Distrib. Syst. Engng **4** (1997) 96–108. Printed in the UK PII: S0967-1846(97)82271-1

GEM: a generalized event monitoring language for distributed systems*

Masoud Mansouri-Samani†§ and Morris Sloman‡‖

† Networks & Communications Laboratory, Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6QZ, UK
‡ Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, UK

Received 14 July 1995

Abstract. Event-based monitoring is critical for managing and debugging networks and distributed systems. This paper presents GEM—an interpreted generalized event monitoring language. It allows high-level, abstract events to be specified in terms of combinations of lower-level events from different nodes in a loosely coupled distributed system. Event monitoring components can thus be distributed within the system to perform filtering, correlation and notification of events close to where they occur and thus reduce network traffic. GEM is a declarative rule-based language in which the notion of real time has been closely integrated and various temporal constraints can be specified for event composition. The paper discusses the effect of communication delays on composite event detection and presents a tree-based solution for dealing with out-of-order event arrivals at event monitors.

1. Introduction

Monitoring is essential for all aspects of management of communication networks and distributed systems. Managers receive status and event reports which are used to make management decisions which then result in control actions being taken to modify the behaviour of the managed systems. Large systems may potentially generate large numbers of events, so it is necessary to filter and combine low-level events to form higher-level ones in order to raise the level of abstraction for both human and automated managers and prevent them being swamped by event reports. The ability to combine events is also needed for detecting abnormal occurrences for debugging purposes during the development and testing phases of a system.

Certain characteristics of distributed systems make their monitoring more complicated than centralized systems. A typical distributed system consists of a number of cooperating processes executing on different machines and communicating via message passing. Components must be instrumented to generate the required information in the form of status and event reports. Lack of a global clock and variable and unbounded communication delays makes it difficult to correlate monitoring information in order to arrive at a consistent global view of system state. Centralized correlation of event reports is not practical in very large distributed systems.

The most commonly adopted approach for observing and analysing the behaviour of distributed systems has been event-driven monitoring [1–8]. System behaviour is monitored in terms of a set of *primitive* events representing the lowest observable system activity. Users specify *composite* events relating to event sequences, which may span events from several distributed sources as well as *temporal constraints*. For example a composite event can be defined as event e_1 followed by event e_2 within a window of 5 s. A composite event is detected when the specified pattern of events is recognized but distribution may introduce the variable delays making detection of these composite events particularly challenging—how to detect a composite event which is not invalidated as a result of a late-arriving event to which it refers. Current event-based monitoring systems either do not address this problem or provide limited flexibility to deal with it.

Temporal constraints are essential for many management activities in a distributed environment. For example a network manager agent must activate alternate routing tables if it receives a router failure event with no recovery notification within 15 s. Delayed or incorrect ordering of events representing failures, recovery or scheduled time events may result in incorrect management actions.

The monitoring requirements in a large system are not static but evolve as the system changes, there may be a need to dynamically change the patterns of activity which is monitored without having to bring the operation of the system to a halt so there is a need to dynamically change event specifications. For efficiency purposes it would be best to perform monitoring activities as close to the source

* This is an extended version of a paper presented at the ICODP/ICDP '97 Conference, Toronto, May 1997.
§ E-mail address: mms@hplb.hpl.hp.com
‖ E-mail address: m.sloman@doc.ic.ac.uk

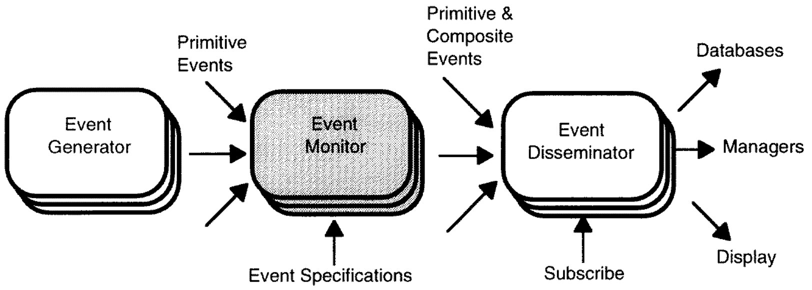


Figure 1. Components of an event monitoring service.

of event reports as possible. This implies the need for a dynamic and distributed event monitoring service.

Figure 1 shows the general overview of an event service, which is an integral part of a generalized, distributed monitoring service [9–11]. Event generators may be instrumented monitored objects or components responsible for polling and observing the activity and changes in the status of monitored objects. Event monitors process event reports from different nodes in the system by performing filtering, composition and notification on these reports based on a given specification. Event disseminators send event reports to clients who subscribe to receive them. Note that event monitors can themselves act as generators for other monitors.

In this paper we describe GEM which is a declarative and interpreted language used for specifying the operation of event monitors [11]. Each monitor contains a command interpreter, and can be controlled interactively by sending it the appropriate GEM scripts. A GEM script declares events classes, rules which define the actions to be taken when an event is triggered and commands to trigger events, disable or enable rules etc. GEM allows the programmer to specify arbitrarily complex (composite) events and guards in the LHS of the rules. We make no assumption about the order of occurrence and detection (arrival) of events. The user can specify a variety of temporal constraints and is able to deal with delays in a flexible and efficient manner using built-in features of the language. The only assumption that is made about time is the existence of a well synchronized global clock.

In section 2 we shall describe how composite events can be specified. Section 3 explains the features of the GEM language followed by example GEM scripts in section 4. Section 5 describes the problems introduced by delays in detecting composite events and shows how it is dealt with in GEM. The implementation of an event-evaluation tree together with a detailed example of the operation of the detection algorithm employed are presented in section 6. Related work is mentioned briefly in section 7.

2. Event specification

2.1. Primitive events

An *event* is a happening of interest, which occurs instantaneously at a specific time. A primitive event forms the smallest building block from which more complex events may be constructed. It is specified as follows:

```
primitive-event-expr :=
   event-id | * | every <time-period-expr>
            | [at] <time-point-expr>
```

An `event-id` identifies the type or the class of an explicit user-declared event (section 3.1 discusses how these event classes are declared). The ‘$*$’ operator is used to mean any user-declared event. The `every` and `at` operators are used to specify periodic and time-point events, respectively (e.g. `every[day+12*hour]` and `at[01:30 29/Feb/98]`). Time-point specifications may be partial by using a wild-card character (e.g. `[Fri 13/-/-]`). For periodic events the start of the period is the time at which the corresponding rule is enabled. Current time is maintained in a global variable called `now` which is continually updated and can be referred to in the guards and rule actions. Temporal events are considered to be implicit and need no explicit declaration before they are used.

