4

Event management and composition framework

This chapter presents the proposed framework for event management and composition in order to build adaptive systems. It is defined as a middleware framework which is suitable for designing and implementing specialized event-based systems.

The framework is component-based due component-based approach allows that parts of the service can be reconfigured at runtime. The instantiation of the framework produces a particular event-based system that ensures event-based communication. The framework is defined around flexible and general models, each of which focuses on a different domain of concern in the design.

The chapter is organized as follows: section 4.1 presents the general architecture of the proposed framework and describes its components and the interaction among them. Section 4.2 presents the models for event management and composition that the framework implements. Section 4.3 presents the functionality of an event service. Finally, section 4.4 concludes the chapter.

4.1 General architecture

Figure 4.1 presents the general component-based architecture of the framework that we propose. The framework implements components for the event observation, filtering and notification. The framework defines configuration components to personalize (i) event types, (ii) composition operators and their associated semantics, and (iii) the composition algorithm. Thus, the framework is instantiated and configured at run time with the requirements of a particular application.

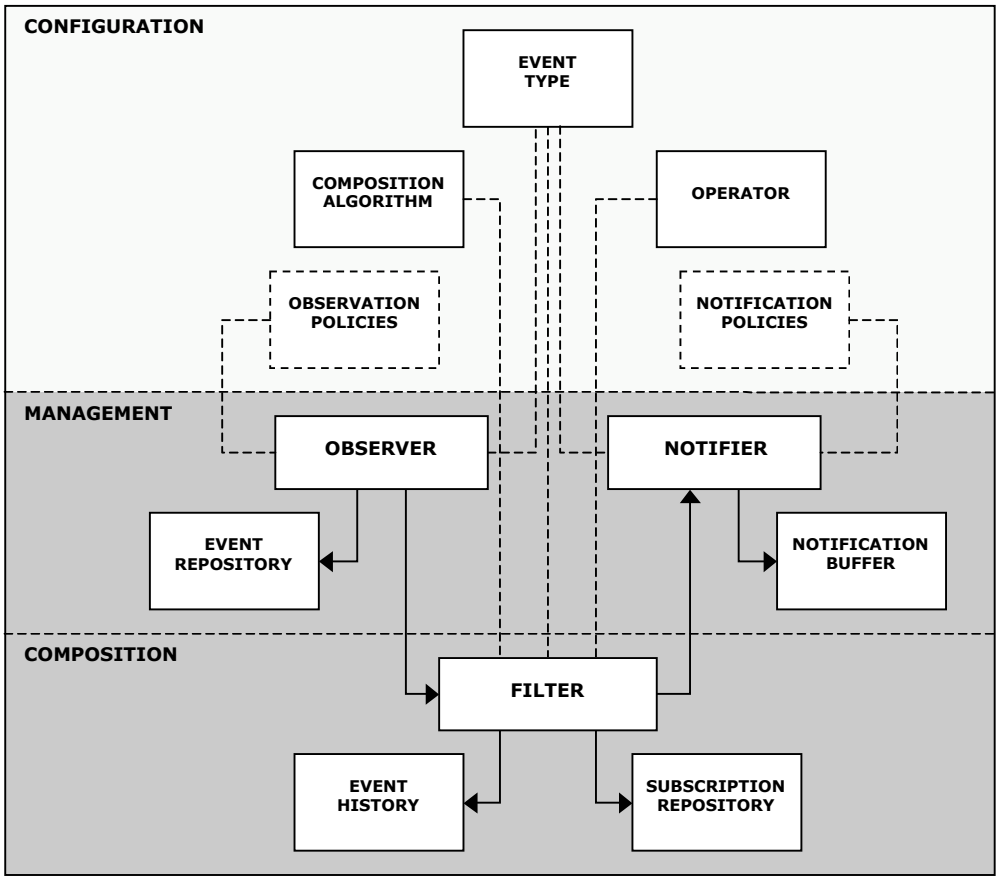


Figure 4.1 Framework architecture

The components for event management and composition that the framework implements are described in the following sections:

4.1.1 Observer

The observer detects events as *operations* of objects of interest. We consider that objects of interest are *information objects* that are located at the producer site, optionally in an *object repository*. Information objects can be persistent (e.g. documents) or transient (e.g. measured values). Objects are described by attributes. Each attribute has a domain that defines the possible attribute values. Objects have a state, which is given by the value of its attributes and can also be composed of other objects (hierarchy of information objects).

