Using Events for the Scalable Federation of Heterogeneous Components

John Bates, Jean Bacon, Ken Moody and Mark Spiteri
University of Cambridge Computer Laboratory, Cambridge, UK

Abstract

The thesis of this paper is that, using our event-based development principles, components that were not designed to interoperate, can be made to work together quickly and easily. The only requirement is that each component must be made event-based by adding an interface for registering interest in events and an interface for injecting actions. A component notifies an event to a distributed client if the parameters of an event, internal to the component, match the parameters of a particular registration. Heterogeneous components can be federated using event-based rules; rules can respond to events from any component by injecting actions into any other component. We show that the event paradigm is scalable by illustrating how event-based components can be located worldwide, using a federation of event brokers. Additionally, we illustrate with 3 event-based systems we have developed: a component-based multimedia system, a multi-user virtual worlds system and an augmented reality system for mobile users. Finally, we show how the event paradigm is also scalable enough to allow event federation of entire systems, not just single components. We illustrate by showing how we have federated the operation of the 3 featured event-based systems. This enables, for example, real-world mobile users to appear as avatars in the appropriate locations in the VR world, and for these avatars to move in response to actual user movements.

1 Principles of Event-based Systems

This section introduces distributed systems development using events. The framework presented here uses events to "glue" together components. By using events as a uniform way of informing other components of activities that have occurred, systems/applications can be built around a generic event-response paradigm. Using this approach, the entire Internet can become a plug-and-play domain.

An event is an asynchronous message containing parametrised details of an activity that has occurred within a component, or has been detected by the component. Services **publish** details of classes of event they actively monitor for, clients **register** interest with services, and services **notify** clients, as appropriate. Single events, or events as part of an ordered combination of events, notified to clients, can be used to trigger further actions within the system/application. Using events as the uniform interchange of activity between the possibly distributed components, simplifies the construction of a complex system/application. Our recent work is making fundamental contributions to the difficult problem of how to compose or federate existing systems to achieve interoperability. A related problem that is relevant within a single domain is how to add new object types (in our case, event classes) to a dynamically running system.

In this paper, we use the term *active system* to describe a large system or application composed of components interoperating via event federation.

1.1 Components and Glue

Event-based programming relies on two main concepts:

- Existing event-based components are used as the building blocks of applications. These may be distributed around the local network or Internet. They can inform interested users of pertinent events, such as a change in the location of a mobile user. The power of event-based system building is that no component is designed to work specifically with any other component.

- To compose many event-based components into an active system, a *federator* component must be written. This is the bespoke part of the system, composed of *event-based rules* that describe actions to take in response to the receipt of certain events from the components within the system. Rules act as a distributed systems integration *glue*, using **any** class of event from **any** components to drive **any** other components by injecting

any actions. Examples are moving a user interface to the application owner's new location in response to an event that the user has moved, or reconfiguring a multimedia application in response to an event detailing a change in bandwidth availability. In this paper we specify event-based rules in the following way:

```
rule <name> = <event expression>;
      {<list of invocations>}
```

In a multi-user application, for example a multimedia conference, each user *owns* a set of components, but the overall system is composed of the interacting set of all users' components. In such a situation, it is likely that each user will have a separate federator component to take care of his/her own components.

In the remainder of this section, we illustrate event-based programming concepts using two case studies. The first shows how a stand-alone drawing board can be made event-based and then used as a component in a CSCW application. The second shows how event services allow clients to tailor their range of interest. For this we use the example of a location service, which is part of our augmented environments framework. The location service is responsible for keeping clients location-aware.

1.2 Event Taxonomy and Brokerage

One event source can generate events of one or more event classes. An event class is analogous to an object class. It has typed attributes, instances of which uniquely identify a captured activity. For example, an event source that provides information about the locations of users can offer monitoring facilities for the following class of event: `LocationEvent(Domain,Name,Type,Location)`.
Class `LocationEvent` identifies that an entity has changed location in a specific domain. The type of the entity can be, for example, a person or a piece of equipment. To actually detect movement of entities, we used active badges [7], a form of electronic tag.