Every event report consists of a string of characters which includes the following *default attributes*:

```
event-id [<source-id>] [<timestamp>]
        [(<attribute-value-list>)]
```

where:

`event-id` identifies the event class (for details of how event classes are declared see section 3)

`source-id` is the name of the component which generated the event: this may be omitted when the source is not important for monitoring purposes (e.g. an alarm message from *any* sensor)

`timestamp` is the occurrence time of the event based on a synchronized global clock (optional): if the event generator is a simple device with no access to a clock, then the event monitor inserts the detection time as an estimate of the occurrence time.

An event report may also have a number of application specific attributes. For example,

```
warning "sensor1" [10:30 27/6/95]
        (-4.5, "Temperature too low")
```

2.2. Composite event specification

A composite event expression is a combination of primitive and other composite event expressions using the event operators described in table 1. The operators &, ! and $+$ can be considered as the basic composition operators

Table 1. Event composition operators.

Operator	Explanation
`e1 & e2`	Occurs when *both* `e1` and `e2` occur irrespective of their order
`e + time-period`	Occurs a specified period of time after the occurrence of event `e`
`{e1 ; e2} ! e3`	Occurs when `e1` occurs followed by `e2` with no interleaving `e3`
`e1 | e2`	Occurs when `e1` or `e2` occurs
`e1 ; e2`	Occurs when `e1` occurs before `e2`

which can be used to define the others. An optional guard, which could refer to event attributes, may be used to mask the occurrence of the event.

```
guarded-composite-event-expr :=
   composite-event-expr [when guard]
```

When the event specified by the composite event expression occurs its guard is evaluated, and if true, the event is said to have occurred. Figure 2 shows the structure of a typical event expression.

Guards are Boolean expressions based on event attributes that may be used to mask the occurrence of an event. They typically include C-like expressions relating to the event attributes and are evaluated only when their associated events occur. For example:

```
temp_update when temp_update.t > 10
((e1 when e1.a =< 10) & (e2 when e2.b > 20))
    when (e1.c == e2.d) && (e1.e >= e2.f)
```

In general the class identifier of an event can be used to uniquely identify the type of an event. Different components in the system can generate multiple-event instances of a particular class. The event-class identifier can be used to refer to an instance of that event, where there is no ambiguity, as in the above example. This is not suitable when an event expression refers to a sequence of events of a particular class and it is necessary to refer to an attribute of a specific event instance within the sequence. In GEM an *event variable* may be associated with a primitive or composite event, using the ':' operator, in order to refer to specific event instances within the same rule. For example, in `x:(e ; e ; y:e)`, `x` is associated with the whole composite event and `y` represents the third occurrence of `e` in the sequence.

2.2.1. Occurrence intervals. The concept of real time is important for management purposes to determine causality or to initiate actions at particular times. In GEM, real time has been integrated within the language, allowing the users to talk about absolute and relative time points and time intervals. Intuitively an event occurs at a single point in time, but a composite event may consist of a number of primitive events occurring at different times. Composite events have an occurrence interval which represents the time period during which the composite event is being detected, and they are detected in the context of an event history. The first primitive event, which starts the occurrence interval at time t_1, is called the initiator and the last one, which causes the composite event to occur at time t_n, is called the terminator. Note that t_n is the actual occurrence time of the composite event. The values of t_1 and t_n can be accessed in guards by using the operators `|@` and `@`, respectively. For example, the following event,

$x{:}(e_1 \;\&\; e_2 \;\&\; e_3)$ when $(@x \;-\; |@x) < [3 * sec]$

occurs when three events `e1`, `e2` and `e3`, occur (in any order) and their occurrence interval is less than 3 s.

Operators `@` and `|@` allow us to detect the temporal relationships of two composite events, such as which occurred first or whether they overlap. Figure 3 shows typical temporal constraints that can be checked as part of a guard for composite events with variables `x` and `y`, respectively.

Note that, due to the way guards are evaluated, the following two event expressions have different effects.

```
(i) at[10:00] ; e1 ; at[12:00]
(ii) e1 when ([10:00] < @e1 )
        && ( @e1 < [12:00])
```

In (i) the composite event occurs at 12:00 when the sequence of specified events occurs, whereas the event specified by (ii) occurs every time `e1` occurs between the two given time points and its occurrence timestamp is that of `e1`.

As discussed later, the user can specify how long event information should be maintained to potentially contribute to *new* rule firings before being discarded. Temporal constraints are particularly important because they can determine when to discard this information.

3. GEM scripts

A GEM script may consist of event declarations, rule definitions and top-level commands which can be downloaded to a particular event monitor.

3.1. Event declarations

An event declaration (for both internally and externally triggered events) has the following format:

```
event <event-id>
    [(<formal-attribute-declarations>)]
```

where `event-id` uniquely identifies the class of the event. For externally triggered events it provides an association between the monitoring report and the event triggered by the monitor when that report is received. The optional formal attribute declarations provide the names and types of

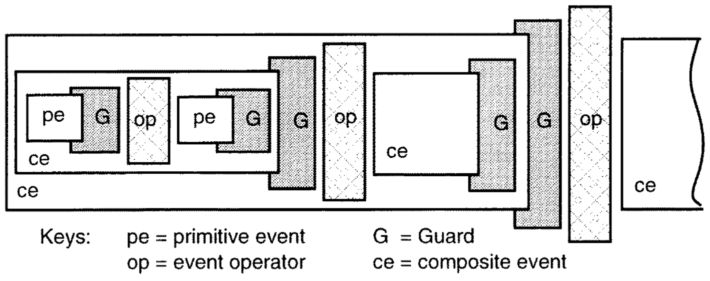


Figure 2. The structure of a typical composite event expression.

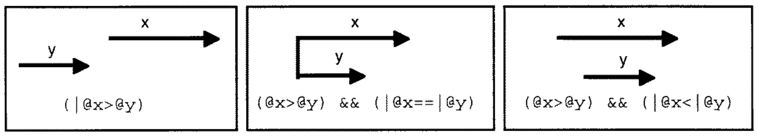


Figure 3. Some temporal constraints based on start and end time points.

application-dependent attributes associated with the named event. Default attributes do not require explicit declaration.

The following types are supported in GEM: `char`, `long`, `double`, `string`, `time`, `itime`. The first four are similar to those in C++. The types `time` and `itime` are used for time-point and time-period expressions, respectively; e.g.

```
event signal
event warning (double temp, itime interval,
               string mesg)
```

Special operators have been provided for accessing default-event attributes. For an event instance `e`, `@e` returns its timestamp and `$e` returns its `source-id`. Event-dependent attribute `a` of an event instance `e` is accessed by `e.a`.

3.2. Rule definitions

As mentioned before, the monitor is passive until an event is triggered. A rule definition specifies a sequence of *actions* to be performed upon the occurrence of an event and has the following format:

```
rule <rule-id> [<detection-window>]
 { <event-expression> ==> <action-sequence> }
```

where `rule-id` is the unique identifier used to refer to the rule, `detection-window` specifies the time window in which the rule operates, `event-expression` specifies the composite event to be detected and `action-sequence` specifies the actions to be executed when the rule fires.