Changes of these objects (creation, update, deletion) are called operations and are induced by an *invoker*. An operation may be invoked directly through some apparatus associated with the object, or it may be invoked indirectly as a result of executing some program or software tool.

Therefore, the operation detection is an active task of the observer (performed according to a schedule) if the producer does not inform it about operations. We consider that any operation is an event. Then, the observer creates an event type to each operation and finally, the observer reports events as event messages to the *filter*. Then, the observer implements event and time models for the event detection and production.

Not all event producers implement an observer, therefore an event service that covers these types of producers implement an observer as a wrapper for each producer. Alternatively, the observer can be moved to the producer site and perform its tasks as an agent of the service there. How events are observed is described by the observation policies associated with the observation model.

4.1.2 Filter

Detected primitive events are stored in the *event history*. The event history consists of all occurrences of the defined events, including components of composite events. The event history begins when the fist event is detected and is ordered by the event timestamps.

The filter has knowledge of the consumer subscriptions (*subscription repository*) and compares the detected events with the subscriptions. The detection of composite events is supported by the composition algorithm. The composition of a composite event builds upon the detection of its component events. If a detected event contributes toward the composition of some composite event a new composition state is derived and the old one is deleted. A final state indicates the completed composition of a composite event. If an event contributes toward the composition of multiple composite events, rule priorities can be used which determine the order or their evaluation.

If a subscription and an event match the filter creates an event message and delivers it to the *notifier*. Therefore, the filter implements the event model to create event messages.

4.1.3 Notifier

The notifier checks the schedule part of the subscription. If immediate delivery is demanded, the event message is converted according to the format specified by the consumer and delivered. Otherwise, it is buffered until the notifications become due. How events are notified is described by the notification policies associated with the notification model.

4.2 Models for event management and composition

In this section we look at the framework models that represent event management and composition strategies that can be implemented by the components of an event system. The design models are as follows: the data handled in the event-based framework is described by an *event model*. The *subscription model* describes how consumers express interest in occurrences of events. Communication issues of the framework are part of the *observation* and *notification models*. The *time model* is concerned with the causal and temporal relations among events.

4.2.1 Event model

The ***event model*** describes the representation and characteristics of the events. An event can be uniquely characterized by the identity of the object of interest involved in the event, the identity of the operation, the identity of the invoker, and the time of occurrence of the event.

The event model proposed in this work considers primitive and composite events (figure 4.2). We further distinguish two categories of primitive events: *time events* and *content events*. Time events represent the passage of time – the events refer to certain points in time. They do not consider a certain object but rather a logical abstraction, e.g., of a global clock. Time events may involve clock times, dates, and time intervals.

Content events involve changes of non-temporal objects, such as sensors. We additionally distinguish *active* and *passive content events*. Active content events are state transitions of an information object at a particular time; they are observer independent. The state of a particular information object is described by the attribute values of the object. A state transition occurs if at least one of the attributes values is changed.

Passive content events model the fact that for a given time interval an object did not change. The subscription *"notify if the sensor did not send data for more than thirty minutes"* refers to passive events. Passive events are content events as well as composite events and they have to be observed.

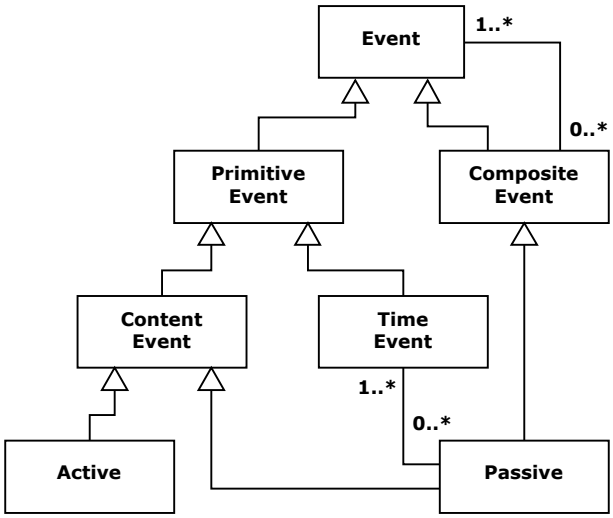


Figure 4.2 Event model

Events may be reported by means of event messages. In a message, an event may be described by a collection of (*attribute; value*) pairs, such as the three pairs in the following example. The event message reports the crossing of a temperature threshold at a sensor:

eroom : event(temperature = 35 °C,
room = 150,
timestamp = 10 : 00 : 00)

