A Framework for Object-Based Event Composition in Distributed Systems

Peter R. Pietzuch and Brian Shand
Computer Laboratory
University of Cambridge
JJ Thomson Avenue
Cambridge CB3 0FD, United Kingdom
{Peter.Pietzuch, Brian.Shand}@cl.cam.ac.uk

1 Introduction

Large-scale distributed systems benefit from new scalable communication mechanisms. In event-based publish/subscribe communication, components are either event sources that publish new events or event sinks that subscribe to events. Events can be seen as notifications that something of interest has happened in the system.

Composite events represent complex patterns of activity from distributed sources. Using mobile detection objects, we aim to distribute the detection of composite events too, increasing scalability; together with a novel composite event language, this improves the efficiency of detection and the reuse of computations in publish-subscribe systems, and also provides a natural object-based representation for composite events.

Consider a distributed computer system such as a stock exchange information service, in which users can be notified of changes in particular stock prices, and other price events. If a user wanted to be notified only when their shares were rising but the market index was falling, they could subscribe for these two events separately, in a traditional publish-subscribe system [3].

However, most of this information would be irrelevant, and a waste of the user's bandwidth. Furthermore, two users with the same notification needs would still have to perform the same correlations independently.

Instead, we propose a detection framework which allows commonly used composite event detections to be placed as near to the event sources as possible, and reused among subscribers. This is illustrated in Figure 1; here Composite Event Detector CED_3 determines when the market is falling. Another detector, CED_2 depends on this and also the behaviour of the user S_4's stock, notifying S_4 when necessary. The figure also shows the reuse of common expressions such as CED_3.

The paper is organised as follows: Section 2 introduces events and composite events and shows how composite events can be cleanly represented as objects. We present our composite event language that is based on the idea of regular expressions. Section 3 describes the entire detection framework and shows how it is integrated with a publish/subscribe system. The paper finishes with related work (Section 4) and our conclusions.

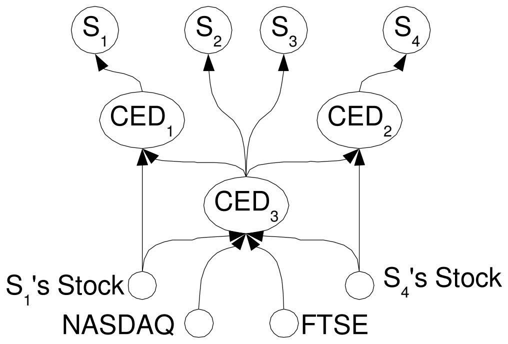

Figure 1. Distributed Composite Event Detection Example

2 Composite Event Objects

Composite Events represent patterns of simpler events. For example in an Active House scenario [1], 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' would be an example of a composite event.

Event objects denote observations in distributed systems, such as publish-subscribe systems. Eugster et al [4] have put forward the advantages of treating events as first-class objects. In this paper, we extend these ideas to composite events too.

Composite events may incorporate simpler composite events, as well as primitive events. For example, 'The door opens, then Brian or Peter is seen in the room' combines the primitive event 'The door opens' with the composite event 'Brian is seen in the room or Peter is seen in the room'. We therefore propose to treat both primitive and composite events homogeneously, with similar object interfaces, providing a framework for efficient distribution and reuse of composite events and the entities which detect them.

Composite event objects can then be constructed in a tree-like structure, with container classes for each of the operators of the composite event grammar (described below): concatenation, duplication, alternation, paralleliza-

```
interface EventObject {
  EventObject[] getChildren()
  EventObject[] getEventsMatched()
  CEExpression getExpression()
  ...
}
```

Table 1. Interface of Event Objects

tion, timing. The common interface of all events objects, primitive and composite, is shown in Table 1 — which uses the standard composite design pattern.

To illustrate this, consider the following expression: 'The front or back door opens, then Brian is seen in the room, immediately followed by Peter.'

For convenience, we define the following abbreviations: Event A = 'The front door opens', Event B = 'The back door opens', Event C = 'Brian is seen in the room', Event D = 'Peter is seen in the room'. The expression then becomes: '(A or B) then (C followed by D)'. Figure 2 illustrates how these primitive event objects can be combined into a composite event object.