Each time an event is triggered the monitor tries to satisfy as many rules as possible (i.e. the event is ‘seen’ by all rules which are dependent on it). When the event specified in the LHS of a rule is detected, the rule fires and its action sequence is executed. As we shall see, the firing of a rule can cause other rules to fire, by explicit triggering actions. We have adopted a simple execution semantics for rules. All the rules that can fire will fire and their action sequences are executed to the end.

Specification of events and the structure of event expressions were discussed at length in section 2. Section 6 describes the mechanism used for evaluation of such expressions.

3.2.1. Detection window.

The problem with an event sequence (e_1 ; e_2) ! e_3 is how long should the system hold the fact that e_1 has occurred in order to wait for e_2 and what happens if e_2 is delayed so that e_3 actually arrives before e_2 even though it occurred after e_2? A detection window (dw), which can be a system default or specified by the programmer, determines the time period to hold event-occurrence history pertaining to a rule before discarding. It indicates how long the programmer is interested in an event and prevents the constituents of an as yet incomplete composite event occurrence from lingering in the system forever. Periodically the event history for each rule is checked and those events which are outside the window are discarded. Conceptually each rule operates on events placed on a conveyor belt. These events fall off the edge at the end of the window, as shown in figure 4. The checking period for the detection window depends on the rate of event occurrences and the available memory to store event histories.

3.2.2. Event actions.

One or more actions can be specified in the RHS of a rule to notify, forward, trigger, enable and disable events.

(a) Notify action. The notify action can be used to explicitly generate and report a new instance of a user-declared event. It has the following format:

```
notify <event-id> [(<attribute-value-list>)]
```

where `event-id` specifies the class of the event to be notified, and `attribute-value-list` is the list of values for the specified event attributes.

The attributes are similar to procedure parameters so position, type and number of the supplied and expected

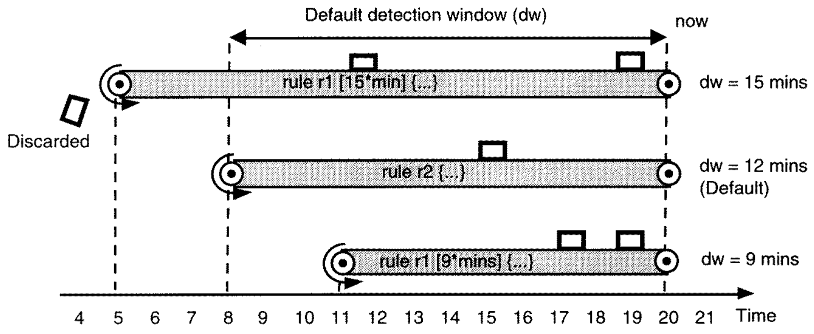


Figure 4. Conveyor belt analogy of a detection window.

attribute values must match exactly. The event timestamp is set to the occurrence time of the composite event in the LHS of the rule and its `source-id` is set to the monitor's own identifier. The newly generated event is not visible internally (i.e. other rules will not see it). Only 'the outside world' will detect its occurrence. For example, given the following event declaration:

```
event e₁
event e₂(long l, double d, string s)
```

we can write:

```
rule testrule {
      <an arbitrary event expression> ==>
          notify e₁;
          notify e₂(10, 3.4, "Valve Failure")}
```

As shown in this example new events with varying levels of details can be generated for different managers.

(b) Forward action. The forward action is used to report user-declared primitive-event occurrences, which contributed to the rule firing, and is particularly useful for filtering purposes. A forwarded event has the original timestamp, `source-id` and attributes whereas the notify action generates new events with the monitor's timestamp and `source-id`

```
forward (<event-id>|<event-variable>)
```

An event class identifier can be used to directly refer to the corresponding primitive-event occurrence. For example, given a predeclared event, `link_failure`, we can write:

```
rule failure { link_failure when ... ==>
               forward link_failure }
```

An event variable can also be used to rewrite the `failure` rule in the following more compact form:

```
rule failure { x:link_failure when ... ==>
               forward x}
```

An event variable can be used to forward specific instances of an event. For example, given the predeclared events e_1 and e_2, the following rules can be defined:

```
rule R₁ { e₁ ; e₂ ==> forward e₂ }
```

R_1 reports the occurrence of an e_2 after an e_1 has occurred.

```
rule R₂ { e₁ ; e₂ ; x:e₂ ==> forward x }
```

R_2 reports the second occurrence of an e_2 after an e_1 has occurred.

```
rule R₃ { e₁ ; x:(e₂ & e₃) ==> forward x }
```

R_3 reports the occurrences of an e_2 and an e_3 (in any order), after an e_1 has occurred.

(c) Trigger action. A user-declared event within the monitor can be triggered by simply 'calling' it.

```
<event-id> [(<attribute-value-list>)]
```

This is similar to the `notify` with respect to setting attributes. For example, given the following event declaration:

```
event signal_loss(long a, string m)
```

We can write:

```
rule r₁ {<some composite event> ==>
        signal_loss(20, "Link Failure")}
```

The newly triggered event is visible to all the other enabled rules within the monitor. The 'outside world' cannot detect its occurrence unless explicitly reported by another rule using the notify or forward action. This provides a convenient abstraction mechanism. An internal event can be defined, triggered and notified in terms of other internal and external events and would typically inherit attributes from the events referenced in the LHS of the rule. An event triggering could in turn cause a number of rules to fire and other events to be triggered.

```
rule r₂ { ...; signal_loss;...  ==> ...}
```

The order in which the rules are fired is implementation dependent, but all the rules which can fire, will fire. A rule cannot directly 'see' the events that it triggers, e.g.

```
rule r₂ { ...  ; signal_loss ; ...  ==>
          ...; signal_loss ; ...}
```

This prevents recursive firing of r_2 although if another rule is triggered by r_2 and it, in turn triggers `signal_loss`, this can cause recursive triggering. Circular-event dependencies may span several rules, and the programmer is responsible for making dependency checks to identify and avoid recursive and infinite triggering of events, and firing of rules.

(d) Enable and disable actions. GEM rules may be enabled or disabled using the following actions, respectively:

```
enable [<rule-id>]
disable [<rule-id>]
```

where `rule-id` specifies the identity of the rule, or if it is omitted, the identity of the current rule is used.

When an event is triggered only the enabled rules are examined. All rules are disabled by default and have to be enabled explicitly. Once enabled, a rule will remain so until it is specifically disabled. As we shall see the event evaluation tree of each rule may maintain historical data, which may ultimately contribute to its firing. These data are automatically discarded when the rule is disabled. These actions provide a powerful mechanism for concentrating on specific aspects of system activity under certain conditions and filtering unwanted information. For example, enabling all the rules related to configuration and performance monitoring and disabling all the fault-monitoring rules during a specific period.

