Using Events to Build Distributed Applications

Jean Bacon, John Bates, Richard Hayton and Ken Moody
University of Cambridge Computer Laboratory
Pembroke Street, Cambridge CB2 3QG, United Kingdom

Abstract

We have extended an Interface Definition Language to handle event registration and notification. Clients register interest in specified classes of events and servers then notify them of any occurrence asynchronously. Event occurrences are identified by parameters which conform to IDL typing constraints and can therefore be used in synchronous method invocations. Methods to handle registration and notification are generic and can be inherited by objects of any class: as a by-product of IDL processing the stubs to handle event creation and decoding are generated automatically. We have implemented a prototype composite event recogniser based on non-deterministic finite state machines. Initial experience with this prototype is encouraging.

1 Introduction

The term "event" is used in many spheres of computing to capture the notion of an autonomous asynchronous occurrence. Here, we are concerned with events within an object-orientated distributed programming (OODP) environment and have designed a unified framework for programming with objects and events.

Current OODP technology is based on objects with typed interfaces together with support for providers to make them available, and for users to locate and invoke them [7, 1]. This object-orientated paradigm is well established but does not capture fully the dynamic, event-driven behaviour of many systems. Some "active" objects monitor continuously for specific occurrences and may then notify interested clients asynchronously. We believe that applications will, increasingly, need to request and receive notification of asynchronous occurrences (events) as well as carrying out synchronous service invocation.

1.1 Current Event Handling Approaches

Handling an asynchronous interrupt in order to avoid polling is familiar from the hardware-system interface. Exception handling at the programming language level comprises client-supplied routines being invoked in response to server-detected events.

In an OODP framework the usual approach to handling asynchronous messages from an active service is the "callback" mechanism. The client must provide a callback interface for each service which can then be invoked by the server. This approach is ad hoc: the client has to declare itself as a service because there is no mechanism for declaring and handling events.

Because different systems use different callback mechanisms a lot of potential for reuse of services is lost. Also, applications are complicated by including different libraries for different services. A uniform mechanism for specifying interest in events and being notified of their occurrence provides a powerful paradigm for building "active applications".

1.2 Emerging Classes of Service

The list below introduces some types of service which are emerging in distributed environments along with the applications which access them:

• *Multimedia Services:* Applications such as Hypermedia can require access to components from different services, e.g. those supporting video, audio, text or pictures. User interaction with one distributed medium component can cause an application to initiate actions on another. For example, clicking on a marked area of a video presentation may cause it to be halted and an audio commentary output.

• *Cooperative Systems:* Cooperative applications support collaboration between a number of users. Occurrences in one application instance must be reflected in others.

• *Services involving Mobility:* e.g. services which utilise the 'Active Badge' system. Each user is provided with a mobile badge device which periodically transmits a unique infra-red signal. Distributed detectors report people's locations to monitoring services. Applications must be able to request and receive information such as when people's locations change.

• *Agent Architectures:* Agents are services which apply "intelligence heuristics" to reporting on their monitoring activities. An example is reporting that mail has arrived only if it is from someone interesting. We envisage agents which apply heuristics to notifications from a range of services.

•*Active Database Systems:* i.e. a standard DBMS augmented with facilities to monitor for certain sequences of database activity. Uses of such monitoring include security checks and constraint enforcement. Standard object invocations by some clients trigger events of interest to others.

All of these service types are 'active' and 'continuous'. They not only carry out user requests but also monitor concurrently for certain events. Several exist for the specific purpose of monitoring. Applications which use these systems must be informed of event occurrences which they can use to initiate further actions.

The activities of different classes of service can often be combined. For example, a video conference might be started at the workstation nearest to a person's current location only if software agents decide that the person is not too busy to deal with it. An architecture is required in which basic services carry out standard event-related functions allowing additional functionality, such as event composition, to be provided by higher level services.

