A Framework for Event Composition in Distributed Systems

Peter R. Pietzuch[*], Brian Shand[**], and Jean Bacon

University of Cambridge Computer Laboratory,
Cambridge CB3 0FD, United Kingdom
{Peter.Pietzuch, Brian.Shand, Jean.Bacon}@cl.cam.ac.uk

Abstract. For large-scale distributed applications such as internet-wide or ubiquitous systems, event-based communication is an effective messaging mechanism between components. In order to handle the large volume of events in such systems, composite event detection enables application components to express interest in the occurrence of complex patterns of events. In this paper, we introduce a general composite event detection framework that can be added on top of existing middleware architectures — as demonstrated in our implementation over JMS. We argue that the framework is flexible, expressive, and easy to implement. Based on finite state automata extended with a rich time model and support for parameterisation, it provides a decomposable core language for composite event specification, so that composite event detection can be distributed throughout the system. We discuss the issues associated with automatic distribution of composite event expressions. Finally, tests of our composite event system over JMS show reduced bandwidth consumption and a low notification delay for composite events.

1 Introduction

Event-based communication has become a new paradigm for building large-scale distributed systems. It has the advantages of loosely coupling communication partners, being extremely scalable, and providing a simple application programming model. In event-based systems, events are the basic communication mechanism. An event can be seen as a notification that something of interest has occurred within the system. Components either act as *event sources* and *publish* new events, or *event sinks* and *subscribe* to events by providing a specification of events that are of interest to them. A *publish/subscribe* (pub/sub) communication layer [1] is then responsible for disseminating events; for efficiency, it can often also filter events by topic or content, according to client specifications.

Many existing pub/sub systems [2–4] restrict subscriptions to single events only and thus lack the ability to express interest in the occurrence of *patterns of events*. However, especially in large-scale applications, event sinks may be

[*] Research supported by UK EPSRC and QinetiQ, Malvern.

[**] Research supported by ICL, now part of Fujitsu, and the SECURE EU consortium.

ACM/IFIP/USENIX Int. Middleware Conference 2003. © Springer-Verlag.

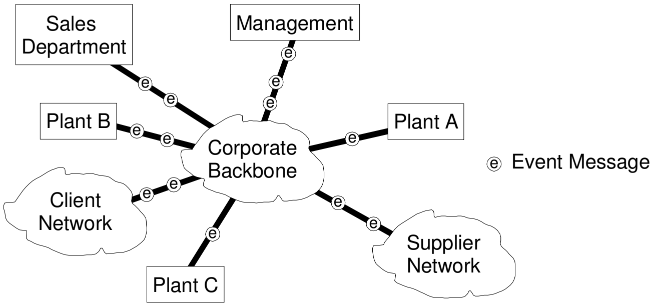


Fig. 1. A publish/subscribe system in a corporate network

overwhelmed by the vast number of primitive, low-level events, and would benefit from a higher-level view. Such a higher-level view is given by *composite events* (CE) that are published when an event pattern occurs. To date, it is usually left to the event sink to implement a detector for composite events making it unnecessarily complex and error-prone.

In this paper, we address the problem by proposing a general framework for composite event detection that works on top of a range of pub/sub systems. This framework includes a generic language for specifying composite events and CE detectors that can detect composite events in a distributed way.

The paper is organised as follows: Section 2 motivates the necessity of composite event detection in large-scale distributed systems. After related work (Sect. 3), we discuss prerequisites of the detection framework (Sect. 4) such as the pub/sub infrastructure requirements, the time model and the event model. The CE detectors and the associated core language are presented in Sect. 5, and Sect. 6 discusses distributed detection. In Sect. 7, we present our implementation over JMS, and evaluate its performance. The paper finishes with an introduction to higher-level specification languages (Sect. 8) and conclusions (Sect. 9).

2 Motivation

Large-scale event systems need to support CE detection, in order to quickly and efficiently notify their clients of new, relevant information in the network. This is particularly important for widely distributed systems where bandwidth is limited and components are loosely coupled. In such systems, distributed CE detection can improve efficiency and robustness.

For example, consider a large corporate network which connects disparate information systems, illustrated in Fig. 1. The computer system at one site might use the network to notify a supplier that more raw materials were required. At the same time, the sales department might notify all plants of projected regional demand for each product, in order to guide production. Finally, management might want to be informed of all orders over £10 000 from new clients, or of plants increasing production when demand was falling.

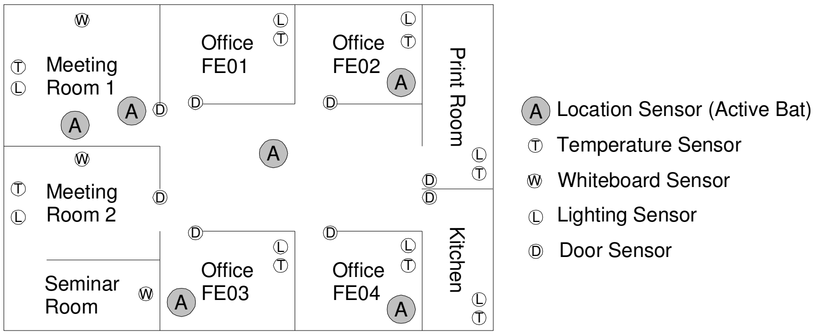


Fig. 2. An Active Office environment

In a small company, simple point-to-point messaging between departments would be sufficient. However, this would require considerable administration in a larger organisation, as each information producer would need a list of all intended recipients. A pub/sub system would reduce this overhead, allowing more flexible communication and easier bootstrapping of the system.

Nevertheless, without CE detection, many messages would still be sent unnecessarily, because specific event combinations or patterns could not be expressed by recipients. Instead, in the example above, management would have to be notified independently of all large orders and of all new clients. Furthermore, reuse of common subexpressions would be impossible, if for example both management and accounting were interested in orders over £10 000.

For reliability and efficiency, each CE detector should be distributed near to its event sources. Otherwise, if one site's connection to the rest of the network failed, local notification of composite events might fail unnecessarily. Besides, sending these events off-site for detection would have been a waste of bandwidth, if all relevant events were known to be locally produced.

Just as a general purpose pub/sub system supports flexible messaging, so too can a generic CE framework extend this support. Therefore, this paper proposes a general purpose middleware system for CE detection, independent of the specific underlying pub/sub infrastructure. By making CE detection closely interoperate with the underlying communication infrastructure, we obtain a system that is more efficient than an *ad hoc* implementation of CE detectors at the application level.

2.1 Application Scenario: The Active Office

The Active Office is a computerised building which is aware of its inhabitants' behaviour (cf. Fig. 2). Workers wear Active Bats [5] to inform the building of their movements at least once a minute. Other sensors monitor doors, office temperatures, electronic whiteboard usage, and lighting. A content-based pub/sub system is used so that applications can be notified of specific events, such as 'location events where Peter is seen in room FE04'. We used the following two application scenarios to test our CE detection framework:

Scenario 1. The building services manager wants to know about temperature events under 15℃ in an occupied room.

Scenario 2. Jean wants the list of participants and the electronic whiteboard contents of any meeting she attended to be sent to her wireless PDA, but only if she does not login to the workstation in her office within 5 min of the meeting.