In the following example, rules A and B refer to the user-declared events e_1 and e_2.

```
rule A { e1 ==> forward e1; e2; disable }
rule B { e2 + [5*sec] ==> enable A }
enable A
enable B
```

The effect of these rules and the `enable` command is to report the occurrence of an event of type e_1 and to ignore all other occurrences for the next 5 s. This is particularly useful in the fault-management area and in particular the alarm-correlation application in which a large number of similar alarms may be generated as a result of the same fault. All the duplicates can be removed for a specified length of time.

3.3. Control commands

GEM provides a number of top-level control commands which are independent of rules and are used for controlling the operation of the monitor. Some of these commands are similar to actions that can be performed in the RHS of the rules.

(a) Event trigger command. As we described in section 3.2 an event can be triggered in the RHS of a rule. A similar effect can be produced at the top level by simply calling the event with appropriate attributes. The format is similar to that of an event report described in section 2.1:

```
event-id [<source-id>][<timestamp>]
         [(<attribute-value-list>)]
```

This is useful for debugging purposes, checking event dependencies, or initializing the state of the event evaluation tree. For example, given these event declarations:

```
event e1 ( long l, string s, itime it )
event e2
```

the following can be used as valid event-triggering commands:

```
e2
e2 [8:30 15/5/95]
e1 (3, "some string", [5*hour+30*min])
e1 [10:30:0.0 Mon 15/5/95]
    (3, "some string", [5*hour+30*min])
e1 "s10" [10:30 15/5/95]
    (3, "some string", [5*hour+30*min])
```

(b) Enable and disable commands. The enable and disable are similar to the corresponding actions described in section 3.2. They can be used to activate or deactivate rules. The format is the same with the exception that the wild-card character '$*$' can only be used at the top level to enable or disable all the existing rules, e.g. `disable*`.

(c) Stop command. The stop command is used to halt the operation of the monitor. It results in the discarding of all the event information held by the monitor.

(d) $\sim$ (remove) command. As monitoring requirements change the set of declared event classes and defined rules will have to change. The remove command '$\sim$' allows us to delete those which are no longer needed. An event class can be removed using a command of the following form:

```
~ <event-id>
```

The specified event class can only be removed (undeclared) if there are no rules, which specifically refer to it. A rule can be removed using the following command:

```
~ rule <rule-id>
```

As a result of this command the event information maintained by the rule is discarded and the monitor removes the rule.

(e) Load command. The load command can be used to instruct the monitor to read GEM commands from a specified file, e.g. `load "testfile.mi"`.

4. Examples of GEM scripts

Example 1. The following rule causes a `disaster` report to be generated when the occurrence of a `power_cut` event is detected whose timestamp is between 24/12/96 and 2/1/97 and whose source is the main generator. The detection window is set to 10 min.

```
rule major_disaster [10*min] { x:power_cut
   when ([24/12/96] < @x) && (@x < [2/1/97])
       && ($x == "main generator")
       ==> notify disaster }
```

Example 2. The following rule can be used to generate a `shut_down` notification to the system administrator to shut down all the machines in order to protect them from the Friday 13th virus! The event expression is a partial time-point specification.

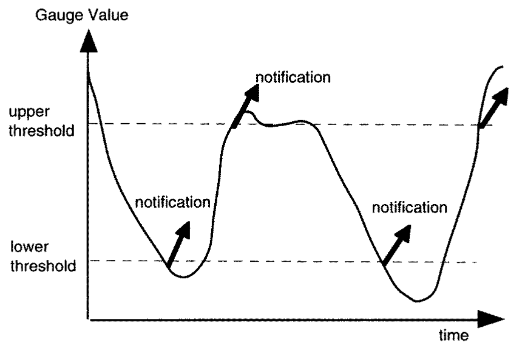


Figure 5. Gauge threshold operation.

```
event shut_down (string reason)
rule protection { [Fri 13/-/-] ==>
        notify shut_down("Friday 13th") }
enable protection
```

Example 3. The following script specifies that a gauge threshold notification must be generated when the value of the val attribute of a gauge_changed event crosses low or high thresholds as shown in figure 5. The upper and lower thresholds are 10 and 5, respectively. The interval between these thresholds is called the hysteresis interval. The gauge value may oscillate about either of the thresholds and consequently multiple notifications may be generated. The enable and disable actions in the rules are used to force such notifications to be generated alternately when the value crosses the thresholds.

```
event gauge_changed(double val)
event upper_exceeded
event lower_exceeded
rule gauge_rule1 { x:gauge_changed
   when x.val >= 10 ==>
         notify upper_exceeded;
         enable gauge_rule2; disable }
rule gauge_rule2 { x:gauge_changed
   when x.val <= 5 ==>
         notify lower_exceeded;
         enable gauge_rule1; disable }
enable gauge_rule2
```

Example 4. The following rule causes a warning event to be triggered internally for a pair of (temperature, pressure) alarms if both are received from the same source and are generated within 10 s of one another. The composite event occurrences are detected in chronological order given a maximum expected delay of 15 s.

```
event temp_alarm(double t)
event press_alarm(double p)
event warning(string sensor_id;
              double t; double p)
rule warning [15*sec] {
   ((x.temp_alarm & y:press_alarm)
         when (($x == $y ) &&
         (( @x-@y )< [10*sec] ))) + [5*sec]
         ==> warning($x, x.t, x.p)}
enable warning
```

The detection window of 15 s takes into account the maximum delay and the temporal constraint specified in the guard. Although the events specified with variables x and y may occur in any order, the interval specified by @x-@y is always positive, no matter which one occurs first.

The scheduled time event (...+[5*sec]) represents the period of time that the rule must wait after the detection of the subevent before it can fire. During this interval if an event arrives which chronologically precedes one of the existing pairs of events and a successful match can be found, the algorithm automatically picks it up instead. Therefore the older matching pair is correctly recognized as a composite event given a maximum delay of 5 s.

The event-dependent attributes of warning event are set using the attributes of the subevents in the LHS (i.e. the source-id and the t attribute of the temp_alarm event and the p attribute of the press_alarm).