1.3 Our Event Approach

The use of objects has become an accepted basis for distributed programming. Strongly typed interfaces enhance ease of programming, correctness, reusability and portability. However, new applications are exposing the need to extend the paradigm to accommodate dynamic, asynchronous behaviour. Our integration of events into the object framework addresses this shortcoming.

We believe that many applications are developed 'from scratch' because appropriate monitoring facilities are not provided by existing services. To redress this we propose a standard extension to services to incorporate **event specification** (by services), **registration** (by clients at services) and **notification** (by services to clients).

Some applications will require notification of events from multiple services and will need to detect specific patterns of composite event occurrence from these different sources. If services are built to conform to our proposed standards, their clients can compose events into expressions. Since composite event detection is needed in many applications we have built a **composite event service** to meet this requirement.

1.4 Structure of this Paper

In section 2 we describe our model for typed events, registration and notification. Examples of events which services might notify are specified using an Interface Definition Language (IDL) extended for events. We show how a client registers an interest in certain events at a server, and we describe facilities to help clients and services to manage event occurrences. We also describe prototype services we have built using our extended IDL.

In section 3 we illustrate 'composite events': a mechanism for specifying composite monitoring scenarios using primitive events as building blocks. We discuss a prototype composite event service we have developed.

In section 4 we describe prototype applications we have built to explore our ideas. Experimental work is still in progress and in section 5 we discuss our early findings.

2 Event Services

The event approach aims to augment rather than replace current distributed programming methods. We have therefore extended an existing Interface Definition Language (IDL) to handle events. This enables a server to declare, in a strongly-typed way, the events it can notify. It also allows a client to see the server's specification of events and to select those of interest. In this section, event examples will be given in the extended IDL. We have also developed plug-in server and client modules to provide generic event operations, such as registration and notification. In this section, we first describe our event model and then discuss aspects of our prototype implementation.

2.1 Event Classification

In our model an event instance is an object. To declare an event requires a class and the sequence of typed parameters that instantiates a particular event object. An example of an event declaration is

```
Badge : INTERFACE =
  Seen: EVENTCLASS [ badge : BadgeId;
                     sensor : SensorId ];
END.
```

This is taken from the interface `Badge`, declared as part of an active badge service. The service monitors the locations of badge wearers by receiving the latest information from badge sensors. Since an event is declared in an interface description, each of the parameters of the event can be of any type which can be constructed in the IDL. This enables events to share types with interface methods, which is essential when using event parameters in further calls to the service.

Event instances are objects. Event clients and services exchange these event objects. Each event class has its own creation routine, for which parameters must be provided. For example, the badge service can create a `Seen` event object to send to interested clients. An event representing badge 13 being detected by sensor 23 can be created in the following way:

```
e = Badge_Seen(13,23);
```

All event classes are derived from the super-class `Event`. As illustrated below, this allows generic operations to be provided in client and service modules.

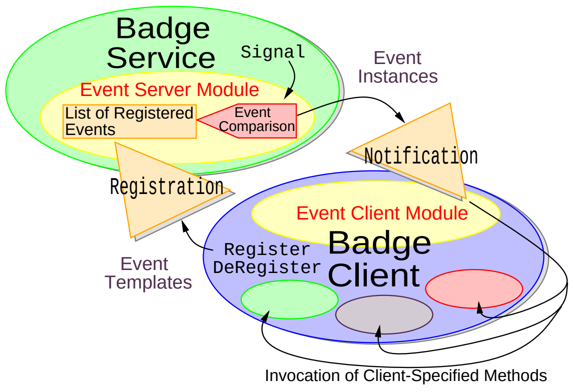


Figure 1: Registration and Notification

2.2 Registration and Notification

This section describes event interactions between event clients and services, illustrated in figure 1.

A service has the ability to *signal* detected occurrences as event instances. Clients will only be told of the occurrence of events (*notified*) if they have *registered an interest* in these events (see below). Taking the badge service as an example, it is constantly receiving information on the location of individuals, which it can signal as events. It can create an appropriate event object (as in `e` above) and signal it in the following way:

```
EventServer.Signal(e);
```

The `Signal` method is a generic operation which is provided by the event service module (see below).