Distributed event-based objects advertise the event classes they can monitor for by exporting them to an *event broker*. Clients can query the event broker, to search for appropriate event sources.

An event taxonomy describes the set and organisation of event classes in a particular domain. We are investigating the organisation of event classes into class hierarchies of events, to support automatic propagation of interest in sub-classes. For example, consider the class `LocationEvent`. If we want to track the whereabouts of user `Mark.Spiteri`, we register interest with a local event broker. We intend that that should be all that is required to track him wherever he goes in our domain (and even world-wide – see below). He may be tracked by many different means, such as active badge or login detection within a building, outside by GPS [1]; in either, by security camera image recognition etc. Each class of specialist tracking can be encoded as a sub-class of `LocationEvent`. The event broker will automatically register our interest with all known subclasses.

Event sub-classing is also useful in allowing specialisation of event classes, whilst supporting compatibility with super-classes. For example, a new event class `FineGrainedLocation` can provide a 3D fix on a user, rather than just a named location. This can be built on top of a GPS system or a fine-grained office-based system [15], thus providing latitude, longitude and altitude information. In our design, the event broker is the custodian of the class hierarchy for its local event domain. Querying the event broker for a particular class of event returns all event sources that can monitor for the class and all its sub-classes. It is also possible to query a sub-class directly, events of which provide the specialised information not available in the super-class.

By using a class hierarchy, new subclasses may be added dynamically and registration of interest in a class may be extended to a new subclass. We are investigating how to use this mechanism to add new services that notify new event classes without recompiling clients.

Because event interfaces do not always describe the purpose of the event class, we are extending event brokers to support the association of free text with event classes. The broker will support searches, similar to Web search engines, returning event classes that match text queries. We are employing a locally developed search engine in our implementation [12].

1.3 Registration for Filtering

Some event sources generate thousands of events a second. Transmission of all of these is network intensive and requires a client of the event source to perform filtering to decide whether a particular event is of interest. Our event system allows interested parties to *register interest* in events with certain characteristics. Only events that match the registered patterns are dispatched to the interested parties, thus reducing network traffic and filtering requirements.

To register interest with an event source, clients provide an *event template*. This is an event instance, with fields for exact match filled in and those for wildcard match expressed as variables. Examples for class `LocationEvent` are as follows:

- `LocationEvent("cl.cam.ac.uk", "John.Bates", "Person", L)` – Report wherever John is seen in the Computer Lab.
- `LocationEvent("cl.cam.ac.uk", P, "Person", "Meeting Room")` – Report when anyone is seen in the Computer Lab meeting room.
- `LocationEvent(D, P, "Person", L)` – Report when anyone is seen in any of the places we receive events from.

When event instances are *notified* to clients, values for all class attributes are given, e.g. the instance `LocationEvent("cl.cam.ac.uk", "John.Bates", "Person", "Meeting Room")` means John has been seen in the meeting room. As a comparison to another event-based component, the drawing board advertises event class `LineDrawn(user,x1,y1,x2,y2)`. This allows us to find out about lines that are drawn on the board. We will show how this is useful shortly.

Our policy is to apply access control on event registration. We use the locally developed OASIS system [8]. We are currently investigating to what extent authorisation should extend automatically to new subclasses. For example, it might be that a given client is allowed to register with an active badge-based location service but is not permitted access to face recognition via security cameras.

1.4 Federation Using Rules

Event-based components also have an interface for *action injection*. This allows a client of a component to invoke methods that inject actions into the component. The action injection interface methods for a component are often the same as the classes of event that the object can notify (see drawing board example below).