Example 5. The following script is based on a slightly altered version of an example given in [3]. The rule EXPECTED_ALARM_RULE detects that a carrier group ALARM_A on a network element occurred and during the following 1 min interval an expected carrier group ALARM_B did not occur at the same network element. On detecting such an event, it is forwarded by the monitor.

```
event ALARM_A
event ALARM_B
rule EXPECTED_ALARM_RULE {
       ({x:ALARM_A ; z: (y:ALARM_A
         + [1*min])} ! w:ALARM_B)
              when (?z && (x == y)) ||
                   (?w && ($x == $w))
                   ==> forward x }
enable EXPECTED_ALARM_RULE
```

The equality between event variables can be used to see if they refer to the same event occurrence(s). A guard associated with event expressions constructed using ‘|’ or ‘!’ event operators may refer to the attribute of an event which has not actually occurred. To cope with this situation the operator ‘?’ can be used to check and see if the event has actually occurred before referring to its attributes. In the above example, events z or w may occur after x. This is conceptually similar to checking that a pointer is not zero before using it in C/C++.

5. Dealing with delays

Variable communication delays in a distributed environment means that events may be detected some time after their occurrence and possibly in a different order. Event composition techniques, commonly used in centralized systems, may not detect certain composite events or may result in erroneous detections and consequently unwanted rule firings. For example, the sequence of event instances (e_1 e_3 e_2) may be detected by the monitor in the following order (e_1 e_2 e_3). Given the delayed sequence of events rule r_1 will erroneously fire on e_2 and its action sequence is executed before the cancellation event e_3 arrives as shown in figure 6.

rule r_1 { {e_1 ; e_2} ! e_3 ==> ... }

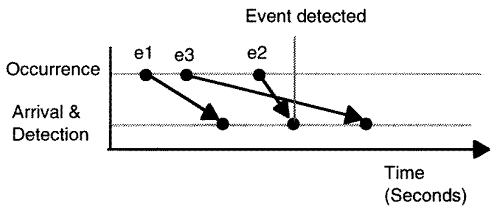


Figure 6. Occurrence and detection of events.

A detection is valid when no late-arriving event could invalidate or alter the detected composite event. Valid detection is impossible in the absence of explicit assumptions about the possible orderings or the maximum possible delays involved†. In many cases the best that can be done is to assume a maximum tolerated delay and to discard late arriving events.

5.1. Uniform delaying technique

Given a well-synchronized global clock and a known maximum possible communication delay D, one obvious approach is to uniformly delay *every* event by D time units before it is passed to the detection stage [12]. This is illustrated in figure 7.

In the ordering stage every newly arrived event report e with occurrence timestamp t, is inserted in its correct place in a global ordered event history whose elements satisfy the following constraint: $(t + D) < C$, where C is the current time. On every clock update the segment of the history which has been delayed enough is forwarded to the detection stage where an existing composite-event detection algorithm (e.g. based on Petri nets, evaluation trees or state machines) can be used to perform the detection. The application of this technique to our earlier example is shown in figure 8. D is assumed to be 6 s, which is the time each event in the unordered sequence has been delayed. The ordered sequence can now be used for a valid-event detection.

This approach has several disadvantages:

pessimistic, introduces possibly unnecessary delays in event detection by holding all the events for the maximum time delay,

inefficient, every event is ordered with respect to all the others in the maintained history even when temporal constraints are not important for that event in the ultimate detection stage,

inflexible, in some circumstances no explicit ordering may be necessary, for example events generated locally or in a local area network that provides ordering guarantees. This knowledge about specific events cannot be used.

5.2. Event-specific delaying technique

The approach that we have adopted is to push the knowledge and the capability of dealing with delays into the detection stage itself. In GEM a tree-based composite event-detection algorithm is employed which makes no assumptions about communication delays or the order of arrival of event reports. Every new event is inserted in its correct place in an internal hierarchical history, maintained by an *event evaluation tree* (the structure of the tree and the history it maintains is described in section 6). The algorithm performs the ordering or imposes delays, when needed and as specified by the programmer. In GEM delays are dealt with at two complementary levels.

(i) A rule maintains an ordered hierarchical event history over a time span defined by the rule's detection window. A new event is inserted in its correct place in this history provided it is within the detection window, which represents the maximum absolute delay that the rule can cope with.

(ii) A scheduled time event can be used as an explicit terminator event, to deliberately extend the detection interval. This value represents the maximum delay that can be accommodated relative to a specific event.

These can be viewed as absolute and relative timeout facilities. Given the previous example, with $D = 6$ s, one can write a `rule` r_2 as:

```
rule r₂ { ({e₁ ; e₂} !  e₃) + [6*sec] ==> ...}
```

Note that this rule uses a default detection window, which must be greater than 6 s. `Rule` r_2 can now cope with the sequence e_1 e_2 e_3 as shown in figure 9.

At time 5 on the arrival of e_2 the event specified by the subexpression `({`e_1 `;` e_2`} !` e_3`)` is detected by the algorithm. This causes a relative time event to be scheduled for time 10 (e_2`'s occurrence timestamp + delay`). The information about the subevent is maintained in the event evaluation tree while we wait for the scheduled event. At time 7 the cancellation event e_3 arrives. The algorithm makes sure that the subevent and the scheduled time event are automatically cancelled so that the rule is not fired erroneously. A scheduled time event can be associated with any event specified in GEM, and therefore a similar control over delays can be exercised at both primitive and composite event levels. It provides the user with unlimited flexibility and a finer grain of control in dealing with delays‡. This is particularly useful when events referenced in a rule may arrive from sources with different expected delays or where there is a known temporal relationship between event occurrences (e.g. events e_1 and e_2 will never occur more than 2 min apart). Furthermore the detection time is not affected by the external buffering strategy employed at a separate ordering stage.

6. Implementation

We have adopted a tree-based mechanism for composite-event detection. It allows the event evaluation algorithm to deal with out-of-order sequences of event arrivals in an efficient manner.

† Even given certain guarantees about the maximum amount of delays, failures may prevent a valid detection.

‡ It is not always possible to determine a maximum value for delays therefore the specified time period may represent the *tolerated* delay interval rather than an *expected one*.

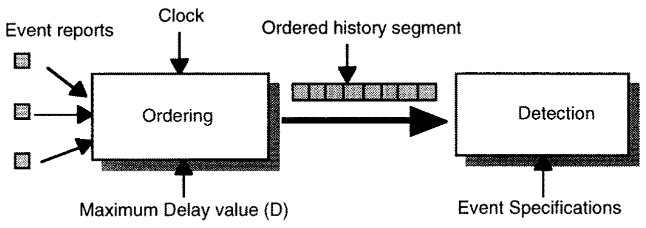


Figure 7. Uniform delaying technique.

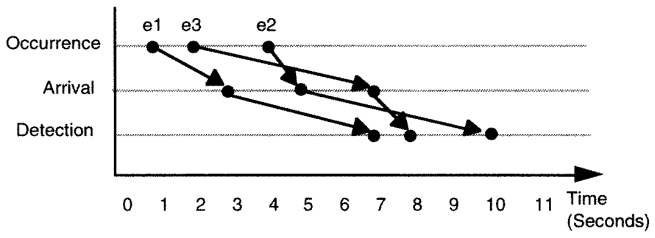


Figure 8. An example of the uniform delaying technique.