There are many advantages of using a CE middleware for services in an Active Office, instead of (or perhaps as well as) offering predefined composite subscriptions on dedicated servers. The most important are the flexibility with which recipients can compose personal subscriptions, and the ease with which composite patterns can be reused and distributed close to event sources. The cost of establishing this network of CE detection broker nodes is then offset by the simplicity of configuring it for new CE subscriptions.

3 Related Work

Historically, composite event detection first arose in the context of triggers in active databases. Early languages for specifying composite events follow the Event-Condition-Action (ECA) model and resemble database query algebras with an expressive, yet complex syntax. In general, the detection process is not distributed.

In the Ode object database [6], composite events are specified with a regular-expression-like language and detected using finite state automata (FSA). Equivalence between the CE language and regular expressions is shown. Since a composite event has a single timestamp of the last event that led to its detection, a total event order is created that makes it difficult to deal with clock synchronisation issues. The pure FSAs do not support parameterised events.

CE detectors based on Petri Nets are used in the SAMOS database [7]. Coloured Petri Nets can represent concurrent behaviour and manage complex data such as event parameters during detection. However, even for simple expressions, they quickly become complicated. SAMOS does not support distribution and has a simple time model that is not suitable for distributed systems.

The motivation for Snoop [8] was to design an expressive CE specification language with powerful temporal support. A CE detector is a tree that reflects the structure of the event expression. Its nodes implement language operators and conform to a particular *consumption policy*. A consumption policy influences the semantics of an operator by resolving which events are consumed from the event history in case of ambiguity. For example, under a *recent* policy only the most recently occurring event is considered; others are ignored. Detection propagates up the tree with the leaves of the tree being primitive event detectors. A disadvantage is that the nodes are essentially Turing-complete making it difficult to formalise their semantics and to reason about their behaviour. The use of consumption policies can be non-intuitive and operator-dependent.

In [9], Schwiderski presents a distributed CE architecture based on the 2g-precedence model for monitoring distributed systems. This model makes strong assumptions about the clock granularity in the system and thus does not scale to large, loosely-coupled distributed systems. The language and the detection algorithm used are similar to Snoop and suffer from the same shortcomings. It addresses the issue of events being delayed during transport by *evaluation policies*: *asynchronous evaluation* enables a detector to consume an event as soon as it arrives sometimes leading to incorrect detection, whereas *synchronous evaluation* forces a detector to delay evaluation until all earlier events have arrived, and assumes a heartbeat infrastructure. Although detection is distributed, no decision on the efficient placement of detectors in the network is made.

The GEM system [10] has a rule-based event monitoring language. It follows a tree-based detection approach and assumes a total time order. Communication latency is handled by annotating rules with tolerable delays. Such an approach is not feasible in an environment with unpredictable delays.

Research efforts in ubiquitous computing have led to CE languages that are intuitive to use in environments such as the Active Office. The work by Hayton [11] on composite events in the Cambridge Event Architecture (CEA) [12] is similar to ours in the sense that it defines a language that non-programmers can use to specify occurrences of interest. Hayton uses push-down FSAs to handle parameterised events. However, the language itself can become non-intuitive as the semantics of some operators is not obvious. Even though detectors can use composite events as their input, distributed detection is not dealt with explicitly. As in previous work, scalar timestamps are used.

Distributed pub/sub architectures such as Hermes [4], Gryphon [3, 13], and Siena [2] only provide parameterised primitive events and leave the task of CE detection to the application programmer. Siena supports restricted *event patterns*, but it does not define a complete pattern language.

In our CE detection framework, we adopt the interval timestamp model introduced in [14]. The partial order of timestamps in a distributed system is made explicit by having timestamps associated with an uncertainty interval. A CORBA-based detection architecture is presented in [14] that implements this time model. The notion of *event stability* is defined in order to handle communication delays. We extend this to cope with delays in wide-area systems.

4 Design and Architecture

The CE detectors in our framework recognise concurrent patterns of simpler events, generating a composite event whenever a match is found. The component layers of our detection architecture are illustrated in Fig. 3: Distributed CE detectors are compiled from expressions in our *core CE language*. Patterns can be specified using higher-level languages, which are first translated into the core CE language before compilation and execution.

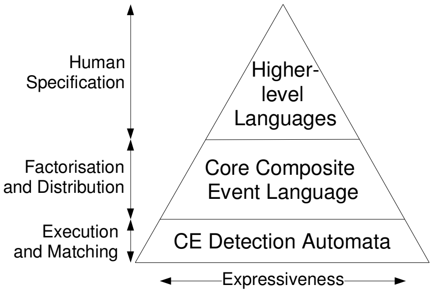


Fig. 3. Components of the composite event detection framework

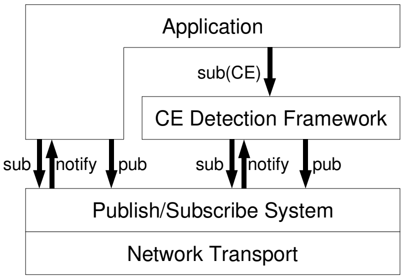


Fig. 4. Interface between the CE detection framework and the pub/sub system

The CE framework relies on and interacts with the underlying event system, in order to detect complex patterns of events. This section outlines the prerequisites for this interaction: an interface to a pub/sub infrastructure, and formal models of events and time. Given these prerequisites, the full expressive power of our CE languages can be used.

4.1 Publish/Subscribe Infrastructure Support

One of our design goals was to keep the CE detection framework strictly separated from the pub/sub infrastructure used. The interface to the event system (Fig. 4) makes only minimal assumptions about the functionality supported allowing our framework to be deployed on a large variety of pub/sub systems. Our current test-bed uses the Java Message Service (JMS) [15], but other pub/sub systems could equally be used: earlier work was based on Hermes [4], a distributed event-based middleware architecture, and CORBA Events would also be suitable.

In addition to the time and event model described below, the underlying pub/sub system needs to support (1) publication of primitive events by event sources, (2) subscription to these events by event sinks, and (3) relaying of events from sources to sinks. Many systems also filter events en route for efficiency; our CE framework uses this if available, but no particular publication or subscription model is assumed. Our event model uses the abstraction of a *describable event set* as an atom for CE detection. If the pub/sub system supports content-based filtering, a describable event set will be defined by a parameterised filtering expression. In a topic-based system, it will conform to a certain event type only.

In particular, the pub/sub system does not need to be aware of CE types. As illustrated in Fig. 4, application event sources submit CE subscriptions to the CE detection layer. Any composite events that are then detected by a CE detector are published to the pub/sub system disguised as primitive events. It is then the responsibility of the pub/sub system to disseminate these encapsulated CE occurrences to all interested event sources. The same mechanism is used for the communication between distributed event detectors (cf. Sect. 6).