In order to be notified of the occurrence of events, a client must register interest with the relevant service. An *event template* is given to specify the parameters denoting the events of interest. A template is an event object which can be instantiated with parameters of a valid type, and variables. If specific parameter values are given, then an occurrence will be notified only if it matches in these parameters. A variable, given in place of some parameter, indicates a 'wild card' which can match any value in signalled events. Examples of event template object instantiations are

`when = Seen(13,47)`
– notify whenever badge 13 is seen at sensor 47.

`who = Seen(P,23)`
– notify every time a badge is seen at sensor 23.

When a client has created an event template, it must register it with the relevant service. To do this, it must use the generic event client method `Register`. The following example shows the registration of interest in any badge event about anyone seen in any room:

```
template = Badge_Seen(P,R);
EventClient.Register(env,EventHandler,template);
```

The `EventHandler` field in the `Register` method indicates a client method which is to be invoked when an event matching the template (parameter `template`) is notified. The `env` parameter is returned as a parameter to this method, along with the event instance which has been signalled. A header for an event handling method, which must be specified by the client, is expressed in the following way:

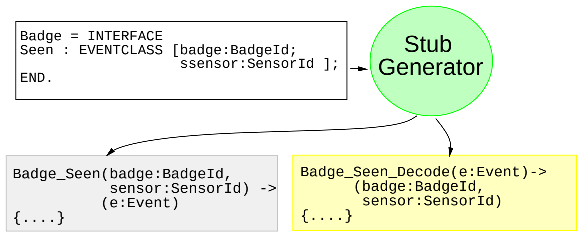
```
void EventHandler(Opaque *env, Event *event)
```

The `env` field is used for client-specific purposes. Its typical use is as a pointer to a client-defined structure that provides extra information when an event is notified. The event object parameter can be queried, via its methods, to decode its instance specific parameters. This can be done in the following way:

```
event.Decode() -> {Instance-Parameters}
```

Since event parameters are defined using the same IDL as service methods, the extracted parameters can be used directly when invoking further interface methods.

If a client no longer requires notification of a particular event template, then it is possible to deregister interest.



```
Badge = INTERFACE
Seen : EVENTCLASS [badge:BadgeId;
                   ssensor:SensorId ];
END.
```

```
Badge_Seen(badge:BadgeId,
           sensor:SensorId) ->
          (e:Event)
{....}
```

```
Badge_Seen_Decode(e:Event)->
      (badge:BadgeId,
       sensor:SensorId)
{....}
```

Figure 2: Examples of automatic stub generation

2.3 Event Timestamps

Event objects also have a timestamp attribute. The event service module sets this attribute when events are signalled. Despite the problems of distributed time, this can be useful in various application-specific circumstances. We discuss it with reference to event composition in section 3.

2.4 Implementation Concerns

In order to support this model of registration and notification in distributed programs we have had to extend current OODP models in the following two ways.

2.4.1 Automatic Stub Generation

The first is the IDL extension to support events. The IDL we selected is that of MSRPC II [8], a locally developed high-performance RPC system. MSRPC II's IDL is an extended version of the ANSA model [1]. The benefit of extending an IDL for events is not just the uniform programming environment, but also the potential for automatic generation of code. In the same way that client and server stubs are generated for the marshalling and unmarshalling of method invocations, the event interface can be used as a specification for a stub generator that automatically creates the event-specific stubs required by event objects. These stubs include the code required to create event object instances and templates and event-specific methods, such as the `Decode` method to access the event instance parameters.

As an example, from the `Badge` interface specified above, the stubs to create and decode event and template instances can be generated (see figure 2).