Federating the operation of event-based components involves authoring a glue component, known as a federator. This component contains event-based rules. Each rule specifies one or a combination of events that, if detected, trigger a set of actions on components. As an illustration, consider how several drawing boards can be used to create a shared drawing application (see figure 1). Each board owner can author a federator; the federator registers interest in drawing events with all of the involved drawing boards (each of which may differ from the rest in implementation). When the federator is notified that a line has been drawn,

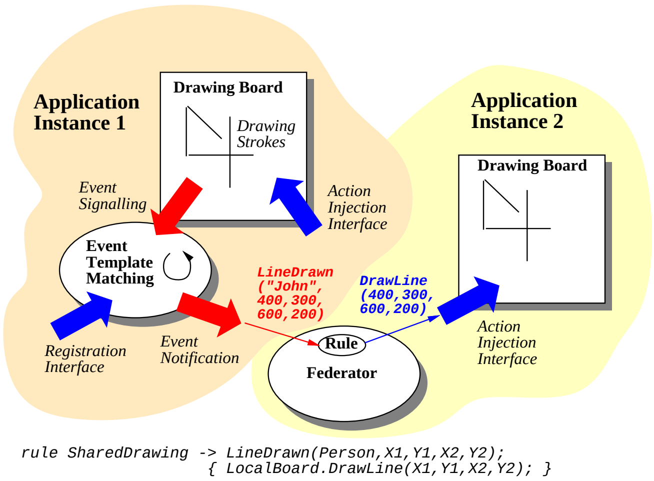

Figure 1: Creating a shared drawing board using events

it can inject an action into the local board to draw a corresponding line. Similarly, other users' federators register with our local board; our board notifies events whenever we draw on it. This approach is more flexible than hard-coding a shared-drawing application; the stand-alone drawing boards can be used in many more contexts than if it was part of a hard-coded system.

1.5 Federating Event Brokers

One special-purpose use of federation is to connect up sets of the world's event brokers to create a global event marketplace. An example is connecting all sites that offer up-to-date financial information. This allows a registration to propagate to all connected domains. Schemes to implement this depend on the compatibility of the connected domains. Possibilities are:

- Event brokers that share the same event taxonomy can pass on queries without translation.
- Event brokers that do not share the same event taxonomy require translation between registrations. We are investigating gateway services to implement this inter-domain translation. These gateway services can also be responsible for translating event notifications from one taxonomy to another. The registration resides with the gateway and the gateway registers on behalf of the user.
- Domains that have some compatibility and similar functionality can be federated using subclassing schemes. For example an agreed super-

class `LocationEvent` may exist in more than one domain and may specialise to different subclasses `badge-locate, login-locate` etc. in different domains.

- A set of domains offering a federated event taxonomy can improve efficiency by providing large intermediate brokers to hold details of the classes offered in the union of the domains.

The ideas are being tested initially within only a few experimental domains, but the intention is to support registration and notification across gateways between real domains. It is not obvious to what extent transparent registration across domains should be a built-in policy and this is an issue we are investigating. Practical aspects of building these services will be to cope with issues such as scale (large numbers of event notifications and large numbers of clients interested in a given event) and the duration of registrations. The fact that hierarchical filtering and access control are intrinsic properties of our design gives the scaling and security constraints one needs to live in the Internet.

1.6 Composite Events

It is often useful to trigger the injection of actions after the detection of a complex pattern of events. Such composite detection can involve parameter checking of event occurrences, state lookup and monitoring for particular event orderings. We have developed a grammar to allow events to be composed. The main operators are as follows:

`-> evexpr`: This statement specifies that the expression `evexpr` must occur after any previous expressions.

`-> evexpr1 - evexpr2`: This composite event expression is detected if the expression `evexpr1` occurs after any previous expressions **without** any intervening occurrences of `evexpr2`.

`=> evexpr1` and `=> evexpr1 - evexpr2`: In addition to the single arrow (`->`), the double arrow (`=>`) can be used. The single arrow indicates interest in only the first relevant occurrence of an event. The double arrow indicates interest in every occurrence of an event.