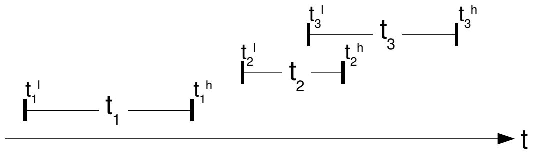


Fig. 5. Illustration of interval timestamps for events

4.2 Composite Event Detection Framework

The Java interface to the CE detection service, presented to applications, is shown below in part. Applications may use this for all event services, or contact the underlying pub/sub infrastructure directly for primitive event subscriptions.

```
public interface DistCEDServiceInf {
  public void registerCEType(CEType type, CEPublisherInf publisher);
  public void unregisterCEType(CEType type, CEPublisherInf publisher);
  public CEInf createCE(CEType ceType);
  public CEType createCEType(String typeName);
  public void publish(CEInf ce, CEPublisherInf publisher);
  public void subscribe(CEType type, CESubscriberInf subscriber,
                        CEQoSInf qos, CESubscriberCallbackInf callback);
  public void unsubscribe(CEType type, CESubscriberInf subscriber);
}
```

Before an event type can be published it must be `register`ed with the CE detection service so that e.g. an appropriate type/topic is created in the underlying pub/sub system. After that, a new event instance can be created using the `createCE` method. The `publish` method will pass the publication down to the pub/sub system. A call to `subscribe` subscribes to primitive or composite events. A CE subscription may trigger the instantiation of new CE detectors.

Time Model. Each event in our framework has an associated timestamp, denoting when it occurred. In a large-scale system, it may often be impossible to decide which of two events occurred first. Therefore we assume that there is a *partial* order relation on timestamps '$<$', showing which events definitively occurred before others. This is extended to a total order '$\prec$', using a tie-breaker convention (see App. A) allowing events to be treated as a well ordered sequence of symbols for detection.

This may be illustrated using a two-part interval timestamp for events, rather than a single conventional timestamp. These interval timestamps are used implicitly throughout the rest of the paper. They can represent the clock uncertainty of a distributed time service such as NTP, and also the time interval associated with a composite event. The intervals can factor in the estimated receiver-specific delay on receiving UTC, including radio transmission lag or network delays.

Figure 5 illustrates three interval timestamps t_1, t_2 and t_3. Here, $t_1 < t_2$, $t_1 < t_3$ but $t_2 \not< t_3$ and $t_3 \not< t_2$. On the other hand, using the total order, $t_1 \prec t_2 \prec t_3$; these operators are formally defined in App. A.

Event Model. Events provide notification of observations in a distributed system. *Primitive events* represent observations from outside the event system, while *composite events* represent patterns of events. The constituents of a composite event may be primitive events, or other, simpler composite events. Despite this distinction, all events are treated homogeneously; we assume only that events have timestamps, and can be consistently ordered for each subscriber (e.g. by interval timestamp, source IP address and local event generation count).

In an Active Office [5], primitive events might be ‘The door opens’ and ‘Peter is seen in the room’. Similarly, ‘The door opens, then Peter is seen in the room’ could be a composite event. Thus event sequences, of interleaved primitive and composite events, can be used to formalise the detection of composite events (cf. App. A). Furthermore, our event ordering still supports distributed detection, since each CE detector’s subscription is used to sequence only the events needed for its composite event pattern.

5 Composite Event Detection

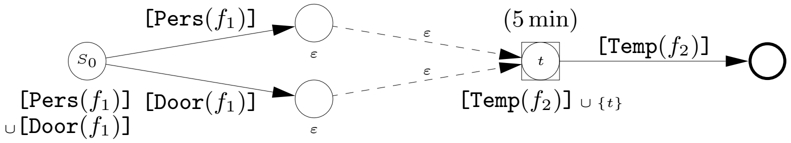
The CE detectors in our framework are simple automata, with a regular structure. Unlike conventional FSAs, these automata provide support for a rich time model and parameterisation, as well as the ability to detect concurrent event patterns. A novel language is used to express these patterns; this *core CE language* can then be compiled into automata for matching.

Distribution support is important for communication efficiency; Section 6 discusses how each pattern may be factorised into subexpressions. These subexpressions can then be matched independently on distributed nodes — these mobile detectors were discussed in an earlier paper [16]. Patterns may also be more intuitively defined using higher-level specification languages, described in Sect. 8. However, this is only a matter of convenience, not expressiveness; any patterns described in a higher-level language are first translated into the core CE language, before being compiled into automata. Figure 3 illustrates the relationship between these different aspects of the CE framework.

For example, in Scenario 1, the building services manager subscribes to temperature events under 15 ℃ in an occupied room. (For simplicity, we consider the movement, door and temperature events of only a single room, although multiple rooms could be represented in a single CE expression using parameterisation. We also prefilter the `PersonEvents`, limiting repeat notifications to one per minute.) We consider a room to be occupied if it has exhibited movement or door events within the last 5 min. In our core CE language, we might represent the primitive events (as exposed by JMS) using:

- `[PersonEvent(location='Office FE02')]`
- `[DoorEvent(location='Office FE02')]`
- `[TempEvent(location='Office FE02' AND temp<15)]`

Scenario 1 could be written (using $\texttt{[Pers(}f_1\texttt{)]}$, $\texttt{[Door(}f_1\texttt{)]}$, $\texttt{[Temp(}f_2\texttt{)]}$ for the expressions above, for brevity) as: $(\texttt{[Pers(}f_1\texttt{)]} \mid \texttt{[Door(}f_1\texttt{)]},\ \texttt{[Temp(}f_2\texttt{)]})_{t=5\,\mathrm{min}}$. This would be compiled into the following automaton, for use as a detector:

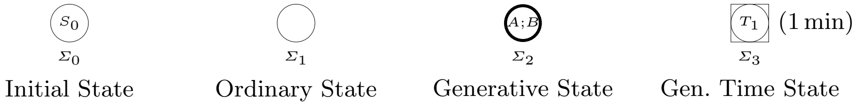


We based our core language and automata on regular expressions and FSAs for a number of reasons. Firstly, their expressive power is well understood, but they require only limited, predictable resource usage, and are thus a safer tool for distributed detection than a more general language. Still, they are powerful, and are frequently used for pattern detection and matching. Furthermore, regular expressions may also easily be factorised into subexpressions, for distributed detection of independent expressions. Finally, we felt that it would be more sensible to extend the commonly accepted regular expression operators with necessary additions, rather than arbitrarily define new operators, with the concomitant risks of redundancy or incompleteness. Our core CE language and automata therefore only minimally extend regular expressions and FSAs, to allow temporal relationships, input filtering, and parallel detection to be expressed.

5.1 Composite Event Detectors

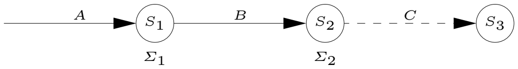
The automata which detect composite events contain a finite number of states and state transitions, but each state also maintains the timing information of the previous symbol detected. In a given state, the automaton decides when to make the transition to another state by considering new input symbols only from a per state describable subset of the global event input sequence I (cf. App. A.2).

Structure of Automata. Our automata have two types of state: *ordinary* and *generative*. A generative state causes a new event to be created, either a composite of the events matched so far (with a specified type), or an instantaneous time event in the future (with a freshly allocated local identity). The timestamp of the composite event will start at the earliest start time of the constituent events, and end at the latest end time. A time event may be used later in the automaton, to progress or fail after a given timeout (cf. App. A.2). Each state has an *input domain of describable events*, the family of events it can match. When in a given state, the automaton processes only those new events that lie within the state's domain. The diagram below shows four states: an initial (ordinary) state, an ordinary state, a generative state for a composite event of type '$A; B$', and a generative state for a time event. The input domains are $\Sigma_0 \ldots \Sigma_3$.