A consequence of this model is that there is a one-to-one correspondence between operation invocations and event occurrences. However, not every event will result in an observation of the event, and not every observation will result in a notification being communicated to some consumer. An event is simply a phenomenon that occurs regardless of whether or not it is observed.

Another consequence of this model is that events corresponding to the initiation of operation invocations are not associated with the object of interest. Such events are associated instead with the invoker.

4.2.2 Subscription model

The ***subscription model*** describes consumers interest in occurrences of events. Consumers describe the events in which they are interested as subscriptions. Hence, detected events are filtered according to the consumer subscriptions.

Subscriptions consist of two parts: the description of the events (*query subscription*) and additional information about the consumer and the conditions for notification (*parameter subscription*). The filter evaluates the query subscription periodically against the occurred events.

The model considers primitive and composite subscriptions (figure 4.3). Primitive subscriptions consist of predicates on (*attribute; value*) pairs. An example subscription is:

$$s_{temp} = subscription(temperature > 30° C)$$

Composite subscriptions may be unary or binary subscriptions referring to unary or binary event operators respectively.

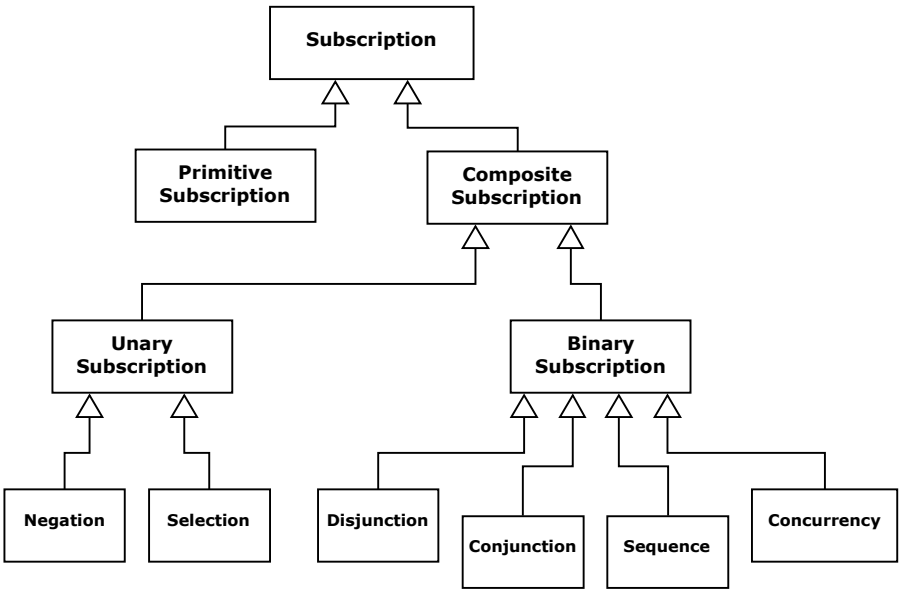


Figure 4.3 Subscription model

4.2.3 Observation and notification model

The ***observation model*** defines the way event occurrences are observed for the purpose of notifying interested consumers. The model defines how events are requested and observed.

The ***notification model*** is concerned with the way the events are delivered to the subscribers. A notification is a message reporting about events. Consumers are notified according to the schedules given in their parameter subscriptions.

The observation and notification models adopt *communication strategies* to observe and notify events.

Communication strategies

Events have to be observed at producer sites and notifications are sent to consumers. For the external communication of event services, two connections are considered: *producer-service communication* and *service-consumer communication*. We describe the producer-service communication whose principle can be applied for the service-consumer communication. We distinguish *passive/active* observations that use a communication mode *synchronous/asynchronous* to the event occurrence. Table 4.1 shows the combinations of the observations and communication modes.