`evexpr1 | evexpr2`: This expression is detected if either `evexpr1` **or** `evexpr2` is detected. These must be monitored for concurrently, since each expression may be composed of sub-expressions that may overlap temporally.

`evexpr1 ^ evexpr2`: This expression is detected if `evexpr1` **and** `evexpr2` have been detected, in any temporal order.

A composite event service has been developed to support efficient run-time monitoring for composite events. Event rules can incorporate composite events, allowing event clients to monitor for and respond to complex patterns of event occurrence.

1.7 Storage and Retrieval of Events

It is useful to be able to store sessions of events. Sessions can later be replayed in real time, queried, or a combination of both. An example is, if a book is discovered to be stolen from the Computer Lab library, we can replay all location and video motion tracking events recorded in the library from the time the Librarian left to the time he returned. We have implemented an event store service to realise this functionality [13]. The basic storage model conforms to the ODMG database specification [5]. A higher level language has been developed on top of the event store, to provide a convenient interface for advanced temporal queries and replay.

1.8 Support for Building Event Components

We have built a class library to assist a component builder in implementing event functions. Including the event library includes automatically an interface for registration and a module for storage of registration templates. It also includes an internal interface to allow the component to signal internal events; these events are compared to all registered event templates; any matches are signalled to the relevant clients.

The underlying implementation for composite event monitoring is efficient, since it is based on simple state transitions using non-deterministic finite state machines.

1.9 Relationship to Other Work

Our event system is built as a layer above distributed programming systems, such as CORBA, DCOM or Java RMI. It does not require any special support from such systems. We developed our event system initially using a locally developed distributed object system, and later with CORBA.

CORBA [6], Java [9] and Microsoft [11] have defined event services. Our work differs from these as follows:

1. *Open distribution*
We are interested in the use of events in a global Internet. It is important to contrast the differences between events in an open distributed systems context and an intra-application context. The term "event" has long been used to describe asynchronous occurrences of interest within an application; for example, within programs to manage user interfaces and sensor systems. Toolkits to help manage such event-based systems have been developed on a system-specific basis. Recently, for example, JavaBeans has established a standard mechanism for registering interest in events with any Java Bean. This is still proprietary and non-distributed.

Inter-application events are different. We assume a world in which such bespoke event-based systems exist, but see them as "event sources". The sources are able to make selected events available to externally interested clients. Rather than encoding applications monolithically, we can make use of the rich open world of the Internet, in which many different event sources can notify us of many different things. We can find out how other users are interacting with their applications and use this as a mechanism of cooperation; we can find out the location of any user world-wide from monitoring location-tracking sensor systems; we can find the fault in a metropolitan-area network by correlating faults from many different network sources. Existing event sources thus become our building blocks. Our applications can be programmed at a high level, involving registration and response to notification.

Events in a global Internet are more complex than intra-application. There are problems of networking (loss of messages, lack of global ordering at recipient and network failure) and distribution (naming and location of services, client and service failure, authentication and authorisation). There is also the problem of dialect - interpreting events from different heterogeneous domains.

2. *We have fine-grained registration*
Some event systems have addressed distribution; the two main cross-language distributed systems platforms, CORBA and Microsoft ActiveX, have a model of "event channels". In these, clients can register with an event source after which events are multicast to clients, across a network, when they occur. We differ from these approaches in that we support fine-grain registration. Our "event classes" have associated attributes. It is possible to specify one or more parameters in a "registration template" for EXACT MATCH. Any others are treated as wildcards for ANY MATCH.

This distinction is very important in an open distributed context. Consider the example of tracking users world-wide. It is unacceptable to receive every event about every user, even from a single domain. We may want to know only about users in a particular named location (the meeting room); We may want to know only about one particular named user (Mark.Spiteri) or a named group of users (Opera). Our fine-grained registration is the only way to keep network costs and client filtering at an acceptable level.