2.4.2 Client and Server Event Modules

The second extension is the client and server 'plug-in modules' to provide an interface to support generic event operations (such as registration and signalling). Because all event classes are derived from the superclass `Event`, client and server modules can deal with events generically. The stub routines are responsible for instantiating event objects and for accessing their internal state, simplifying the job of the client and server modules.

The server event module manages the functions of registration and notification transparently to the service. It provides registration interfaces for events. Event templates are stored by the module, and whenever an event is signalled by the server code it checks the event against all templates. Any that match are notified to the appropriate client.

The client module provides facilities for handling notifications from a server. It provides a notification interface, a reference to which is specified when registering events with the service. This enables the service to call the client when its templates are matched.

To facilitate the access to registration interfaces requires a binding agent, such as the trader in the ANSA (and MSRPC II) model. In our implementation an event service's registration interfaces are optionally exported to the trader by the event service module. When a registration call is made to the client module it attempts to import an interface of the appropriate type from the trader.

2.5 Prototype Objects and Services

We have developed a number of event services to provide an application development testbed.

• *The badge service* keeps track of the locations of individuals wearing active badges. It is possible to register interest in individuals and sensors.

• *The phone service* manages a phone device. This is connected to an audio service to allow digitised speech to be sent down the phone line. Event classes which can be registered with this service include {`LineFree, LineBusy, PhoneRing, StoppedRinging, KeyPress`}. The `KeyPress` event type is used to detect whether someone on the other end of the phone has pressed one of the phone keys.

• *The login service* monitors for logging into and out of machines. It can also be used to locate people who do not have active badges by seeing whether they are using X displays located within the building. Example event classes from this service are {`Login, Logout, ActiveDisplay, InActiveDisplay`}.

• *The database name service* illustrates how event facilities can assist the construction of an active database. This service complements the active badge service by providing mappings such as that from a badge id to a user's name and from a sensor id to a room name. It also stores the locations of items, e.g. which room holds a particular display. Our service allows `Lookup` to be performed using an event template. It also allows clients to register interest in database items and then to be notified on update. For example, we might have looked up Ken's badge id in order to present it to a badge service to monitor his location. If sometime later Ken gets a new badge then the database can tell us that an update has occurred so that we can amend our monitoring.

•*Interconnectable multimedia objects* have been developed, each of which has registration and notification capabilities [2]. These include text, picture, video and audio objects for device access and filtering. We will discuss how these objects can be used in more complex systems such as cooperative applications in section 4. Another type of object which is described further below is the Whiteboard object. This provides a user with a graphical drawing board. Any interaction using the mouse pointer as a pen results in a line being drawn. Each line drawn also generates an event and it is possible to register interest in these events. Such simple components provide building blocks for more complex applications.

• *The time service* will register a request for an event to be notified to a client at a specified time. This service is used in composite events (see section 3) to enable time offsets to be included in monitoring scenarios.

3 Composite Events

Often, activity within applications is triggered not just by a single event but by a complex pattern of events. Such composite detection can involve parameter checking of event occurrences, state lookup and monitoring for particular event orderings.

Some examples of composite event scenarios are[1]:

- Monitoring for everyone who leaves the building after a fire alarm goes off
- Monitoring for everyone who is still in the building 2 minutes after the fire alarm goes off

In order to remove the burden of implementing this detection on a per application basis, uniform mechanisms for composite event detection can be provided. Such mechanisms have been used widely in the area of active databases, but they are not fully appropriate for our problem. For example, the grammar and finite state machine approach described in [6] does not allow concurrent monitoring. Once a monitor is checking for one sequence, it remains busy until it has finished that detection. Other approaches are more complex to implement [4, 5] and are designed for a centralised database. These do not directly address issues required for a distributed implementation; current work in Cambridge is establishing a composite event model for distributed active databases [9].

We have developed a technique which is simple to implement, in which composite events can be specified easily, which can cope with multiple concurrent detections and which is useful within a distributed environment. The approach is based on finite state machines with acyclic transition graphs. We have enhanced them to allow multiple tokens ('beads') to be active within the machine at any one time. This technique enables parallel detection of multiple occurrences of composite events.