Our composite event specification language is an extension of regular expressions, allowing temporal relationships and parallel behaviours to be specified, while retaining the basic syntax and structure of ordinary regular expressions. In this way, we build on proven expressive power and completeness, instead of arbitrarily choosing our own notation. Furthermore, if the maximum event rate is known in advance, then our language has identical expressive power to ordinary regular expressions, but with the advantages that our expressions are simpler to read and easier to distribute. Thus our composite event detectors could be implemented using finite state automata, if necessary, using predictable computational resources.

The following examples of composite events illustrate our language; the complete grammar is shown in Appendix A. (Terms marked † have the usual regular expression interpretation.)

Describable Events are the primitive event sets that can be matched. This space is assumed to be closed under finite intersection and union and complement, and includes the empty and universal sets of events. This would contain the events $A \ldots D$ above, and also event sets such as 'Anyone being seen in the room', and the universal set U.

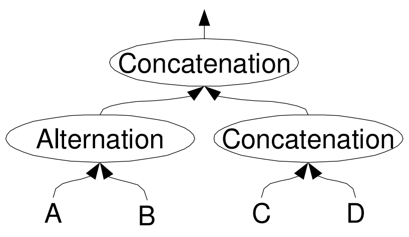

Figure 2. The Structure of a Composite Event Object

Atoms match some describable events but not others. These atoms specify the events to match, and also the space of events to be considered as potential matches. If an event in the latter set is encountered, the match will fail. For example, $[C \texttt{ in } CD]$ monitors all events matching C or D, but accepts only those matching C. In this case, CD is known as the *alphabet* of the expression. Note that $x \equiv [x \texttt{ in } x]$. This extends the ordinary regular expression syntax, in which the alphabet is always the set of all printable characters.

Concatenation† represents one regular expression immediately followed by another. E.g. $A[C \texttt{ in } CD]$ matches AC in AC or ABC, but not AD or ADC.

Duplication† allows a regular expression to be matched repeatedly, e.g. $(AC)^*$ matches any number of occurrences of A and C, so in $ABCACCAC$ the event pattern $ACACAC$ would be matched.

Alternation† matches either of two expressions, e.g. $AB|CD$ matches AB or CD or CAD, but not AD.

Parallelization matches two interleaved expressions, e.g. $<AB, CD>$ matches $ABCD$ or $ACDB$, but not $ABDC$.

Timing matches a composite event only if it occurs within a given time window, e.g. $(A, B)!1s$ matches AB only if B's timestamp is within a second of A's timestamp.

Alphabet Sharing is a convenient notation for limiting the patterns that a composite expression matches. For example, $C\&D$ forces the subexpressions to share the same alphabet, CD. Thus $C\&D \equiv [C \texttt{ in } CD][D \texttt{ in } CD]$, matching CD but not CCD.

Returning to the expression of Figure 2, this can be represented as $(A|B)(C\&D)$. Here, the shared alphabet of C and D ensures that Brian may not leave the room and return before Peter's entry, for the pattern to be matched.

This expression may easily be divided into the constituent composite events $(A|B)$ and $(C\&D)$, which can be matched independently before being combined. Particularly if the A's and B's were generated in a different place from the C's and D's, then this might be the most efficient approach to matching, minimising the communication bandwidth required. Furthermore, other expressions which relied on $(A|B)$ for example could reuse the computation with no extra computational cost.

The following section illustrates in detail how this distribution may be achieved.