Another research group with an active interest in event-driven systems is the Distributed Systems Group at Trinity College Dublin. Their approach uses multicast and client-side filtering to decide on whether events should be delivered to applications [14]. We take the view that filtering at the event source, or at a nearby event service, is preferable for scalability. The Distributed Systems Group at Imperial College, London have worked on events in the context of a network management architecture [10].

In summary, other "event systems", such as Java Beans, user interfaces in general and sensor monitoring, fit into our general event model as instances of event sources. ActiveX and CORBA use blanket registration and are not sufficiently flexible and efficient for global working.

2 Event System Case Studies

This section describes three systems frameworks we have built; each allows complex applications to be glued together using events. The frameworks illustrate the broad application areas of events. Respectively, the frameworks support multimedia/CSCW applications, collaborative virtual world applications and environment augmentation for mobile users.

2.1 Conventional Reality (CR) – Component-based Multimedia

The first event-based framework we developed was IMP (Interactive Multimedia Presentation) [2]. The IMP framework enables the rapid development of event-based multimedia applications. IMP scripts allow the definition of event rules, to specify control of run-time activity. This activity takes the form of driving one component, in response to events from another. For example, if the video image of Scarlet is clicked on then display information about Scarlet.

Classes of component in this system include live video and audio sources, stored video and audio sources, video windows, audio sinks, video filters, such as motion detectors, drawing boards and text I/O components. Classes of event include user interaction events, such as drawing lines, typing text, clicking on objects in a video clip and speaking words. Other event classes are associated with the presentation of data, e.g. temporal events and events associated with the presentation of marked frames of video. Although some components are connected together using streams, e.g. a video source and sink, others can be connected together in more flexible ways using events, such as the shared drawing board example described above. A shared text editor can be envisaged, in which each participant uses his/her favourite editor, such as Emacs or VI. If the editors were event-based, this could be enabled through an IMP script. Activity from one instance, such as typing in a shared paragraph, could be used to drive the display of the changes in the other users' editors.

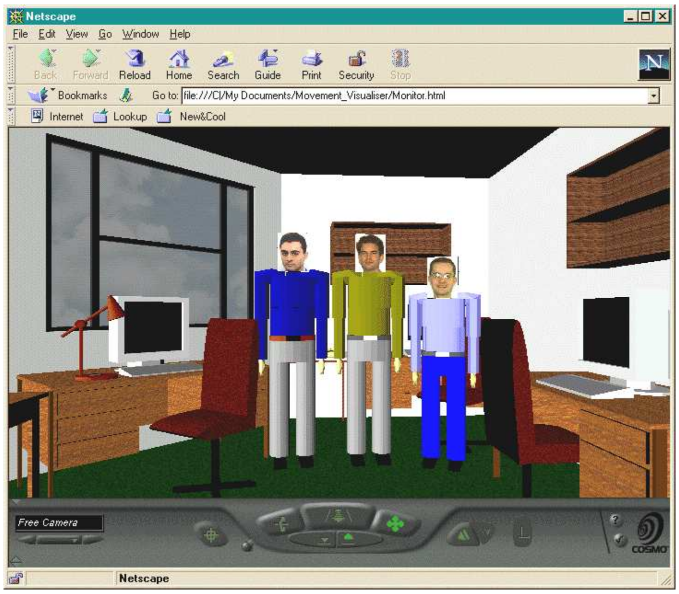

Figure 2: The event-based virtual Computer Lab

2.2 Virtual Reality (VR) – Distributed Virtual World System

In a virtual world, a user steers a course through a 3D graphical environment, interacting with objects populating the world. Making each user's VR world event-based means that multiple users' instances of the same world can be interfaced, without the need for explicit interoperation. If a user moves location in the virtual world, events are notified, specifying the new location. This is analogous to tracking a mobile user in the real world. If a user interacts with an object in the virtual world, e.g. a virtual whiteboard, events are notified to interested parties. This is analogous to interacting with the UI of a traditional application.