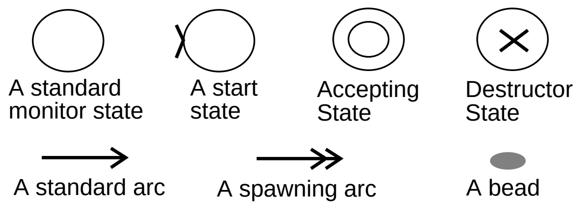


Figure 3: Key to composite event symbols

[1]To implement these examples would require people to wear 'active badges'. They would also require badge and alarm services which have registration and notification facilities as well as a composite event service. We assume these continue to function during a fire.

3.1 Building a Monitor

A monitoring machine is made up of states and arcs (see figure 3). Arcs are labelled with events indicating that the occurrence of such an event allows the arc to be traversed. On initialisation, each machine has one bead in the start state. Each state can be standard, accepting or destroying. If a bead enters an accepting state then the composite event is detected. If a bead enters a destroying state then it is destroyed. A bead will move to a new state when an event occurs which matches the label on an arc leading from its current state. Arcs can be standard or spawning. If the arc is spawning, then the bead is copied. One instance traverses the arc into a new state and one is left in the current state.

Each arc is labelled by a standard event template as described in section 2. The parameters can thus be either variables or values. As a bead traverses the states it records the values of all events which cause it to move. We term this its *path*. The same variable can be used in more than one event template. The first occurrence instantiates the value, and this value is used in the event template of any subsequent occurrences of the variable. This is known as *parameter matching*.

We will illustrate monitoring machines using the examples introduced earlier.

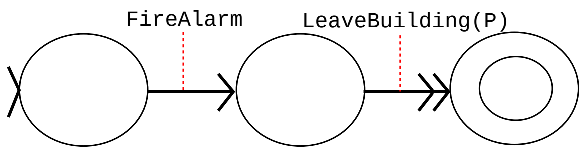


Figure 4: People leaving the building after an alarm

Figure 4 shows the machine to monitor for people who leave the building after a fire alarm goes off. When the `FireAlarm` event occurs the single bead at the initial state is advanced. The arc labelled with the `LeaveBuilding` event is spawning, since we need to create a new bead every time someone leaves the building. The `LeaveBuilding` event has one parameter, the name of the person leaving. If we specified in the event template a specific person, e.g. "Jean", then the link would only be traversed if Jean left the building. In the example, since we specify a variable, each person who leaves will be recorded. The arc leads to an accepting state. Each bead reaching that state contains a person's name within its path.

Figure 5 shows the machine to monitor for all people who have not left the building 2 minutes after the fire alarm has gone off. In this case we have a spawning arc to allow a bead for each person who enters. If a person leaves the building then that bead is killed. If

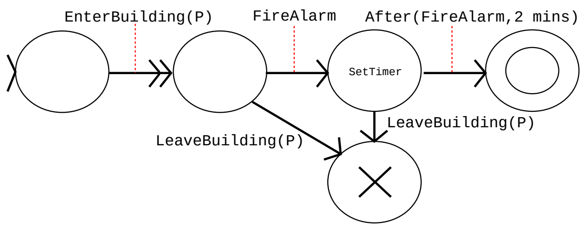


Figure 5: Those still in the building after an alarm

the fire alarm goes off then a timer is set with a time service, which can itself generate events. If the time elapses, then the time event causes any remaining beads to enter an accepting state. The path of each of these beads contains the name of someone who has not left the building.

3.2 Composite Event Definitions

A machine to recognise a composite event can be specified using a simple grammar. When a bead enters an accepting state the machine signals a new event. Composite events can themselves be components of other composite events. A composite event can be defined as a parametrised template, and can therefore fill the role of a primitive event. Thus the monitor in figure 5 can be defined as

```
Define TrappedPerson(P:Person) = ....
```

Parameters can be inherited from sub-events, in this case the value of `P` from the `EnterBuilding` event.