3 Mobile Detection Objects

Our framework for object-based event composition is based on *Hermes*, an event-based middleware architecture [8]. Hermes consists of *event clients*, which can be

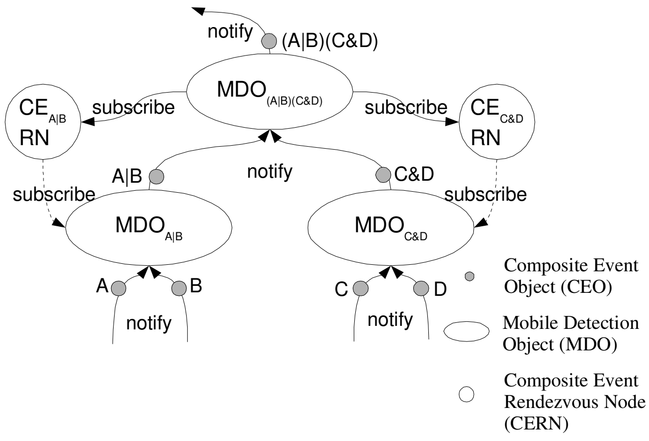

Figure 3. The Composite Event Detection Framework for the Expression $(A|B)(C\&D)$

```
interface MobileDetectionObject {
  subscribe(MDO mdo, CEventType cet)
  unsubscribe(MDO mdo, CEventType cet)
  notify(CompositeEventObject ceo)
  migrate(Node newNode)
  ...
}
```

Table 2. Interface of a Mobile Detection Object

event publishers or subscribers, and *event brokers*, which route events from publishers to subscribers. Event brokers are interconnected and form a logical overlay network for event dissemination. Hermes uses peer-to-peer routing techniques to deliver events. Each event broker maintains a set of application-defined objects that can influence all routing decisions at that broker. We use this mechanism to install *Mobile Detection Objects* (MDO) at brokers that detect occurrences of composite events.

A Mobile Detection Object contains a finite state automaton, as explained in the previous section, and detects patterns in the incoming event stream of a broker. Since one of our requirements was to detect complex composite events in a distributed fashion, MDOs can cooperate and delegate the detection of subexpressions to other MDOs or take advantage of already existing detectors. A Composite Event Expression (CEE) in our language can be decomposed into subexpressions that can be managed by different MDOs. The MDOs in the system are organised in a tree that reflects the syntactic structure of the corresponding CEE. These MDOs are agent-like [5] because they move freely through the network of event brokers trying to optimise the event detection process.

An example detection system with MDOs is shown in Figure 3 for the Composite Event Expression $(A|B)(C\&D)$. The whole expression is decomposed into subexpressions so that the MDO$_{(A|B)(C\&D)}$ takes advantage of the two detectors, MDO$_{A|B}$ and MDO$_{C\&D}$. Once an MDO detects a composite event, it creates a Composite Event Object that is sent to all interested MDOs and other subscribers.

A remaining problem is how subscribers or other MDOs locate existing, potentially migrating MDOs in the network. This is solved by introducing a proxy object for each composite event type that keeps track of all the MDOs for this type. These *Composite Event Rendezvous Nodes* (CERN) are located at well-known points in the network. A canonical representation of the composite event type is converted into a hash value that is used to find the corresponding CERN [8]. When a CERN receives a subscription for a composite event type, it subscribes to all existing MDOs on behalf of the original subscriber. In Figure 3, MDO$_{(A|B)(C\&D)}$ sends its subscription to CERN$_{A|B}$ which forwards the subscription to the current location of MDO$_{A|B}$. MDO$_{A|B}$ will send any future Composite Event Objects directly to MDO$_{(A|B)(C\&D)}$. If a new subscriber joins the system, and is interested in the composite event expression $A\&B$, then it can take advantage of the existing detector MDO$_{A\&B}$ and subscribe to it via the CERN$_{A\&B}$.

Table 2 summarises the main interface methods exported by a Mobile Detection Object. The `subscribe` method is usually invoked by a CERN on behalf of another object. It contains the composite event type and the identity of the subscriber. The `notify` method informs the MDO about the occurrence of a new event in the form of a Composite Event Object. The event could originate from an event publisher or another MDO. A call to `migrate` forces the MDO to leave the current event broker and move to a different one in the network.