Observation	Mode	Initiator
passive	synchronous	invoker or information object
passive	asynchronous	separate observer on producer site
active	synchronous	observer (triggered by event occurrence)
active	asynchronous	observer component of the service

Table 4.1 Communication strategies

Passive synchronous observation can be initiated by the invoker or the information object. The invoker changes the information object and synchronously announces the state change to the observer (figure 4.4(a)).

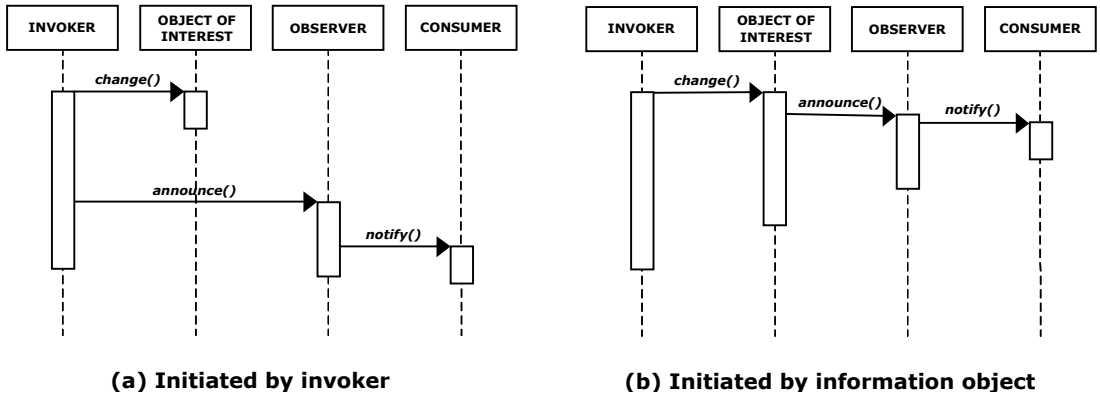


Figure 4.4 Passive synchronous observation

Passive synchronous observation can also be initiated by the information object. The object itself announces its state change to an observer (figure 4.4(b)). Both forms are facets of a single strategy, because both object and invoker reside on the producer site. For *passive asynchronous observation*, the producer employs an observer of events on its site, which asynchronously reports the events to the service.

In *active synchronous observation* the service observer is triggered by the event occurrence. This triggering is already a synchronous (passive) observation in itself. Then, the observer performs a more detailed observation than the triggering can provide. In *active asynchronous observation* the invoker changes the information object, the observer verifies the object state on a regular or irregular basis. The state change is detected by the observer after a certain delay (figure 4.5).

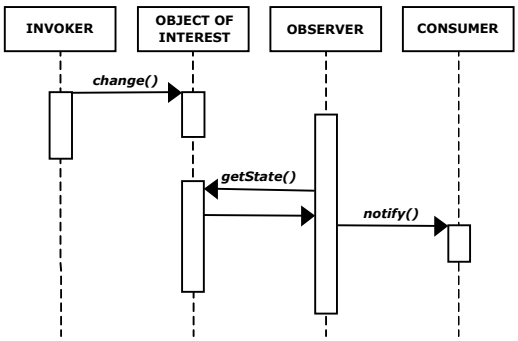


Figure 4.5 Active asynchronous observation initiated by observer

The communication strategies *passive observation* and *active observation* are often referred to as *push* and *pull*. In the *push model*, the producer informs the event service about events. In the *pull model*, the event service observes events at producer sites.

4.2.4 Time model

The ***time model*** is concerned with the causal and temporal relations between events. The most elemental concept concerning to time is the *time line*. This concept is particularly significant because events occur in a time line. The characteristics of this line make it possible to adopt particular algorithms of scheduling, but also to specify temporal relations between these events.

A *granularity* is a partition of a time line in called *convex subsets grains*. The partitions in weeks, month and years correspond to granularities. The granularity obtained by dividing a time line is called *minimal granularity*.

The set of granularities on the same time line is structured in a hierarchy according to the relation of a partial order *finer than* which makes it possible to say for example than the granularity `second` and more fine that `hours`. The interest of this relation is to allow conversions between grains belonging to various granularities. Two types of functions can be defined:

- *approximation* makes it possible to approximate a grain of one granularity G_1 through G_2 grain which contains it (*zoom in*).
- *expansion* makes it possible to associate a set of G_1 grains to each grain of a granularity G_2 (*zoom out*).