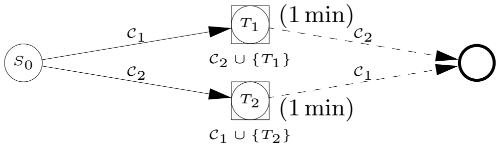


Each state can have any number of outward transitions. There are two types of transition: *strong* and *weak*, which can match events which strongly or weakly follow the previously detected event. These correspond to the partial and total event orderings ‘$<$’ and ‘$\prec$’ respectively. Each transition has a describable family of events attached, any of which will cause it to be taken.

New events in the input domain of a state but not in any transitions will cause the match to fail. These new events must strongly follow the previous event if all outgoing transitions are strong, or weakly follow otherwise. If there are two or more matching transitions, they will be followed nondeterministically. When a state with no outgoing transitions is reached, an event is generated if it is generative, then the machine (or the current nondeterministic branch) immediately terminates. The diagram below illustrates both strong and weak state transitions. If a, b, c are the events which matched A, B and C, then $a < b$ and $b \prec c$. Furthermore, $b \in B$ is the first event in the input stream I_{Σ_1} for which $a < b$. Similar constraints apply to c.



Limitations of Automata. The extended automata address many of the disadvantages of standard FSAs. Firstly, temporal support is provided by explicit event timestamps and special timer events. Concurrent events are also supported; the following automaton generates a new event when composite events $\mathcal{C}_1, \mathcal{C}_2 \in \mathcal{D}$ occur in parallel within 1 min of each other. $\mathcal{C}_1$ might represent ‘Peter is seen in the building but not in his office’ and $\mathcal{C}_2$ ‘Peter’s phone rings’. The resulting event could be used to divert the call to wherever Peter was last seen.



Conventional FSAs have other limitations too. Most importantly, they cannot handle event interrelationships such as event parameterisation. For example, to detect how long each door in a building is left open, a mechanism is needed for expressing free parameters which apply the same expression to all rooms: detect opening (x) followed by closing(x). Our framework can resolve this issue by filtering on the CE attributes of all opening and closing event pairs as soon as they are detected, reporting only matching pairs. This is still efficient, since the unnecessary composites are discarded as soon as they are detected, and every possible pairing would have been considered.

Finally, when nondeterministic FSAs are made deterministic, the number of states can grow exponentially. Although our automata potentially exhibit this behaviour, it does not happen in practice: since distribution takes place at the level of CE expressions (see Sect. 5.2), not automata, resolution to deterministic automata is not required; instead, a list of active states is held. Furthermore, in typical composite expressions this list is usually short, since distributed detection makes parallel detection of independent subexpressions the norm.

Formal Definition of Automata. Each automaton consists of a set of states S, state domains $a_S : S \to \mathcal{D}$, and strong and weak transition domains a_{TS}, a_{TW} :

$S \times S \rightarrow \mathcal{D}$. There is also a start state $S_0 \in S$. Finally, $G \subseteq S \times (\mathbb{T} \uplus \mathcal{D})$ defines the generative states (an extension of accepting states) and their actions.

The current state of an automaton is $C \subseteq S \times T \times \mathbb{P}(\mathbb{E})$ where T is the set of possible timestamps. In other words, the current state consists of a number of triples, each representing a state, a timestamp, and a list of detected events. From the perspective of the automaton, the list of events is opaque, except that extra detected events may be added to it, when a transition is made.

5.2 Core Composite Event Language

A CE language allows expression of CE patterns. In this section, we introduce our core CE language, which can easily be compiled into automata, but is still human readable, and outline its grammar. This language also defines the level at which subexpressions are chosen for distributed detection. App. B contains the transformation from expressions into automata, and gives precise operator semantics. The operators of the core CE language extend those found in regular languages, namely concatenation, alternation, and iteration, with operators for timing control, parallelisation, and weak/strong event sequencing. In contrast with other CE languages, we avoided redundant operators to simplify analysis.

Atoms. $[A, B, C, \cdots \subseteq \Sigma_0]$. Atoms detect individual events in the input stream. Here, only events in $A \cup B \cup C \cup \ldots$ will be successfully matched. Other events in Σ_0 will cause a failed detection, and events outside Σ_0 will be ignored. We abbreviate negation using $[\neg E \subseteq \Sigma]$ for $[\Sigma \backslash E \subseteq \Sigma]$, and also write $[E]$ instead of $[E \subseteq E]$. (Negation ensures any other events in Σ will stop the detection, such as timeouts or stopper events.)

Concatenation. $\mathcal{C}_1\mathcal{C}_2$. Detects expression $\mathcal{C}_1$ *weakly* followed by $\mathcal{C}_2$.

Sequence. $\mathcal{C}_1; \mathcal{C}_2$. This detects expression $\mathcal{C}_1$ *strongly* followed by $\mathcal{C}_2$. Thus $\mathcal{C}_1$ and $\mathcal{C}_2$ must not overlap in a sequence, but they may in a concatenation.

Iteration. $\mathcal{C}_1^*$. Detects any number of occurrences of expression $\mathcal{C}_1$. If $\mathcal{C}_1$ detects a symbol which causes it to fail, then $\mathcal{C}_1^*$ will fail too. (So $[A][A \subseteq \{A,B\}]^*[C]$ would match input AAC but not $AABC$.)

Alternation. $\mathcal{C}_1 \,|\, \mathcal{C}_2$. This expression will match if either $\mathcal{C}_1$ or $\mathcal{C}_2$ is matched.

Timing. $(\mathcal{C}_1, \mathcal{C}_2)_{T_1=\text{timespec}}$. The timing operator detects event combinations within, or not within, a given interval. The second expression $\mathcal{C}_2$ can then use T_1 in its event specification.

Parallelisation. $\mathcal{C}_1 \,\|\, \mathcal{C}_2$. Parallelisation detects two composite events in parallel, and succeeds only if both are detected. Unlike alternation, any order is allowed, and the events may overlap in time.

The following examples illustrate the use of the core CE language to describe composite events. Let B be the events corresponding to 'Brian enters the room', let P be 'Peter enters the room', and let A be 'anyone enters the room'.

1. Brian enters the room followed by Peter: $[B]; [P]$
2. Brian enters the room before Peter: $[B \subseteq \{B, P\}]$
3. Brian enters and Peter follows within an hour: $([B], [P \subseteq \{P, T_1\}])_{T_1=1\,\text{h}}$
4. Someone else enters the room when Brian is away: $[B]\,[\neg B \subseteq A]\,[B]$

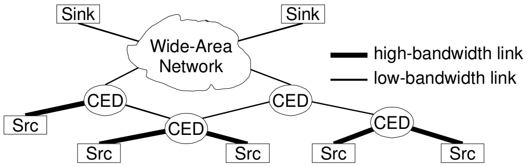


Fig. 6. Illustration of distributed composite event detection

6 Distributed Detection

In a large-scale distributed application, events are published at geographically dispersed sites. A centralised CE detector would have to subscribe to all primitive events that are part of a CE expression in order to detect occurrences of composite events. This could become a bottleneck and a single point of failure.