3.3 State Change as an Event

The approach embodied in active databases allows not only synchronous method invocation but also the registration of interest in state changes with notification when these occur. As described in section 2.5, we have adopted this approach in the design of a Name Service Database. `Lookup` may be performed using an event template, also clients may register an interest in order to be informed of state changes.

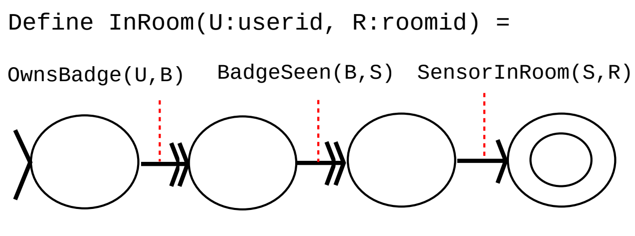


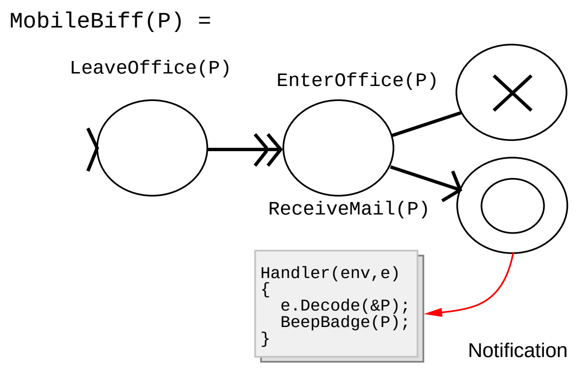
Figure 6: Lookup using event templates

This allows the functionality of parameter matching in composite events to be extended. Certain services may deal with different types of value, for example a badge service may only understand badge ids and sensor ids, and a login server can only understand user ids. Often translation between these worlds is necessary. An example is shown in figure 6. This shows the `InRoom` composite. Two of the arc labels are `Lookup` templates, only `BadgeSeen` is a standard event. A bead is spawned for every user whose id matches the parameter provided for `InRoom` and who owns a badge in the database. Each returns a badge id. A new bead is spawned each time each badge is seen. Finally each sensor id is looked up to obtain a room id which is matched with the parameter provided.

3.4 Composite Event Notification

When a composite event is notified to a client, the full path of the associated bead can be included. However, in the case of parametrised composite events, such as the `TrappedPerson` example, only the required parameters are returned, in that case a `Person`.

In the same way as for primitive events, a client routine can be specified with each composite event registration. On notification, the event can be decoded within this routine and the event parameters used in service invocations.



```
Handler(env,e)
{
  e.Decode(&P);
  BeepBadge(P);
}
```

Figure 7: Mobile Biff

3.5 Distribution Considerations

Distributed environments complicate the model of composite events. Because of network delays and the loadings of individual event servers, events can arrive at a composite service in an order different from their occurrence. To illustrate the problem, consider a machine with the structure shown in figure 7. If `P` is instantiated to "John", this machine accepts a composite event if mail is received when user John is anywhere but his office. This can be used to trigger the beeping of his badge. This is not desirable when he is in his office as it annoys his officemates. In some circumstances the event indicating John has entered his office may be delayed, and, although it occurred first, notification may be received after notification of the event indicating that mail has arrived. In a naive implementation John's badge will beep annoyingly.

A simple solution is to delay the execution of a composite event until it is certain that all relevant events have been received. In the example in figure 7, when the `ReceiveMail` event occurs, the machine could wait to ensure that all pertinent badge events had been received before continuing.

Clock drift between distributed machines makes it meaningless to try to order events that occur very close together. For most applications the delay between events that must be ordered correctly will be greater than the clock drift. We are investigating techniques for quantifying the probable ordering when this is not the case.