According to granularities concerned, these functions are more or less complex. By the granularity concept, a set of types is identified which intervene directly in the definition of events:

- An *instant* is a point in a time line which can be represented by an integer when a discrete time representation is adopted.

- A *duration* is a number of grains used like measure of the distance between two instants to allow the expression of displacements in time compared to a given instant. In general, it is characterized by a positive integer and by a granularity, e.g.: 8 seconds.

- An *interval* is represented by two instants, or by an instant and a duration. Considering that the lower and higher limits of an interval have the same granularity, the interval can be represented by a granularity and two positive integers.

In this work we consider an interval-based semantics, therefore events occur over a time interval and are denoted by $E[t_1, t_2]$ where E is the event, t_1 is the start time of the event and t_2 is the end time of the event. The start and the end times of primitive events are assumed to be the same ($t_1 = t_2$).

4.3 Event management and composition service

Figure 4.6 present the functionality of an event management and composition service for a personalized event-based system. Events are caused by invokers that perform actions on the information objects (objects of interest). An observer may learn of events in two ways: either an observer is notified by the invoker, or the observer proceeds according to a time schedule. The observer creates a message reporting the occurrence of the event and forwards it to the filter.

The filter evaluates the subscriptions at the stored events. The possible combinations of events have to be evaluated according event algebra or data mining techniques. Finally, when an event matches a certain subscription, the filter forwards the event message to the notifier. Notifications for the consumers may be buffered, checked for duplicates, merged, and delivered according to the consumer subscriptions.

