Alerts	No Assignment
			Alerts	Pick-up alert
			Alerts	Delivery alert

B.6 Event Processing Agents

EPA Identifier	EPA type	Input Terminals	Output Terminals
Bid Request Creator	Enrich	Bid Request Channel, Store Reference Global State	Delivery Request channel
Location Service	Translate	GPS channel	Delivery Request Channel
No Bidders	Pattern	Delivery Bid Channel	No Bidder Channel
Automatic or manual Matching	Filter	Delivery Bid Channel	Store, Automatic matching EPA
Automatic Matching	Pattern	Automatic matching EPA	Assignment Channel
Assignment not done	Pattern	Assignment channel	Assignment Timeout channel
Pick Up Alert	Pattern	Pick-up confirmation channel	Alerts channel
Delivery alert	Pattern	Delivery Confirmation Channel	Alerts Channel
Ranking	Pattern	Ranking	Ranking

Increase Ranking Decrease	Pattern	Input Channel Ranking Input Channel	Output Channel Ranking Output Channel
Improving Note	Pattern	Ranking Output Channel	Improvement Note Channel
Daily Assignments calculator	Aggregate	Assignment Channel	Daily Assignment Channel, Input Evaluation Input Channel
Daily Statistics Creator	Aggregate	Daily Assignment Channel	Daily Channel
Permanent Weak Driver	Pattern	Daily Channel	Evaluation Input Channel
Idle Driver	Pattern	Evaluation Input Channel	Evaluation Output Channel
Consistent Strong Driver	Pattern	Evaluation Input Channel	Evaluation Output Channel
Consistent Weak Driver	Pattern	Evaluation Input Channel	Evaluation Output Channel
Improving Driver	Pattern	Evaluation Input Channel	Evaluation Output Channel

B.7 Global States

Global State Identifier	Global State Type	Meta-Data	Input Terminals	Output Terminals
Location-Reference	Reference Data	Geospatial DB schema		Location Service EPA
Driver-Variable	Global Variable	Driver Ranking	Ranking EPA	Delivery Request Channel
Store-Reference	Reference Data	Store Minimal Ranking, Store		Delivery Request Channel

location

B.8 Channels

Fixed Channels

Channel Identifier	Routing Scheme	Routing Rules/ parameter	Event Type	Input Terminal	Output Terminal
Bid Request	Fixed		Bid Request	Store	Bid Request Creator
GPS Channel	Fixed		GPS Location	GPS sensor	Location Service EPA
Automatic matching channel	Fixed		Delivery Bid	Manual or automatic matching EPA	Automatic matching EPA
Manual matching channel	Fixed		Delivery Bid	Manual or automatic matching EPA	Store
Assignment timeout channel	Fixed		Assignment not done	Assignment not done EPA	System Manager
Pick-up Confirmation Channel	Fixed		Pick-up Confirmation	Store	Pick-up Alert EPA
Delivery Confirmation Channel	Fixed		Delivery Confirmation	Driver	Delivery Alert EPA
No Bidder Channel	Fixed		No Bidders Alert	No Bidders EPA	System Manager
Alerts channel	Fixed		All alert events	All alert EPAs	Store, Manager
Ranking Input channel	Fixed		Delivery Alert	Delivery Alert EPA	All Ranking EPAs
Ranking	Fixed		Ranking	All	Monitorin

Output channel			Events	Ranking EPAs	g System, Driver's Guild
Daily Assignment Channel	Fixed		Assignment	Daily Assignment EPA	Daily Statistics EPA
Evaluation Input Channel	Fixed		Daily assignment EPA, Daily Statistics EPA	Store, Assign EPA	All evaluation EPAs
Evaluation Output Channels	Fixed		All evaluation events	All Evaluation EPAs	Driver's Guild