Instead, our framework provides a mechanism for distributing CE detectors. Detectors can be installed at various locations in the network and cooperate with each other. This cooperation is achieved by decomposing CE expressions stated in the core language into subexpressions that are then detected by detectors running at different nodes. Figure 6 shows an example of a network of cooperating CE detectors. The detectors are located close to event sources that publish events at a high rate, thus requiring high-bandwidth links. After CE detection, bandwidth consumption is reduced since composite events occur less frequently. Composite events are then sent to remote event sinks over a low-bandwidth, wide-area network. No CE detector is overwhelmed by the rate of primitive events, as it subscribes to at most two event sources.

The main difficulty when distributing detectors is to decide on their optimal placement within the system. This is complicated by the fact that the reasons for distributing detectors are potentially conflicting. For example, to minimise bandwidth usage, existing detectors should be reused for subexpressions as much as possible — even between applications, if this is appropriate. However, if minimum latency is required, detectors should be replicated at various regions in the network which leads to higher bandwidth consumption. As a result, an optimal solution must be a trade-off that takes the static and dynamic characteristics of the system and the requirements of the application into account.

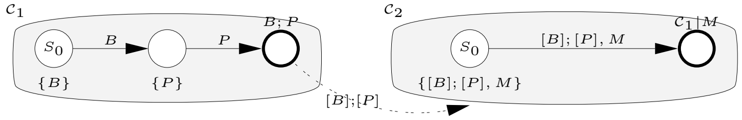
In our framework, mobile CE detectors (Sect. 6.1) detect composite events in a distributed fashion. A distribution policy (Sect. 6.2) ensures that detectors are installed at sensible locations and specifies a policy for their movement and behaviour during their lifetime. Network delays that can lead to incorrect detection are addressed by a detection policy (Sect. 6.3).

6.1 Mobile Composite Event Detectors

We introduce the concept of a *mobile composite event detector* to add distributed detection to our framework. A mobile CE detector is an agent-like entity encapsulating an automaton that detects an expression from our core language. It subscribes to event sources to receive event input streams and publishes the composite events detected by the automaton. The detector is capable of moving from one location to another in the network. This assumes the existence of a logical overlay network of nodes that supports the migration of components. Consequently, our work is built on top of a network of *event brokers*, for example corresponding to Hermes brokers or JMS nodes, where each event broker is capable of running one or more mobile detectors.

Whenever a CE subscription is added and no detector for this expression can be found in the system, a new mobile CE detector is created at an existing broker. It can carry out a number of actions: the detector can *factorise* the CE expression along its abstract syntax tree and *delegate detection* of subexpressions to other (already existing) detectors. For this, it can *instantiate new detectors* if it needs to reduce its own load. It can *migrate to another node* in the network that is for example closer to the event sources that it has subscribed to. Finally, the detector can *destroy* itself once it is no longer required.

Consider the Active Office application scenario introduced in Sect. 2.1. Let B be the event type corresponding to 'Brian enters the room', let P be 'Peter enters the room', and let M be 'a meeting takes place in the room'. A user is interested in occurrences of 'Brian enters the room followed by Peter'. The corresponding mobile CE detector $\mathcal{C}_1$ for the expression $[B]$;$[P]$ is shown below.



When another user subscribes to occurrences of the composite event 'Brian enters the room followed by Peter or a meeting takes place', this new expression $([B];[P])|M$ can be rewritten as $\mathcal{C}_1|M$. Therefore, the new detector $\mathcal{C}_2$ can reuse the existing detector $\mathcal{C}_1$ by subscribing to $[B]$;$[P]$. The communication between the two detectors happens exclusively through the underlying pub/sub system.

6.2 Distribution Policy

The behaviour of a mobile CE detector with respect to its actions is governed by a *distribution policy* — a set of heuristics to be followed by the detector. Several dimensions are addressed by a distribution policy:

(1) The location of mobile CE detectors must be determined. On the one hand, bandwidth usage can be reduced by moving detectors close to event sources. Primitive events that constitute a composite event may be of interest only to the CE detector and should therefore not be widely disseminated throughout the entire system unnecessarily. On the other hand, CE detectors should be close to application components that subscribe to them to improve reliability and detection delay. (2) The degree of decomposition and distribution must be stated in the policy (with optional hints from the application). Whenever a new CE subscription is created, existing detectors for this subexpression should be reused to save bandwidth and computational effort. (3) Detectors

must be replicated since, in a typical system, certain composite events will be more common than others. Detection for very common composite events should therefore be shared among several detectors for scalability.

6.3 Detection Policy

In a distributed system, events from different event sources travel along separate network routes to a mobile CE detector. Even if we assume that the network itself does not reorder events, out-of-order arrival of events at the detector can occur because of the different associated network delays. Whenever a new event arrives, it has to be inserted at the correct position in the totally-ordered event input stream before the stream is fed into the automaton.

The problem is to decide when the next event in the event input stream can be safely consumed by the automaton without risking that an event with an older timestamp is still being delayed by the network. Premature consumption could lead to an incorrect detection or non-detection of a composite event. Thus, each CE subscription is annotated with a *detection policy* that specifies when a detector can consume an event from an event input stream.

Best-Effort Detection. A best-effort detection policy states that events are consumed from event input streams without delay. Whenever an event is available, it will cause a state transition (or failure) in the automaton. Although this policy may lead to incorrect detection, it can be applied by applications that are sensitive to detection delay and are willing to ignore false positives.

Guaranteed Detection. Under a guaranteed detection policy, an event is consumed from an event input stream only once it has become *stable*[1] [14]. The consumption of only stable events ensures that no spurious composite events are detected. A detector knows that an event is stable after another event with a later timestamp from the same event source has been inserted in the event input stream. An event source that does not publish events at a high enough frequency can publish dummy *heartbeat events* that are used to ‘flush the network’.

In an asynchronous distributed system, a guaranteed detection policy potentially introduces an unbounded delay at the detector. For instance, an event source might fail or decide not to cooperate by not sending heartbeat events. To avoid this problem, we are currently investigating a *probabilistic stability* metric. As opposed to a simple binary stability measure, a detector attempts to model the probability that a particular event in an event input stream is stable and the event is only consumed if its stability metric is above a certain threshold.

7 Implementation using JMS

This section describes the implementation and performance results of our CE framework over JMS using JORAM [17], an open-source implementation of the

[1] An event is stable if there is no other event with an earlier timestamp in the system that should be part of this event input stream and should thus be consumed instead.

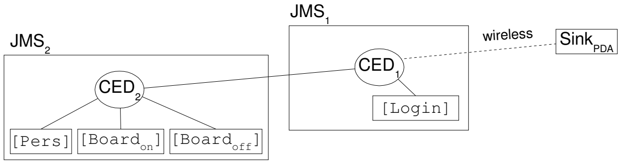


Fig. 7. Implementation of Scenario 2 using the CE framework

JMS API. Application programs can then publish and subscribe to composite events using the `DistCEDServiceInf` interface, presented in Sect. 4.1, that is provided by the event brokers in the system. In the pub/sub messaging model supported by JMS, a publisher registers a topic with a particular JMS provider, such as a JORAM or J2EE server. Whenever a message is published on the topic, topic subscribers are notified by the JMS provider via a callback mechanism. Content-based filtering on the fields in the message header is supported.