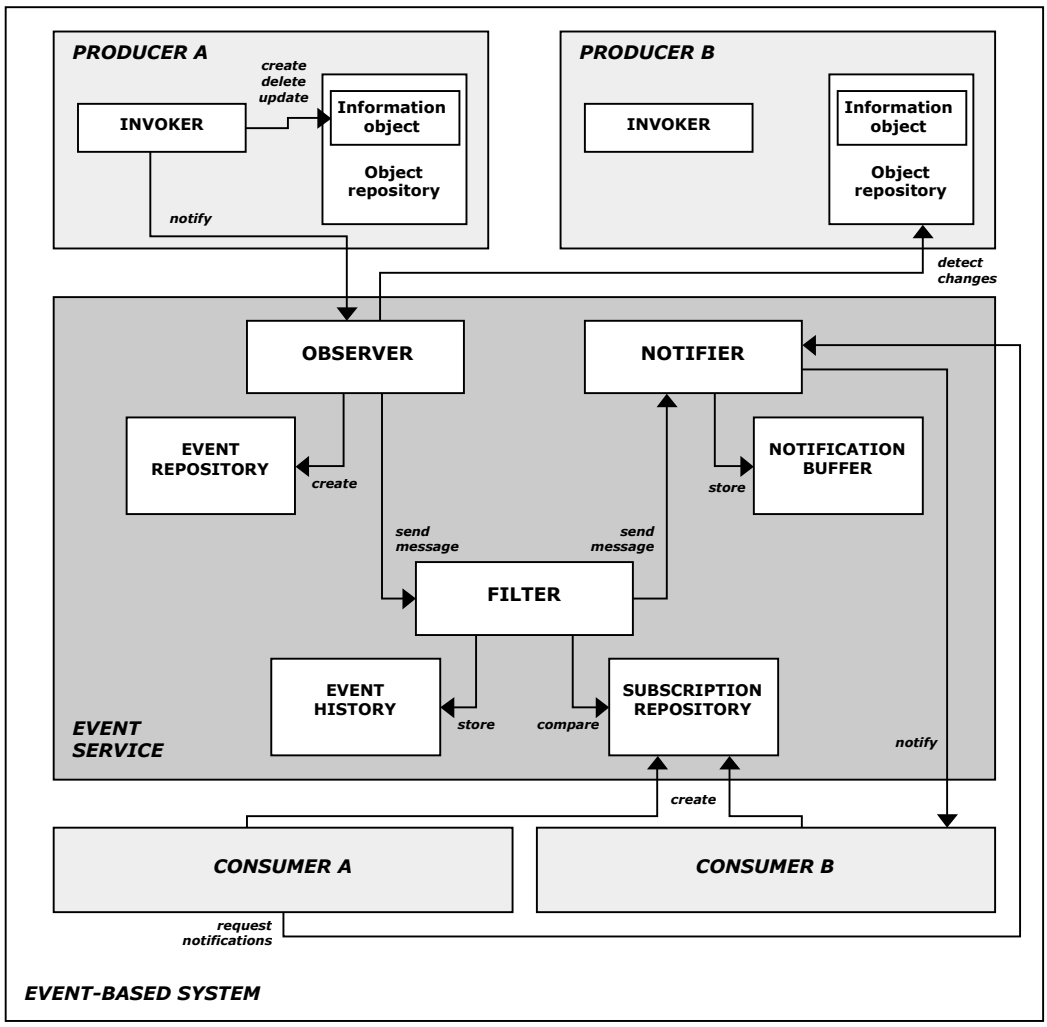


Figure 4.6 Event service functionality

The event processing (figure 4.7) that the event management and composition service implements is described as following: let $t(e)$ be the occurrence time of an event e. The observer is notified by the invoker at time $t^{obs} \geq t(e)$. A composite event is indicated in figure 4.7 by the horizontal bar at the filter, which combines two events. Depending on its schedule, the notifier sends the message to the interested consumers at time t^{not}.

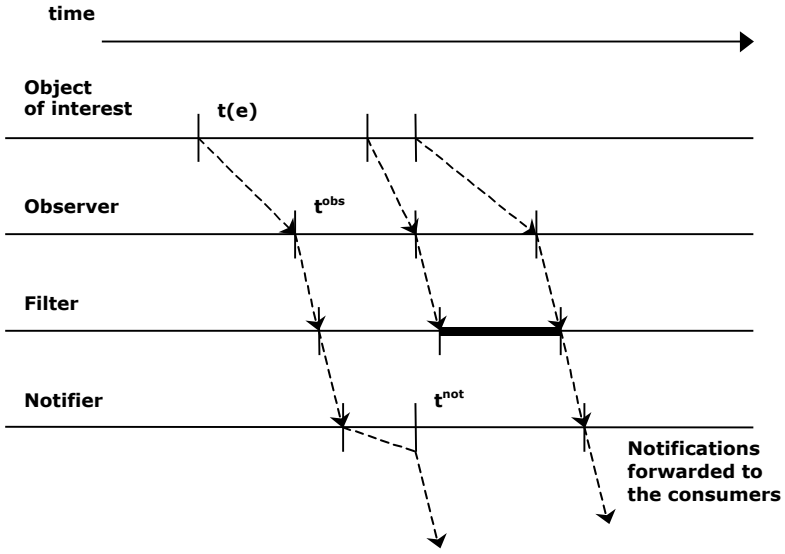


Figure 4.7 Event processing

4.3.1 Primitive event filtering

Three basic filtering approaches can be distinguished: *naive filtering*, *clustered filtering*, and *treebased filtering*. The basic unit for event filtering is the time to filter one event attribute value against one subscription predicate. In the naive approach, all subscriptions are tested successively. In the clustered approach, the predicates are clustered according to the operators used. Then, the most selective predicates are tested first. In that way, similar predicates can be matched faster. For the tree-based approach, the predicates are ordered in a subscription tree according to the attributes they refer to. For each attribute, a selection of predicates has to be tested using binary search. The tree-based filter algorithms show the best performance results. The basic algorithm, which uses sequential search at each node, is shown in algorithm 4.1.

Algorithm 4.1 Tree-based algorithm: sequential tree-search for single matching path

input: subscription tree with attribute levels $a_1 ... a_n$ and branches $b_1 ... b_m$
event-message with (*attribute; value*) pairs
output: list of matching subscriptions or NULL

```
1:  current-node := root                    /* root = a1 */
2:  while (current-node <> leaf-node) do
3:      branch-to-follow := NULL
4:      iterate sequentially through branches i ∈ [1,m]
5:          if branch predicate at bi true for event-message
6:          then branch-to-follow := bi
7:      if (branch-to-follow <> NULL)
8       then follow the branch to the next node: current-node := branch-to-follow _ ai+j
9:      else EXIT
10: output current-node
```

4.3.2 Composite event filtering

Existing composite-event filtering approaches (see section 2.4.1.4) have in common that two steps are necessary to identify composite events: (i) the detection of primitive events, and (ii) the evaluation of the composite events. Thus, the composite event is detected in a separate step after the filtering of primitive events. These two-step methods contain unnecessary filter operations.