Explicit Channels

Channel Identifier	Routing Scheme	Routing Rules/ parameter	Event Type	Input Terminal	Output Terminal
Delivery Request Channel	Content Based	Driver. Area = Store Location and Driver. Ranking ≥ Store. Minimal - Ranking	Delivery Request Driver Location	Build Request Creator EPA Location Service EPA	Drivers
Delivery Bid Channel	Itinerary based	Store	Delivery Bid	Driver	Manual or automatic matching EPA No Bidders EPA
Assignment Channel	Itinerary Based	Driver	Assignment	Store, Automatic matching EPA	Driver, Assignment not done channel, Daily

					Statistics EPA, Daily Assignment
Improvement note channel	Itinerary Based	Driver	Improvement Note	Improve EPA	Driver

B.9 Event Pattern Detection Agents

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
Automatic Matching process	Any	Bid Interval	Delivery Bid	Delivery Bid. Committed Pick Up time < Bid Request. Required Pick Up Time + 5 minutes	Synonyms = first; cardinality = single

Pattern annotation: This pattern generates a matching set that contains a single event that contains the first Delivery Bid where its committed pick up time matches the required pick up time.

Comment: This pattern generates a single matching set within the context.

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
No bidders	Absence	Bid Interval	Delivery Bid	Delivery Bid. Committed Pick Up time < Bid Request. Required Pick Up Time + 5 minutes	

Annotation: This is time-out detection, indicating that no bidders that satisfy the Pick Up assertion have been detected during the bid request interval context.

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
Assignment not done	Absence	Response Interval	Manual Assignment		

Annotation: This is time-out detection, indicating that the manual decision has not been performed on time.

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
Pick up alert	**Absence**	**Pick up Interval**	**Pick up Confirmation**		

Annotation: This is time-out detection, indicating that pick up was not done on time.

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
Delivery alert	**Absence**	**Delivery Interval**	**Delivery Confirmation**		

Annotation: This is time-out detection, indicating that the delivery was not done on time.

Pattern Name	**Pattern Type**	**Context**	**Participants Set**	**Assertions**	**Policies**
Ranking Increase	**Absence**	**Driver Evaluation**	**Delivery Alert**		

Annotation: this detects that a driver did not have any delivery alerts within a temporal sliding event interval of 20 deliveries.

Pattern Name	**Pattern Type**	**Context**	**Participants Set**	**Assertions**	**Policies**
Ranking Decrease	**Count**	**Driver Evaluation**	**Delivery Alert**	**> 5**	

Annotation: this detects that a driver did had more than 5 delivery alerts within a temporal sliding event interval of 20 deliveries

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
Improve note	Sequence	forever	Ranking Decrease, Ranking Increase		Synonyms = override

Annotation: this pattern detects that a driver had ranking increase after he got ranking decrease. The override guarantees that there will always be a single one of each of these events in the relevant events set. Forever is a universal context that is always true.

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
Permanent Weak Driver	Always	Monthly Driver	Daily Assignment	Assignments Number < 5	

Annotation: this pattern detects that a driver had less than five assignments in each of his working days during the month.

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
Idle Driver	Sometimes	Monthly Driver	Daily Assignment	Assignments Number = 0	

Annotation: this pattern detects that a driver had at least one working day during the month without any assignment,

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
Consistent Weak Driver	Always	Monthly Driver	Daily Assignment, Daily statistics	Assignments Number < Daily Mean – 2 * Daily STDV Matched-by Day	

Annotation: this pattern detects that a driver's number of assignments is consistently low. Note that this assertion employs two event types, both of them derived, Daily Assignment, and Daily Statistics.

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
Consistent Strong Driver	Always	Monthly Driver	Daily Assignment, Daily statistics	Assignment Number > Daily Mean + 2 * Daily STDV Matched-by Day	

Annotation: this pattern detects that a driver's number of assignments is consistently low. Note that this assertion employs two event types, both of them derived, Daily Assignment, and Daily Statistics.

Pattern Name	Pattern Type	Context	Participants Set	Assertions	Policies
Improving Driver	Non Decreasing	Monthly Driver	Daily Assignment	Assignments Number	

Annotation: this pattern detects that the number of assignments is equal or increasing over the month.