We have built an event-based VR system (see figure 2). We used the Java VRML2 toolkit [4] to implement the graphical aspects of the world and the entities populating it. Our prototype world is a representation of our Laboratory. Avatars, representing various members of the department, as well as objects to interact with, are also implemented. In response to injected actions, avatars and objects can be added to the world dynamically; the VRML2 to realise an object is spliced into a world instance at run-time by the controlling Java program. The Java program is also responsible for forwarding events from each user's world to interested clients.

2.3 Augmented Reality (AR) – Mobile Environments for Mobile Users

Augmented Reality means enhancing a user's real-world working environment using computers. Unlike CR and VR, a user is not tied to a fixed physical location. In our system [3], users are mobile; we detect where they are by tagging them. The location of untagged users can be determined, to some extent, by video motion detection. A location service (already introduced) supports registration and notification of user location. User activities can be ascertained through activity capture, e.g. by workstation activity monitoring, monitoring books taken out of the library, documents printed etc. Each monitoring service is event-enabled. Additionally, we know what equipment is near to a user, by tagging some equipment and recording fixed-location equipment in a database.

Applications can make use of this location and activity-awareness. We have developed location-aware mobile agents that can follow users from workstation to workstation. The agents can manage the process of bringing a user's environment around with him/her. We are also deploying this framework in the context of a Home Area Network (HAN). It will be possible to set up event-driven rules by voice commands, e.g. "...if the kids come home, turn on the central heating and switch on the microwave..". Use of event-based infrared pointing devices (wands) allows voice generated rules to be attached to the device a user is pointing at.

3 Federating Conventional, Augmented and Virtual Realities

We have described the principle of event federation. In a component-based system, using a simple

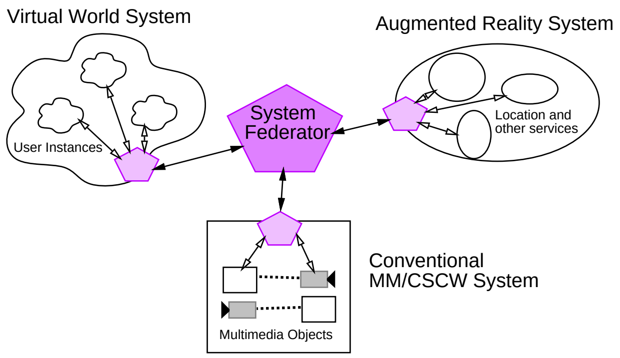

Figure 3: Federating CR, AR and VR

glue component, any event, or ordered combination of events, can trigger any action on any component. This principle is scalable in that it is applicable not just to the federation of small components; active systems, as a whole, can be seen as heterogeneous event-based components and can thus be federated. The federator(s) within each active system can be extended to allow external clients to register interest in system-wide events. If events internal to the system match the registered templates, then they are notified to the external clients.

As part of our experiments, we federated the operation of the three active systems we developed. Figure 3 illustrates the architecture of this federation. Not only can the components within each system be distributed, but the entire systems themselves can be distributed. The federation enables all three active systems to share the same activity space, and for actions in one world to be federated into a corresponding activity in the other two worlds. We conclude this section with three examples of how the systems are federated using rules. We are assuming that the virtual world is a virtual representation of the real environment. In our experiments we represented the Computer Lab as a virtual world.