Therefore, we adopt a method for the filtering of composite events that integrates the detection of composite events into the detection of primitive events: after the filtering of a primitive event, its contribution to a composite event is tested. In that way, the composite event is detected successively. No additional step is required for the identification of the composite event after the last contributing primitive event has been detected. The identification of the composite event is accelerated and the overall filtering time is reduced.

For illustration, consider the following three example subscriptions:

Consumer A: $E_A = E_1$ (subscription regarding primitive events)
Consumer B: $E_B = (E_1 ; E_2)$ (subscription regarding composite events)
Consumer C: $E_C = E_2$ (subscription regarding primitive events)

Figure 4.8 shows the principle of composite event detection in two steps for these three subscriptions. The triangle represents the primitive subscription pool. The primitive subscription pool represents a structure for indexing and filtering primitive subscriptions. In the two-step method, the pool contains all subscriptions regarding primitive events.

Each incoming primitive event has to be filtered against all subscriptions in that pool. Consumers with subscriptions regarding primitive events (i.e., consumers A and C) are notified after the detection of these events. This detection of the primitive events is the first step in the event detection mechanism.

The results of the primitive filtering serve as input for the composite filtering (figures 4.8(a) and 4.8(b)). The subscriptions regarding composites are stored in the composite pool, represented by the square in figure 4.8. The incoming primitive events are assigned to the composite subscriptions. If all contributing events for a certain composite did occur, the composite event is signaled to the interested clients (client B in figure 4.8(c)). If the time span between the primitive events is larger than T, the composite subscription is not matched, and the detected primitive events are dismissed.

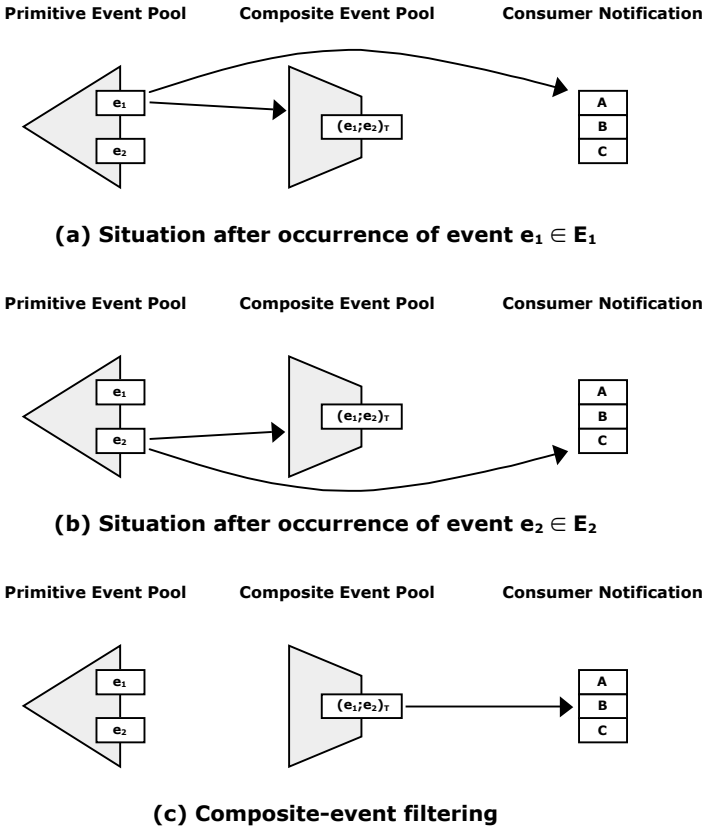
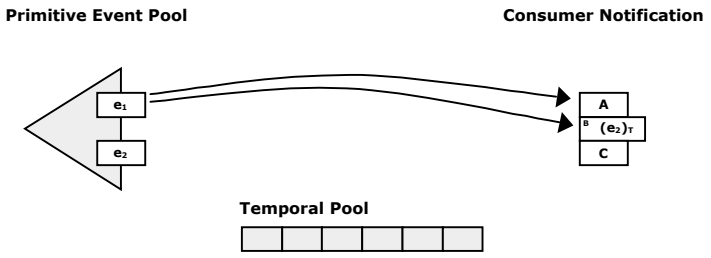