3.6 Registration in Composite Events

For each bead, when it enters a state, interest is registered in all events on arcs leading from that state, and the events from the previous state are deregistered. It is common for many beads to be waiting for the same event, and for one event to be on several arcs, so a *registration cache* is used to reduce the amount of unnecessary registration and deregistration.

The cache keeps track of which event templates have been registered, together with a list of beads interested in each. For each event class, a template with a variable parameter is more general than one specifying a value in the same field. Only when a bead requests registration of a completely new event need a registration to the external server take place.

One problem with this approach is that of *event misses*. When a machine has moved into a new state, but before interest in the next event set is registered, an event which would affect the machine can occur. This problem may be handled by pre-registering interest in the next event set before the relevant state is entered. This may have to be done without knowing full details of some fields, thus event templates may contain variables rather than matched parameters.

3.7 Specifying Composite Events

We are at present designing an algebraic language for specifying composite events. The current method for defining a monitoring machine involves listing all states and state transitions. For example, one can specify the machine shown in figure 7 in the following way:

```
S1: LeaveOffice(P)->>S2
S2: ReceiveMail(P)->S4, EnterOffice(P)->S3
S3: KILL
S4: ACCEPT
```

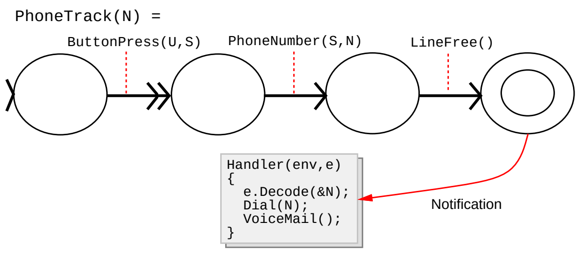
We are using this approach temporarily while we develop a more transparent declarative language.

4 Event-based Applications

In this section we describe two styles of application development using events. These applications use the prototype services described in section 2.5 and the composite event service described in section 3.

4.1 Applications Using Composite Events

Composite events allow powerful applications to be built rapidly. We are at an early stage with our exploration of such applications but we have built several prototypes. We have found it is easy to combine events from different types of service into composite event scenarios.



```
Handler(env,e)
{
  e.Decode(&N);
  Dial(N);
  VoiceMail();
}
```

Figure 8: Phone tracker application

One simple application is the phone tracker. The composite event for this is shown in figure 8. If a user presses their badge button then the phone number nearest to the badge sensor where they clicked is looked up. There is then a check to see whether the phone server line is free. If all is well (the composite event has been detected) that phone is dialled and any waiting mail is spoken.

Another example of a composite event is our version of the "Who" program. This detects who is in the building using both active badge and login services (those without badges might be found at an X display known to be within the building). The machine keeps track of the latest situation regarding who is in the building.

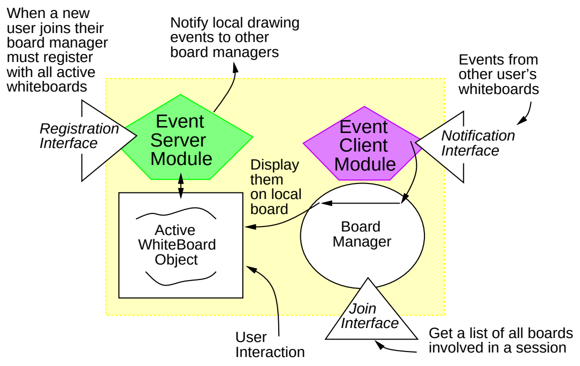

When a new user joins their board manager must register with all active whiteboards
Notify local drawing events to other board managers
Events from other user's whiteboards
Registration Interface
Event Server Module
Event Client Module
Notification Interface
Display them on local board
Active WhiteBoard Object
Board Manager
Join Interface
User Interaction
Get a list of all boards involved in a session

Figure 9: A shared drawing instance

4.2 Cooperative Applications

Events can greatly simplify the development of applications which involve more than one user cooperating in some task. We can illustrate this by a prototype shared drawing application we have developed. This application uses a number of event-driven drawing boards (described in section 2.5), one for each user.