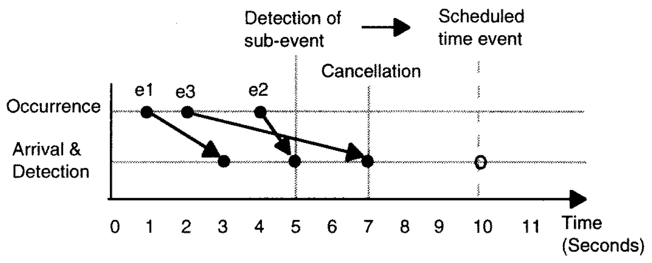


Figure 9. An example of the event-specific delaying technique.

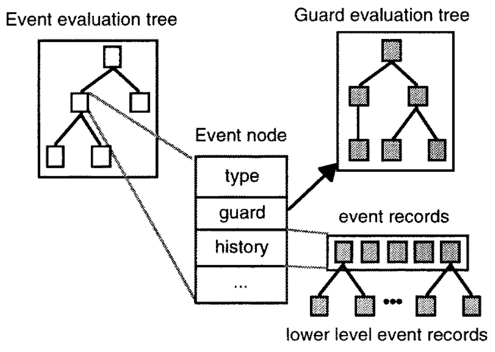


Figure 10. The structure of the event-evaluation tree.

6.1. Event-evaluation tree

When a rule is interpreted, the event expression in its LHS is used to create an *event-evaluation tree*, as shown in figure 10. The structure of the tree closely matches that of GEM's event expressions (figure 2) where the leaves and internal nodes represent guarded-primitive- and composite-event expressions, respectively.

Each node has three main attributes:

- *type*, identifying the type of the node (primitive, +, &, |, etc)
- *guard*, representing the predicate that must be evaluated when an event is detected
- *history*, an ordered history of event records.

Every time a primitive event is triggered, the relevant leaves are informed of the occurrence. Each leaf node enters a corresponding event record in its history if its guard is satisfied and informs its parent node of the existence of the new record†. Depending on its semantics each parent in turn uses the new record from its child, its associated guard, and possibly the records in the histories of its other children (if any) to generate as many new event records as possible. The parent may generate and add to its history several such records. The process continues until the rule fires or no new records can be generated.

Each record may depend on and refer to one or more other records at lower-level nodes and can be viewed as an event-instance tree, the leaves of which refer to primitive event instances. The histories maintained by the event-evaluation tree form a hierarchical history, which avoids unnecessary orderings. Only related, rather than all events, are ordered. This history is updated in a consistent way as new records are inserted. Duplication of event information within each rule is kept to a minimum as event records only maintain pointers to other event records. A guard evaluation tree is associated with each event-evaluation

† Multiple-event records in the histories of different leaf nodes (possibly belonging to various rules) may refer to the same event occurrence.

node. This allows incremental evaluation of the guards and therefore early filtering of events which in turn reduces unnecessary computations. As we saw in section 5.2, one or more records may be invalidated as a result of late-arriving events and have to be cancelled. Cancellations may also occur as a result of rule firings as shown in the following example. The structure of the tree allows such cancellations to be dealt with easily and efficiently.

Dealing with delays at the detection stage adds to the complexity of the event-evaluation tree and the detection algorithm. In addition to evaluation of event expressions, the algorithm must perform event ordering and cancelling invalidated partial results. This complexity is unavoidable for remote monitoring but unnecessary for monitoring events generated locally.

6.2. An example of composite event detection

Figure 12(*a*) shows the event-evaluation tree and its hierarchical history created for a rule of the form:

```
rule test_rule [11*min]
     {(a & b) ; ({c ; d} ! e) ==> ... }
```

The notation $\mathtt{e}_\mathtt{t}$ is used to refer to an event instance of class `e` with the occurrence timestamp `t`. Assume that each unit on the time line represents 1 min and that the rule was enabled at time 5 with a detection window of 11 min. Figures 12(*a*)–(*e*) illustrate the changes in the hierarchical history as the sequence of event instances, shown in figure 11, is detected.

Figure 12(*a*) shows the state of the event-evaluation tree and its leaf node histories at time 13, after the detection of the following primitive events: $\mathtt{c}_6$ $\mathtt{b}_9$ $\mathtt{d}_{10}$ $\mathtt{a}_{11}$. The figure also shows the event records (representing composite sub-events) which have been inserted in the histories of the corresponding internal nodes. As we have seen any event whose occurrence timestamp falls outside the rule's detection interval is ignored by the rule otherwise a corresponding event record is inserted in its correct place within the relevant history and can be used for evaluation.

At time 14:00 the event $\mathtt{e}_7$ is detected (figure 12(*b*)). Given the semantics of the '!' operator, it causes the composite event record ({$\mathtt{c}_6$; $\mathtt{d}_{10}$} ! `e`) to be cancelled. It illustrates one of the problems that delays introduce in the detection process. Some of the intermediate results may be invalidated due to late arriving events and would have to be cancelled. Any other higher-level event records depending on the cancelled event would also have to be cancelled.

At time 15:00 event $\mathtt{d}_{14}$ is detected (figure 12(*c*)) and a corresponding event record is inserted in `d`'s history. It does not result in the generation of any higher-level event records. At time 16:00 event $\mathtt{a}_8$ is detected which in turn causes the detection of the composite event ($\mathtt{a}_8$ & $\mathtt{b}_9$), as reflected in the hierarchical history in figure 12(*d*). Note that the new event records are inserted in their correct occurrence place in this history, irrespective of detection order.

Figure 12(*e*) illustrates the detection of event $\mathtt{c}_{13}$ at time 17:30 minutes. This has resulted in the detection of composite events at various levels, and in particular the occurrence of the top-level composite event whose constituent primitive events are ($\mathtt{a}_8$, $\mathtt{b}_9$, $\mathtt{c}_{13}$, $\mathtt{d}_{14}$). The full curve is used to show records participating in this occurrence. This occurrence results in the firing of the rule and the execution of its action sequence, which may use the event information. Once all the actions are executed all the records contributing to this firing are marked for consumption. Consumed records will not be able to participate in any further evaluations and rule firings. As a result any higher-level records which depend on consumed records would have to be cancelled (e.g. the crossed record in '&' node's history with timestamp 11, because it depends on the consumed-event record representing $\mathtt{b}_9$). Note that the detection algorithm selects events in a chronological order (e.g. choosing $\mathtt{a}_8$ rather than $\mathtt{a}_{11}$) and that $\mathtt{c}_6$ has been discarded. There may be no obvious relationship between the occurrence interval (8–14) and the detection interval (8–17:30). A scheduled time is used as a unique terminator to link the two intervals and deliberately extends the detection interval to deal with late-arriving events.