Example Rule 1
Consider visualising the movement of real-world users in a virtual representation of their environment. In our experiments, as well as *live* events, we used stored sessions. These allow us to visualise what happened sometime in the past, which broadens the application potential to areas such as security and the testing and reviewing of building evacuation schemes. The following rule implements the visualisation, as well as showing mobile users on a map, for the benefit of CR users.

```
rule MoveInVR => AR.FineGrainedLocation(Domain,
Name, "Person", Location, Lat, Long, Alt);
{ CR.Map.UpdatePosition(Name, Lat, Long, Alt);
VR.GetPersonObject(Name).Move(Lat, Long, Alt); }
```

Example Rule 2
Due to the previous rule, if a virtual user enters a location where there is a real user, then the virtual user will see a virtual representation of the real user. The following rule detects this circumstance and displays a representation of the virtual user on the workstation nearest to the real world user, so the real world user knows he/she is being virtually viewed.

```
rule ShowVRinAR => AR.Location(Domain,
PersonB, "Person", Location) =>
VR.Location(Domain, PersonA, "Person", Location)
- AR.Location(Domain, PersonB, "Person", not
Location) ;
{ AR.DisplayAvatar(PersonA, Location); }
```

Example Rule 3
If a CR user is looking through a camera (i.e. has created a video source) in a location where a VR user currently resides (virtually), then display the avatar of the VR user.

```
rule ShowVRinCR => CR.CreatedVideoSource(Domain,
Location) => VR.Location(Domain,
P, "Person", Location) -
CR.DestroyedVideoSource(Domain,Location);
{ CR.DisplayAvatar(P, "LocalWS"); }
```

4 Conclusions

We have shown how event rules offer a flexible method of enabling interoperation of distributed components, via federation. This mechanism works for components of any granularity, ranging from *disposable components* (i.e. created for one session), such as multimedia objects, through long-lived services, such as the location service, up to entire systems. An example of the third case is the federation of CR, VR and AR systems, enabling users to see and interact with each other, if they are in the same physical space. The event system can be built over any distributed systems platform, e.g. CORBA or DCOM. It offers scalability in that it is inherently distributed; event brokers in related domains can be interconnected to cater for the worldwide event marketplace.

References

[1] N Ackroyd and R Lorimer. *Global Navigation: a GPS User's Guide*. Lloyd's of London, 2nd edition, 1994.

[2] J. Bates and J. Bacon. Supporting Interactive Presentation for Distributed Multimedia Applications. *Multimedia Tools and Applications*, 1(1), March 1995.

[3] J. Bates, D. Halls, and J. Bacon. A Framework to Support Mobile Users of Multimedia Applications. *ACM Mobile Networks and Nomadic Applications (NOMAD)*, 1, 1996.

[4] G. Bell, R. Carey, and C. Marrin. The Virtual Reality Modeling Language Specification. Technical report, August 1996. Version 2.0, ISO/IEC CD 14772.

[5] R. Cattel et al. *The Object Database Standard*. Morgan Kaufmann, 1997.

[6] Object Management Group. Event Service Specification. Technical report, March 1995.

[7] A. Harter and A. Hopper. A distributed location system for the active office. *IEEE Network*, 8(1), 1994.

[8] R. Hayton, J. Bacon, and K. Moody. OASIS: Access Control in an Open, Distributed Environment. In *IEEE Symposium on Security and Privacy*, 1998.

[9] Javasoft. Java Beans. *http://java.sun.com/beans/*.

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

[11] Microsoft. Delivering Content to the Web: Creating Active Channels. *http://www.microsoft.com/msdn/sdk/inetsdk/help/delivery/authoring/active_channels.htm*, 1998.

[12] T. Mills, K. Rodden, and K. Moody. COBRA: A New Approach to IR System Design. In *Proceedings RIAO'97*, 1997.

[13] M. Spiteri and J. Bates. An Architecture to Support Storage and Retrieval of Events. In *Middleware'98*, 1998.

[14] G. Starovic, V. Cahill, and B. Tangney. An Event-based Object Model for Distributed Programming. In *Proceedings of Object-Oriented Information Systems*, 1995.

[15] A. Ward, A. Jones, and A. Hopper. A New Location Technique for the Active Office. *IEEE Personal Communications*, 4, 1997.