Although a JMS provider can be a distributed service, most current implementations are centralised, though they may provide redundancy through replication and clustering. Thus, clients may need to connect to several providers, such as a local and a remote message server. Therefore, the binding of our CE framework to JMS does not assume that all events (primitive or composite) have a single JMS provider. Instead, our implementation uses a JNDI directory to look up the JMS server for a particular topic. For composite events, we use this to ensure that we establish only a single CE detector for a given CE type, since all such detectors will produce the same events. Since the directory may itself be distributed, this does not imply unnecessary centralisation.

To support automatic distribution of CE detection, all event brokers subscribe to a common *administration topic* (`DistCEDAdminTopic`) that is hosted by an *admin JMS server*. When a new CE expression needs to be detected, the event brokers collectively decide how and where to instantiate the mobile CE detector (i.e. the expression's automata), and register the locations of newly created detectors with the JNDI directory. In the following experiments, the distribution policy is a simple choice function derived from the hash of the CE type name, although we are investigating the more complex policies outlined in Sect. 6.2.

7.1 Evaluation and Results

To test the CE framework implementation on JMS, we simulated the Active Office Scenario 2 described in Sect. 2.1. The movement of people was treated as a Markov process, with a probability matrix describing the likely movements in each time interval. We used the office layout shown in Fig. 2 with 9 rooms and 15 occupants. Eight of the occupants were classed as residents, predisposed to use the offices, while the remainder were visitors preferring the meeting rooms. The event sinks in the scenario were PDAs connected by an (expensive) wireless link with limited bandwidth. The goals of the experiment were to minimise the usage of that link and to achieve a low notification delay for composite events.

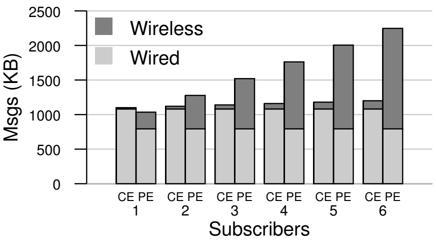


Fig. 8. Amount of data sent

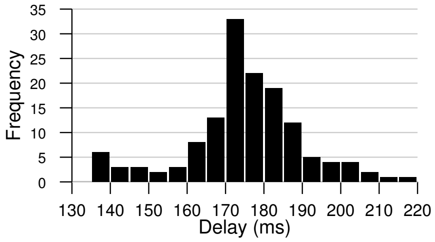


Fig. 9. Delay distribution of $\mathcal{C}$

The CE subscription $\mathcal{C}$ that was presented to our CE framework as the subscription submitted by the event sinks was as follows:

$$\mathcal{C} \equiv \left([\mathcal{C}_1(f_1)]\,,\ [T_1] \subseteq \{T_1, \texttt{Login}(f_2)\} \right)_{T_1 = 5\,\mathrm{min}} \tag{1}$$

$$\mathcal{C}_1 \equiv \texttt{[Board}_{\mathrm{on}}\texttt{]}\ \left[\texttt{[Pers(}f_3\texttt{)]}\ \texttt{[Pers(}f_3\texttt{)]}^*\ \texttt{[Board}_{\mathrm{off}}\texttt{]} \subseteq \{\texttt{Pers(}f_3\texttt{)}, \texttt{Board}_{\mathrm{off}}\} \right] \tag{2}$$

where f_{1-3} are JMS filter expressions that are omitted for brevity. Figure 7 shows how the detection of $\mathcal{C}$ was distributed over two event brokers by the CE framework. The detector CED_2 was responsible for the subexpression $\mathcal{C}_1$. All the primitive events that it subscribed to and the resulting composite events were located on the server JMS_2. CED_2 then detected the complete expression $\mathcal{C}$ and output its composite events to a different server JMS_1.

We compared our CE framework (CE) against a JMS-only solution (PE), in which the wireless PDAs subscribed to all the primitive events and performed the CE detection themselves in an *ad hoc* manner. Figure 8 shows the total data transferred over the wireless and wired networks with a changing number of subscribers. As expected, there is a small overhead when using our CE framework for a single subscriber. However, as the number of subscribers increases, less data needs to be sent over the wireless network because CE detectors can be reused. For 6 subscribers, our CE framework generates 53% of the total traffic generated by the primitive *ad hoc* solution, and only 8% of the wireless traffic. Note that the traffic over the wired network stays roughly constant as it is mainly caused by primitive event sources sending messages to the JMS servers.

The additional notification delay introduced by our CE framework is small. The plot in Fig. 9 shows the distribution of delay that it takes for a subscriber to be notified of an occurrence of $\mathcal{C}$ after the composite event logically happened in the system, i.e. its last primitive event was published. The notification delay stays below 220 ms and is fairly constant during the course of the experiment.

8 Further Work: Higher-Level Specification Languages

When designing a language for the specification of composite events for ubiquitous applications, two conflicting requirements arise: Primarily, the language should facilitate the implementation of efficient detectors and be decomposable for distributed detection, i.e. the language should be optimised to be *machine-processable*. On the other hand, the syntax and semantics of the CE language should be clean and intuitive so that it is *human-processable*. Therefore, we introduce the idea of *higher-level specification languages* for humans to express composite events in a natural and domain-dependent way. These languages are then compiled down into our automata. Whereas our core CE language is optimised for machine detection, the higher-level languages focus on CE specification by end users or programmers. The following are three examples for such languages:

The Pretty Language. The 'pretty' language has a verbose syntax similar to many current rule-based specification languages. It does not have a minimal set of operators. CE specifications in the pretty language, such as `'Event A' followed_by 'Event B' within '1 h'` resemble English language statements making it easier for non-programmers to express composite events.

Programming Language Binding. A binding of composite events to a programming language such as C++ or Java attempts to hide CE specification by integrating it into the programming language making its usage easier for programmers. This can be achieved with a sequence of method calls on event objects that build a CE expression: `eventA.after(eventB.repeated(3))` At runtime, these method calls are translated into a core CE language expression.

Graphical Composition. In the Active Office, users may interact with the system at runtime by specifying its behaviour with rules based on composite events such as 'Turn off the office light after 7pm'. A graphical composition tool could be used that is based on a simple model that is familiar to users. For instance, CE streams could be visualised as water flows in pipes allowing different types of piping to be composed to build composite events.

9 Conclusions

In a world with many mobile entities and complex, internet-based applications, events will become the dominant communication paradigm. CE detection in these large-scale systems provides a means of managing the complexity of a vast number of events. We consider our work as a first step to face this challenge, by providing novel scalable middleware services such as generic CE detection.

In this paper, we have presented a general CE detection framework as an extension of an existing pub/sub middleware. The framework assumes a realistic, interval-based time model and its event model makes few assumptions about the pub/sub communication infrastructure employed. Our CE detectors are an easily-implementable extension of conventional FSAs. They can handle timestamps, concurrent events, and come with a core CE language that is expressive and decomposable. Higher-level specification languages can provide more domain-specific ways to specify composite events. The abstraction of mobile CE detectors allows distributed CE detection, making the framework more scalable and robust. We introduced the concept of distribution and detection policies that control the distributed behaviour of detectors. Finally, the implementation of our CE framework over JMS demonstrates that it can improve performance in a real pub/sub application, compared to client-side JMS subscriptions.

In further work, we plan to extend our simulation environment to experiment with various distribution policies and determine how these depend on the application, the distribution of event flows, the location of event sources and event sinks, and the topology of the network.