7. Related work and conclusions

Event monitoring, in particular composite-event detection, has received much attention, particularly in the area of active databases [13–16]. They provide rich notations for specifying composite events but due to their centralized nature the proposed detection algorithms are unsuitable for event composition in a loosely coupled distributed environment. In particular these algorithms assume that events are detected at the time or in the order that they occur. As we have seen, such assumptions cannot be made in a distributed environment and, therefore, using such algorithms may lead to erroneous and alternative detections and hence unwanted rule firings.

Various mechanisms for composite-event detection have been employed. Samos [17] uses Petri nets for event detection. Ode [16] uses a finite automata. A similar mechanism is employed in Meta [18] for evaluating temporal expressions. The mechanism used by [19] is based on finite-state machines with enhancements to allow detection of concurrent composite events. We have adopted a tree-based mechanism as it seems to be the most suitable for on-the-fly detection of concurrent composite events in presence of delays. Shim and Ramamoorthy [12] and Chakravarty *et al* [13] use a similar mechanism, but our event-evaluation tree maintains a hierarchical history which allows us to deal with out-of-order event arrivals in an efficient manner.

A number of systems use a centralized correlation approach to filter and analyse events generated from distributed sources [3, 19–21]. The problem with this approach is that all events have to be sent to the correlation server, which can cause considerable network traffic in large systems. Our approach allows filtering and composition of events to be distributed and take place close to the source of these events and so it scales for very large systems. A similar approach has been adopted by [6] who provide the ability to define dynamic configurations of the

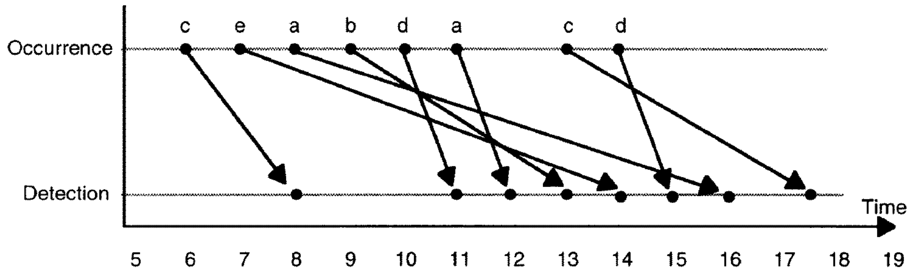
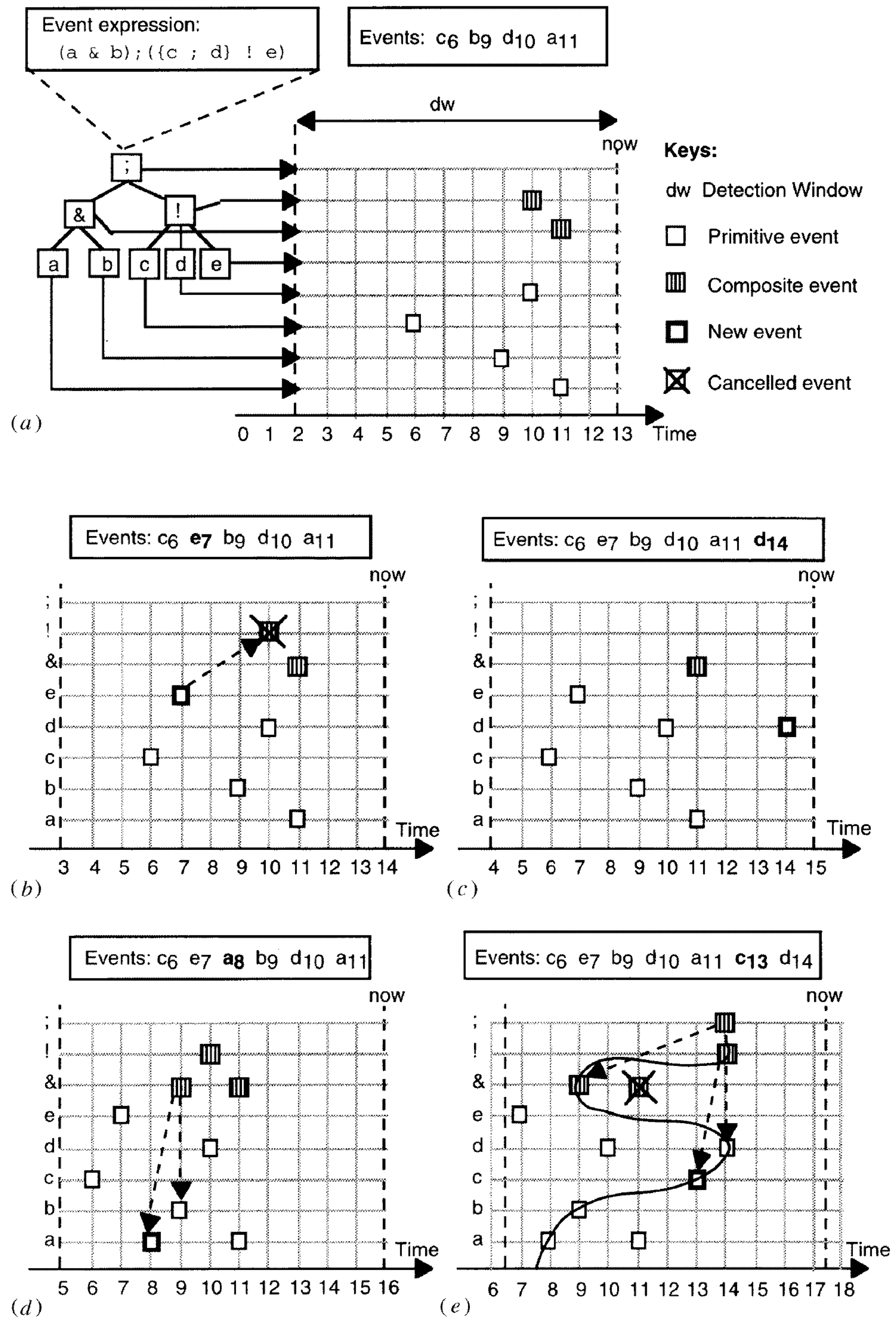


Figure 11. Occurrence and detection times of a delayed-event sequence.



Event expression:
(a & b);((c ; d) ! e)
Events: c_6 b_9 d_{10} a_{11}
dw
now
Keys:
dw Detection Window
Primitive event
Composite event
New event
Cancelled event
Time
(a)
Events: c_6 e_7 b_9 d_{10} a_{11}
(b)
Events: c_6 e_7 b_9 d_{10} a_{11} d_{14}
(c)
Events: c_6 e_7 a_8 b_9 d_{10} a_{11}
(d)
Events: c_6 e_7 b_9 d_{10} a_{11} c_{13} d_{14}
(e)

Figure 12. (*a*) Event-evaluation tree at time 13, (*b*) time 14, (*c*) time 15, (*d*) time 16, (*e*) time 17:30.

monitoring system, but it is not certain how event detection is performed in the presence of delays.