The model we have taken in this application is illustrated in figure 9, which shows the equipment required by one user and how it communicates with others. The model is replicated as opposed to centralised. In other words each user has a full instance with all the functionality of the application rather than all shared traffic going through a central service. This approach scales better and allows for failures, but a centralised approach could be employed as easily.

The first action in starting an instance of a shared board application is to create a whiteboard object to interact with. The first person to start up a session names it. Any other users may join this session by communicating with a party who is already involved. The new member is passed a list of the whiteboard objects involved, with each of which they must register an interest in events. Any drawing events notified from other whiteboards are drawn on the local board by the local instance of the board manager.

In this way the whiteboard objects are used in the application without them having any knowledge of it. This is another example of the reusability potential of an event-based programming approach. Other cooperative applications can be developed in a similar way, e.g. we have also built a prototype video conferencing system in which we use events for floor control.

5 Further Work and Conclusions

In recent years we have developed a multi service storage architecture (MSSA) which supports objects of different media types and is capable of extension through value-adding service (VAS) layers [3]. It is our work on VASs, for example an Interactive Multimedia Presentation support platform (IMP) [2], that has led to the work presented here. We have found that the handling of asynchronous events is fundamental to many emerging applications and that a unified approach to programming with objects and events greatly eases the task of application development.

We have extended an IDL to incorporate events. A service may, through a standard event interface, specify the event classes that it supports, allow clients to register an interest in them and notify clients of their occurrence. A timestamp and parameters specific to the event class identify each occurrence. Event handling is no longer ad hoc and service specific, implemented through callback routines.

We see a widespread need by applications to recognise composite events involving primitive events of different classes. We have implemented a composite event recogniser as a generally available service. During detection of composite events parameter values associated with the component primitive events can be matched, and this facility is enhanced by simple database lookup.

Although at an early stage, these ideas have been tested through prototype implementation and evaluation. We have built a number of services incorporating the event interface and have found application development considerably eased by this standard treatment. We have also built and used a composite event detection service based on non-deterministic finite state machines. At present the service interface for composite events is primitive, being defined by specifying an acyclic finite state recogniser. We are working on a high-level event algebra that will enable us to specify a wider class of composite events.

In summary, we believe that the approach is intuitive and makes it considerably easier to develop applications. The prototype has been implemented by a simple extension to widely used software technology.

Acknowledgements

We are grateful to the EPSRC for supporting this work, under grant GR/J42007.

References

[1] Architecture Projects Management Limited *Advanced Networked Systems Architecture Testbench Implementation Manual*, 1993.

[2] J. Bates and J. Bacon. Supporting interactive presentation for distributed multimedia applications. *Multimedia Tools and Applications*, 1(1), 1995.

[3] J. Bacon, R. Hayton, S. L. Lo, and K. Moody. Access control for a modular, extensible storage service. In *Proceedings of the First International Workshop on Services in Distributed and Networked Environments*, 1994.

[4] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. Technical Report UF-CIS-TR-93-007, University of Florida, 1993.

[5] S. Gatziu and K.R. Dittrich. Detecting composite events in active database systems using Petri nets. In *Proceedings of the 4th International Conference on Research Issues in Data Engineering: Active Database Systems, Houston, Texas*, February 1994.

[6] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18th VLDB Conference, Vancouver, British Columbia, Canada*, 1992.

[7] Object Management Group. The Common Object Request Broker: Architecture and specification. Technical Report 91.9.1, Object Management Group, December 1991.

[8] T. Roscoe, S. Crosby, and R. Hayton. *MSRPC II User Manual.* University of Cambridge Computer Laboratory, 1994.

[9] S. Schwiderski. *Supporting Composite Events for Distributed Active Databases.* PhD thesis, Computer Laboratory, University of Cambridge, 1995. In preparation.