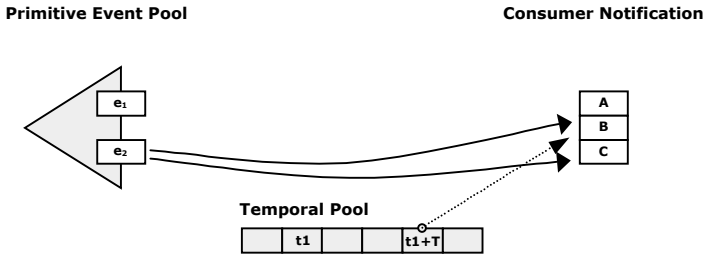
Figure 4.8 Composite- event detection using two-step methods

Figure 4.9 shows the principle of the *single-step detection algorithm*. The primitive event pool and a temporal pool are required, the composite pool is not used. After the detection of a primitive event, the interested clients are notified (consumer A in figure 4.9(a)). For storing the information about composite subscriptions, auxiliary subscriptions are created. After the match of a contributing primitive event, an internal notification regarding the auxiliary subscription is created. This notification triggers the insertion of the remaining composite part into the primitive subscription pool. Consider the auxiliary subscription for the composite subscription of consumer B: an internal auxiliary client is notified about the occurrence of the partial composite event (figure 4.9(a)). The auxiliary subscriptions for the internal client carry the information about the composite subscription as well as the information about its partial evaluation.

In figure 4.9(b), after the detection of event e_1, the subscription for event e_2 has been inserted into the pool. For the observation of the maximal time span T, an auxiliary terminator reference is inserted into the temporal pool. Terminator references cause the removal of the referenced subscription from the pool. The temporal terminator in figure 4.9(b) causes the removal of the subscription for client B at time $t1+T$, where $t1$ is the time of the primitive event e_1. After the match of the final contributing primitive event within the composite subscription, a notification is sent to the client.



(a) Situation after occurrence of event $e_1 \in E_1$ at time t1



(b) Situation after occurrence of event $e_2 \in E_1$

Figure 4.9 Composite- event detection using single-step method

Using the single-step method, the notification about the composite does not suffer additional delays due to additional filtering of binding predicates for the composition. Algorithm 4.2 shows the pseudo-code for the sequence detection using the single-step method.

Algorithm 4.2 Single-step detection of sequence $(E_1;E_2)_T$

input: e – event-message to be filtered
S – subscription pool with currently observed subscriptions
output: notification about $(E_1;E_2)_T$

1: initialize S with E_1, $(E_1 \Rightarrow E_2)$, and $(E_1 \Rightarrow T)$
2: **on** event e
3: **if** $(e \in E1)$ **then**
4: **if** $(E_2 \notin S)$ **then** insert E_2 in S
5: set reference $r_1 : (E_2 \Rightarrow e)$
6: insert $t_1 : (t(e) + T)$ in S with reference $r_2 : (t_1 \Rightarrow E_2)$
7: **if** $((e \in E_2)$ && (∃ reference r with $(E_2 \Rightarrow e)))$ **then** notify about $(E_1;E_2)_T$
8: **if** (e = time event t_1) **then** remove reference r_1
9: **if** (no references from E_2) **then** remove E_2 from S
10: remove t_1 from S

4.4 Conclusion

This chapter presented the proposed event management and composition framework for building adaptive systems. We presented the general architecture for such a framework and describe the component for event management and composition that it implements. The

framework was defined around flexible and general models, each of which focuses on a different design concern.

The framework nature allows to have a configurable infrastructure for personalized systems, therefore the applicative logic of the system is separated from the event management that the frameworks implements. The design of the models uncouples the event modeling from the event management aspect.

The architecture of an event-based system was presented in order to show the functionality of the event management and composition service. The event processing that the service implements was described in terms of primitive and composite event filtering. We adopted the single-step method for the filtering of composite events which integrates the detection of composite events into the detection of primitive events, allowing accelerating the identification of composite events and reducing the overall filtering time.

Therefore, the proposed framework is well suited for building adaptive systems. It provides adaptability and expressiveness to event-based systems allowing the personalization of event types, composition operators and the composition algorithm.