References

1. Eugster, P.T., Felber, P., Guerraoui, R., Kermarrec, A.M.: The Many Faces of Publish/Subscribe. Technical report, EPFL, Lausanne, Switzerland (2001)
2. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and Evaluation of a Wide-Area Event Notification Service. ACM Trans. on Comp. Sys. **19** (2001) 332–383
3. IBM T J Watson Research Center: Gryphon: Publish/Subscribe Over Public Networks. Whitepaper (2002)
4. Pietzuch, P.R., Bacon, J.M.: Hermes: A Distributed Event-Based Middleware Architecture. In: Proc. of the 1st Int. Workshop on Distributed Event-Based Systems (DEBS'02), Vienna, Austria (2002) 611–618
5. Addlesee, M., Curwen, R., Hodges, S., Newman, J., et al.: Implementing a Sentient Computing System. IEEE Computer Mag. **34** (2001) 50–56
6. Gehani, N.H., Jagadish, H.V., Shmueli, O.: Event Specification in an Active Object-Oriented Database. In: Proc. of the ACM SIGMOD International Conference on Management of Data. (1992) 81–90
7. Gatziu, S., Dittrich, K.R.: Detecting Composite Events in Active Database Systems Using Petri Nets. In: Proc. of the 4th RIDE-AIDS. (1994) 2–9
8. Chakravarthy, S., Mishra, D.: Snoop — An Expressive Event Specification Language For Active Databases. Technical Report UF-CIS-TR-93-007, Dept. of Computer and Information Sciences, Univ. of Florida (1993)
9. Schwiderski, S.: Monitoring the Behaviour of Distributed Systems. PhD thesis, Computer Laboratory, University of Cambridge (1996)
10. Mansouri-Samani, M., Sloman, M.: GEM: A Generalised Event Monitoring Language for Distributed Systems. IEE/IOP/BCS Distributed Systems Engineering Journal **4** (1997) 96–108
11. Hayton, R.: OASIS — An Open Architecture for Secure Interworking Services. PhD thesis, Computer Laboratory, Univ. of Cambridge (1996)
12. Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., Seidel, O., Spiteri, M.: Generic Support for Distributed Applications. IEEE Computer (2000) 68–77
13. Banavar, G., et al.: Information Flow Based Event Distribution Middleware. In: Middleware Workshop at ICDCS'99. (1999) 114–121
14. Liebig, C., Cilia, M., Buchmann, A.: Event Composition in Time-dependent Distributed Systems. In: Proc. of the 4th Int. Conf. on Coop. Inf. Sys. (1999) 70–78
15. Sun: Java™ Message Service. `http://java.sun.com/products/jms/` (2001)
16. Pietzuch, P.R., Shand, B.: A Framework for Object-Based Event Composition in Dist. Sys. In: Pres. at PhDOOS Workshop (ECOOP'02). `http://www.cl.cam.ac.uk/Research/SRG/opera/pub/phdoos02-ced.pdf`, Malaga, Spain (2002)
17. ObjectWeb Open Source Middleware: JORAM Java Open Reliable Asynchronous Messaging 3.2.0 Release. `http://www.objectweb.org/joram` (2002)

A Appendix: Formalising the Time and Event Models

A.1 Definition of Interval Timestamps

This appendix formalises our notion of an interval timestamp, while App. A.2 presents our model of the event subscriptions available to the CE service, in terms of describable events and event input sequences.

Conventional timestamps are often inappropriate for distributed event systems. In a distributed system, node clocks may have unknown jitter within a known synchronisation distance. As a result, if two nodes detect events A and B respectively, it may be impossible to decide which occurred first. An interval timestamp, consisting of a start time and an end time, can make this ambiguity explicit, yet remains consistent with the physical time order of the events [14].

Let $t = [t^l; t^h]$ be an interval time stamp with start and end times t^l and t^h ($t^l \leq t^h$). We define the order relations $<$ and $\prec$ and union operator $\cup$ as:

$$t_1 < t_2 \triangleq t_1^h < t_2^l \tag{3}$$

$$t_1 \prec t_2 \triangleq (t_1^h < t_2^h) \vee (t_1^h = t_2^h \wedge t_1^l < t_2^l) \tag{4}$$

$$t_1 \cup t_2 \triangleq [\min(t_1^l, t_2^l); \max(t_1^h, t_2^h)] \tag{5}$$

A.2 Formalising Describable Events and Input Sequences

Users of event systems subscribe for notification of relevant events. Our CE detectors use the same subscription mechanism to describe which events they need to receive. In a sense, therefore, subscriptions (and the associated filter expressions) represent the atomic input streams available to CE detectors.

Let $\mathbb{E} = \{e_1, e_2, \dots\}$ be the space of possible events in the system. Each event e has timestamp $T(e)$, and a unique identifier $u(e)$ ordered by '$<$'. We write $e_1^{t_1}$ to show $T(e_1) = t_1$. Events are then ordered, consistently with their timestamps:

$$\forall e_1,\, e_2 \in \mathbb{E},\; e_1 < e_2 \triangleq T(e_1) < T(e_2) \tag{6}$$

$$e_1 \prec e_2 \triangleq (T(e_1) \prec T(e_2)) \vee (T(e_1) = T(e_2) \wedge u(e_1) < u(e_2)) \tag{7}$$

The space of events may be further categorised. The special *empty event* $\varepsilon \in \mathbb{E}$ is always detected. *Time events* $\mathbb{E}_T \subseteq \mathbb{E}$ are made to occur at a given future instant or after a certain interval, when timers expire. With instantaneous timestamps ($t^l = t^h$), they help detect composite events with time restrictions. If not supported by the pub/sub infrastructure, CE detectors can generate them as needed.

Event systems often allow us to differentiate types of event, by subscribing to subspaces of the event space $\mathbb{E}$, e.g. 'events where a door opens', or 'events where FE04's door opens'. These sets of events are denoted by an upper case letters: E, A, B. (In pub/sub systems, these are often called event types.) Individual event instances, on the other hand, take lower case letters: e_1, e_2, a, b.

Subscriptions also need certain properties, to be useful for CE detection. For example, if subscriptions A and B are valid, then it should be possible to detect events matching both or either of the subscriptions, $A \cap B$ or $A \cup B$. (If this is not supported by the underlying event framework, it can be simulated by detectors, if the event input streams are well ordered together under the total order $\prec$.)

There should be a maximal subscription $\mathbb{E}_D$, of all events that can be matched. There is also a subscription to detect any predefined, matchable event alone (and the special empty event ε is always matched); see (8) below. Finally, CE detectors should also be able to detect events matching one subscription but not another.

Each subscription can be associated with the set of events that would match it. The theoretical collection of all of these subscription sets is the family of *describable event sets* $\mathcal{D} \subseteq \mathbb{P}(\mathbb{E})$. This is a special collection of subsets of $\mathbb{E}$: those which can be detected within the CE framework. $\mathcal{D}$ is closed under finite union, finite intersection and element complementing relative to $\mathbb{E}_D$:

$$\text{Let } \mathbb{E}_D = \bigcup_{E \in \mathcal{D}} E. \text{ Then } \mathbb{E}_D \in \mathcal{D},\ \varepsilon \in \mathbb{E}_D, \text{ and } \forall e \in \mathbb{E}_D, \{e, \varepsilon\} \in \mathcal{D} \qquad (8)$$

The automata which detect composite events need to be able to treat incoming events as a well ordered stream, in order to match sequential patterns of events. By totally ordering events with '$\prec$' this can be achieved, resulting in the *global event input sequence* $I = (e_1, e_2, e_3, \dots)$, where $e_n \prec e_{n+1}\ \forall n \in \mathbb{N}$.