Event-driven monitoring has been used extensively for observing and analysing the behaviour of distributed systems [1–6, 8, 18]. It is used for a variety of tasks such as debugging, performance analysis, program visualization and management. None of the current approaches provide a flexible and user-defined facility for dealing with delays when detecting composite events in a distributed environment.

The CORBA event service [22] only deals with dissemination of events data between multiple suppliers and consumers based on two communication models: supplier push or consumer pull. It does not provide the sophisticated filtering and correlation capabilities described in this paper.

GEM provides a generalized-event monitoring notation that permits user-specified filtering and composition scripts to be dynamically loaded into distributed-event monitoring components. GEM uses scheduled time events and default or user-defined detection windows to cope flexibly with the problems that variable communication delays introduce in detecting composite events [11]. The GEM monitor has been implemented in C++ within the REGIS/DARWIN environment that provides the communication and configuration platform [23]. The instantiation, binding and configuration of such monitors and how GEM scripts can be downloaded into them is described in [10, 11]. A CORBA version of this service has also been implemented [24].

The GEM event monitoring system is being used to detect complex-event sequences and convert these into simple events which trigger obligation policies within managers which then perform the management actions specified by these activities [25, 26]. The event monitors have therefore been restricted to performing very simple activities related to triggering or notifying events. We have separated the event handling from the management, which is in contrast to the approach taken by the triggered databases that combine event handling and database activities into a single component.

Acknowledgments

We gratefully acknowledge EPSRC and HP Laboratories Bristol for a CASE studentship as well as financial support from the Esprit SysMan (7026) Project. We also acknowledge the contribution of our colleagues at Imperial College, to the concepts discussed in this paper.

References

[1] Bates P 1995 Debugging heterogeneous distributed systems using event-based models of behaviour *ACM Trans. Comput. Syst.* **13** 1–31
[2] Haban D and Wybranietz D 1990 A hybrid monitor for behaviour and performance analysis of distributed systems *IEE Trans. Software Engng* **16** 197–211
[3] Jakobson G and Weissman M 1995 Real-time telecommunication network management: extending event correlation with temporal constraints *Proc. 4th Symp. Integrated Network Management (Santa Barbara, CA)* (London: Chapman and Hall) pp 290–301
[4] LeBlanc R J and Robbins A D 1985 Event-driven monitoring of distributed programs *Proc. 5th Int. Conf. on Distributed Computing Systems* pp 515–22
[5] Lumpp J E Jr, Casavant T L, Seigle H J and Marinescu D C 1990 Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems *Proc. 10th Int. Conf. on Distributed Systems* pp 476–83
[6] Spezialetti M and Bernberg S 1995 EVEREST: an event recognition testbed *Proc. 15th Int. Conf. on Distributed Computing Systems (Vancouver)* pp 377–85
[7] Spezialetti M and Kearns J P 1988 A General approach to recognising event occurrences in distributed computations *Proc. 8th Int. Conf. on Distributed Computing Systems* pp 300–7
[8] Wolfson O, Sengupta S and Yemini Y 1991 Managing communication networks by monitoring databases *IEEE Trans. Software Engng* **17** 944–53
[9] Mansouri-Samani M and Sloman M 1994 Monitoring distributed systems *Network and Distributed Systems Management* ed M Sloman (Reading, MA: Addison-Wesley) pp 303–47
[10] Mansouri-Samani M and Sloman M 1996 A configurable event service for distributed systems *Proc. IEEE 3rd Int. Conf. on Configurable Distributed Systems (Annapolis, MD)* pp 210–17
[11] Mansouri-Samani M 1995 Monitoring of distributed systems *PhD Thesis* Imperial College (see [24])
[12] Shim Y C and Ramamoorthy C V 1990 Monitoring and control of distributed systems *Proc. 1st Int. Conf. on Systems Integration* (Morristown, NJ: IEEE Computing Press) pp 672–81
[13] Chakravarthy S, Krishnaprasad V, Anwar E and Kim S-K 1993 Anatomy of a composite event detector *Technical Report* UF-CIS-TR-93-039, University of Florida, Department of Computer and Information Sciences
[14] Chakravarthy S and Mishra D 1993 Snoop: an expressive event specification language for active databases *Technical Report* UF-CIS-TR-93-007, University of Florida, Department of Computer and Information Sciences
[15] Dayal U 1988 Active database management systems *Proc. 3rd Int. Conf. on Data and Knowledge Bases (Jerusalem)* pp 150–69
[16] Gehani N, Jagadish H V and Shmueli O 1992 Composite event specification in an active databases: model and implementation *Proc. 18th VLDB Conf. (Vancouver)*
[17] Gatziu S and Dittrich K R 1994 Detecting composite events in active databases using Petri nets *Proc. 14th Int. Workshop on Research Issues in Data Engineering* Active Database Systems, Houston, TX
[18] Marzullo K, Cooper R, Wood M D and Birman K P 1991 Tools for distributed application management *IEEE Comput.* **24** 42–51
[19] Bacon J, Bates J, Richard H and Moody K 1995 Using events to build distributed applications *Proc. 2nd Int. Workshop on Services in Distributed and Network Environments (Whistler)* pp 148–55
[20] Klinger S, Yemini S, Yemini Y, Ohsie D and Stolfo S 1995 A coding approach to event correlation *Proc. 4th Symp. on Integrated Network Management (Santa Barbara, CA)* (London: Chapman and Hall) pp 290–301
[21] Nygate Y A 1995 Event correlation using rule and object based techniques *Proc. 4th Symp. on Integrated Network Management (Santa Barbara, CA)* (London: Chapman and Hall) pp 290–301
[22] OMG 1995 *CORBA Services: Common Object*

Specifications, Event Service Specification (March) available from http://www.omg.org

[23] Magee J, Dulay N and Kramer J 1994 REGIS: a constructive development environment for distributed programs *Distrib. Syst. Engng* **1** 304–12

[24] Karunadasa D 1996 Event based monitoring service *MEng Thesis* Department of Computing Imperial College. This thesis, Mansouri-Samani's PhD thesis and the Corba GEM monitoring service are available from ftp://dse.doc.ic.ac.uk/gem/

[25] Marriott D and Sloman M 1996 Implementation of a management agent for interpreting obligation policy *IEEE/IFIP Distributed Systems Operations and Management Workshop (DSOM 96) (L'Aquila, Italy)* available from http://www-dse.doc.ic.ac.uk/research/papers.html

[26] Sloman M 1994 Policy driven management for distributed systems *J. Network Syst. Management* **2** 333–60