In a large distributed system, event publishers and subscribers will dynamically enter and leave the system. As a result, the MDO tree for a composite event subscription will have to adapt to the state of the system, taking the set of current composite event subscriptions and the location of event publishers into account. In our framework, new MDOs can be created at arbitrary points in the network and existing MDOs can be destroyed when no longer needed. In addition, existing MDOs can migrate to different brokers in order to optimise the detection process. A number of parameters such as the load of the event broker, the distance to other MDOs or event publishers, and the structure of the event expression need to be considered before making a decision about migration. We are currently identifying heuristics that can govern the agent-like behaviour of MDOs in the system.

4 Related Work

Event composition first arose in the centralised context of active databases [2]. Thus, most composite event languages resemble database query algebras and are often not intuitive for specifying complex composite events. More recently, a number of projects addressed composite event detection in distributed systems: In [9], an architecture for monitoring distributed systems using events is presented. An event algebra for specifying composite events, similar to the ones found in active databases, is introduced and an implementation using tree-based detectors is provided. However, no automated method for distributing the detectors in the network is given, limiting the usefulness of this approach in a large-scale context.

The GEM architecture [7] has a complex, but expressive, rule-based monitoring language for a distributed system. No notion of a composite event type exists which makes it difficult to integrate this system with a modern object-oriented programming language. It introduces techniques to deal with network delays, but does not automatically distribute composite event detectors in the network.

The work of Liebig et al [6] develops a time model for composite event detection in a distributed system using partially ordered interval timestamps. The model is able to correctly deal with delayed events during the detection process. However, static event channels between event producers and event consumers are required without the flexibility of agent-like detectors.

5 Conclusions

This paper has shown a novel, object-based architecture for distributed detection of composite events. By building on existing publish-subscribe middleware and regular expression pattern languages, we aim to develop a powerful yet practical framework, to serve as a development environment with tight integration into existing object-oriented languages such as Java; this will present a programming paradigm well suited to large-scale distributed systems.

We also intend to extend our work, to allow users greater control over event ordering requirements, and over accuracy against network delays. Finally, we plan to determine effective heuristics for decomposing Composite Event Expressions, and for migrating Mobile Detection Objects. Our future prototypes we will use cost models to intelligently estimate the utility of each migration, as the network topology and usage change, allowing efficient detection of composite events.

References

[1] B. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. A. Shafer. EasyLiving: Technologies for Intelligent Environments. In *HUC*, pages 12–29, 2000.

[2] S. Chakravarthy and D. Mishra. Snoop — An Expressive Event Specification Language For Active Databases. Technical Report UF-CIS-TR-93-007, Department of Computer and Information Sciences, University of Florida, 1993.

[3] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. Technical report, EPFL, Lausanne, Switzerland, 2001.

[4] P. T. Eugster, R. Guerraoui, and C. H. Damm. On Objects and Events. In *Proc. for OOPSLA 2001*, Tampa Bay, USA, Oct. 2001.

[5] N. R. Jennings, K. Sycara, and M. Wooldridge. A Roadmap of Agent Research and Development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[6] C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-Dependent Distributed Systems. In *Proc. of the Fourth IECIS International Conference on Cooperative Information Systems*, 1998.

[7] M. Mansouri-Samani and M. Sloman. GEM — A Generalised Event Monitoring Language for Distributed Systems. In *Proc. of ICODP/ICDP '97*, 1997.

[8] P. R. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. To Appear in the Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02), July 2002.

[9] S. Schwiderski. *Monitoring the Behaviour of Distributed Systems*. PhD thesis, Computer Laboratory, University of Cambridge, 1996.

A Composite Event Grammar

Production Rules

```
expr : expr VBAR branch
     | branch

branch : branch term
       | empty

term : term AMPERSAND factor
     | factor

factor : factor DUPL
       | LPAREN factor COMMA factor RPAREN
         TIMESPEC
       | atom

atom : LPAREN expr RPAREN
     | LT exprlist GT
     | describable

exprlist : exprlist COMMA expr
         | expr

describable : SYMBOL
            | LSQUARE symbols RSQUARE
            | LSQUARE symbols IN \
                      symbols RSQUARE

symbols : symbols SYMBOL
        | empty
```

Selected Tokens

```
DUPL       = '[*+?]'
TIMESPEC   = '![0-9]+(.[0-9]+)?[hms]'
IN         = 'in'
```