However, not all events are relevant to all patterns, or at all stages of a particular pattern. Describable event sets provide partial views of the input events, selecting subsequences of I. Thus CE detectors can restrict their view of the input sequence to only the relevant symbols. For example, if $E \in \mathcal{D}$ then $I_E = (e_{E_1}, e_{E_2}, e_{E_3}, \dots)$ denotes the subsequence consisting of elements of E.

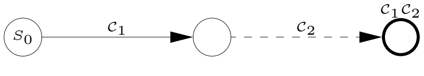
B Appendix: Generating Composite Event Detectors

This appendix details how expressions in our core CE language are transformed into CE detection automata, as outlined in Sect. 5.2. The grammatical components of our core language are listed below, with corresponding automata.

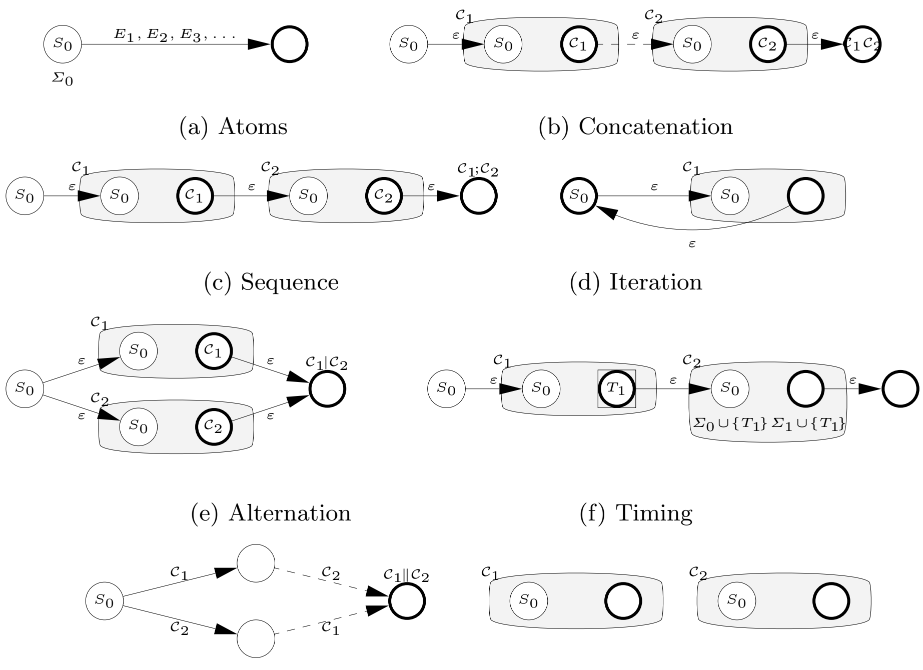
Atoms. $[A, B, \dots \subseteq \Sigma_0]$. Atoms detect individual events in the input stream. The resulting automaton considers an input stream of all events that are elements of Σ_0. Any input event in $A \cup B \cup \dots$ will be successfully detected; an event in Σ_0 but not in $A \cup B \cup \dots$ is a failure which stops expression matching (cf. Fig. 10(a)).

Negation. $[\neg E \subseteq \Sigma] \triangleq [\Sigma \backslash E \subseteq \Sigma]$. **Trivial Input.** $[E] \triangleq [E \subseteq E]$.

Concatenation. $\mathcal{C}_1\mathcal{C}_2$. Detects expression $\mathcal{C}_1$ *weakly* followed by $\mathcal{C}_2$. In the diagram, the shaded boxes are automata matching $\mathcal{C}_1$ and $\mathcal{C}_2$. An empty transition is then added for each generative state of $\mathcal{C}_1$ or $\mathcal{C}_2$, and those states become ordinary (cf. Fig. 10(b)). If $\mathcal{C}_1$ or $\mathcal{C}_2$'s detection were distributed, each submachine could be replaced by a single transition. Removing empty transitions[2] gives:



[2] Outgoing transitions from the second submachine's start state inherit the strength or weakness of the empty transition, but keep their original labellings.


s_0
$E_1, E_2, E_3, \ldots$
Σ_0
$\mathcal{C}_1$
$\mathcal{C}_2$
ε
$\mathcal{C}_1\mathcal{C}_2$
$\mathcal{C}_1;\mathcal{C}_2$
$\mathcal{C}_1|\mathcal{C}_2$
T_1
$\Sigma_0 \cup \{T_1\}$ $\Sigma_1 \cup \{T_1\}$
$\mathcal{C}_1\|\mathcal{C}_2$

(a) Atoms

(b) Concatenation

(c) Sequence

(d) Iteration

(e) Alternation

(f) Timing

(g) Parallelisation

Fig. 10. Composite event detectors

Sequence. $\mathcal{C}_1;\mathcal{C}_2$. This detects $\mathcal{C}_1$ *strongly* followed by $\mathcal{C}_2$. Thus $\mathcal{C}_1$ and $\mathcal{C}_2$ must not overlap in a sequence, but they may in a concatenation (cf. Fig. 10(c)).

Iteration. $\mathcal{C}_1^*$. Detects any number of occurrences of expression $\mathcal{C}_1$. If $\mathcal{C}_1$ detects a symbol which causes it to fail, then the composite machine $\mathcal{C}_1^*$ stops detecting iterations — even when $\mathcal{C}_1$ is distributed to another node (cf. Fig. 10(d)).

Alternation. $\mathcal{C}_1 \,|\, \mathcal{C}_2$. This expression will match if either $\mathcal{C}_1$ or $\mathcal{C}_2$ is matched by the input stream. This may result in nondeterminism for the number of input symbols which are matched by both $\mathcal{C}_1$ and $\mathcal{C}_2$ (cf. Fig. 10(e)).

Timing. $(\mathcal{C}_1, \mathcal{C}_2)_{T_1=\text{timespec}}$. The timing operator can be used to detect event combinations within, or not within, a given interval. In the above expression, event T_1 will be generated at a certain time after $\mathcal{C}_1$ is detected — either a relative time, such as a minute later, or an absolute time. The second expression $\mathcal{C}_2$ may then use T_1 as an event specification, in detecting composite events. Furthermore, $\mathcal{C}_2$ is extended so that all states include T_1 in their input domain. For distribution or reuse, the modified $\mathcal{C}_2$ detector is treated as distinct from the original, and it should be on the same node as the $(\mathcal{C}_1, \mathcal{C}_2)_{T_1}$ detector (cf. Fig. 10(f)).

Parallelisation. $\mathcal{C}_1 \,\|\, \mathcal{C}_2$. This can be used to detect composite expressions $\mathcal{C}_1$ and $\mathcal{C}_2$ in parallel. The diagram assumes that separate detectors for $\mathcal{C}_1$ and $\mathcal{C}_2$ already exist. They must be separate, to maintain the two independent timestamps needed for proper order restrictions on the two input sequences (cf. Fig. 10(g)).