Technical Report

UCAM-CL-TR-387
ISSN 1476-2986

Number 387

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Monitoring composite events in distributed systems

Scarlet Schwiderski, Andrew Herbert,
Ken Moody

February 1996

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

ISSN 1476-2986

Monitoring Composite Events in Distributed Systems

Scarlet Schwiderski* Andrew Herbert‡ Ken Moody*

* Computer Laboratory
Cambridge University
Pembroke Street
Cambridge CB2 3QG, UK

‡ Architecture Projects Management Ltd
Poseidon House
Castle Park
Cambridge CB3 0RD, UK

Abstract

One way of integrating heterogeneous, autonomous and distributed systems is to monitor their behaviour in terms of global composite events. In specific applications, for example database, it is essential that global composite events can take account of general conditions such as the timing constraints on distributed system behaviour. In this paper, the use of global composite events incorporating time events for expressing physical time is investigated. The detection of global composite events is complicated by the inherent features of distributed systems: lack of global time, message delays between sites and independent failures. Global event detectors are distributed to arbitrary sites. Relevant constituent events occur on remote sites and are signalled to corresponding global event detectors, where they are evaluated. Two different algorithms for the detection of global composite events are introduced, which are based on the evaluation of trees: asynchronous and synchronous evaluation. Asynchronous evaluation provides fast but unreliable detection of global composite events, whereas synchronous evaluation is characterized by reliability and unpredictable delays.

1 Introduction

One way of integrating heterogeneous, autonomous and distributed systems is to provide a monitoring service, which aims to detect behaviour patterns of interest in distributed computations. A detected behaviour pattern is associated with information relevant to its cause. A monitoring service collects and evaluates this information and, depending on the outcome, triggers certain actions. The monitoring service can, for example, notify interested users or start a new application program.

Composite events have been used successfully in numerous fields of research for specifying and detecting behaviour patterns of interest. In *distributed debugging*, composite events are employed to compare the expected system behaviour with the actual system behaviour [Bat88b]. In *active databases*, composite events depict the event-part of *Event-Condition-Action (ECA) rules* [DBM88]. The detection of an event causes the triggering of an action, if the condition holds. In databases, ECA rules are applied for integrity and security enforcement, constraint management and rule-based inference.

The basic building block of composite events are *primitive events*, instantaneous, observable "occurrences of interest". Examples of primitive events are the start or the completion of a method call, reaching a certain date and time, or a signal from a sensor. In general, primitive event expressions refer to classes of events rather than to single event occurrences. For example, the expression "customer X uses the ATM in Oxford Street" can occur numerous times at different points in time. Therefore, an expression specifies an *event class.* An instantaneous occurrence, which is determined by a time and location of occurrence and other event data, is called an *event instance.* For example, "customer Robert Smith uses the ATM in Oxford Street on 1/1/95:10:57, debiting £100" is an instance of the above event class. The name, time, location and amount, the so-called *event parameters*, express the circumstances under which the event occurred. Primitive event expressions can be composed with *event operators* like conjunction and sequence to form composite event expressions. Composite event expressions can define arbitrarily complex event scenarios. In a distributed system, primitive event expressions are site-relative, that means, event instances of any particular event class originate at the same site. The components of a composite event expression can involve events at many sites. Hereafter, we call these system-wide composite event expressions *global (composite) event expressions*, as opposed to *local (composite) event expressions*, which relate to event occurrences on a single site.

The detection of composite events differs considerably, depending on whether they are local or global. This is due to the inherent features of distributed systems [Bac92]: lack of global time, autonomy of sites, message delays between sites, and independent failure modes. Centralized systems are not concerned with the problems implied by these features and event detection is therefore comparatively easy.

In a distributed system, we distinguish between *local event detectors* and *global event detectors.* Local event detectors detect local event expressions and consequently employ "simple" centralized detection mechanisms (see e.g. [Cha89, Gat94, GJS92]). Detected local events are then forwarded to relevant global event detectors, which reside at the local and/or remote sites. In this way, the distinctive features of centralized and distributed systems can be isolated from each other. One of the major differences between local and global event detection is the role of time. Generally, the detection of composite events is driven by the *timestamps* of constituent events. Each primitive or composite event instance is assigned a specific timestamp, which identifies when the event happened. The timestamp of a composite event is determined by the timestamp of the last primitive event participating in its occurrence. A centralized system employs a single local clock. Hence, because all timestamps of events are read from a single clock, it is straightforward to determine the ordering of events. On the other hand, a distributed system employs numerous local clocks, one for each of its sites. These local clocks cannot be perfectly synchronized and record slightly different times. Therefore, it is difficult to determine the last event or even the meaning of "last". Before looking at event handling in distributed systems, we review the related fields of distributed debugging and active database systems.

1.1 Characteristics of Distributed Debugging Systems

One way of debugging distributed programs is the monitoring of global event expressions describing expected program behaviour [HZMW91, HW88, Bat88a, Bat88b, Spe91]. The primitive event expressions refer to the statements of a programming language.

One major characteristic of distributed debugging systems is that they do not utilize physical time. First, there is no notion of time events, that is, events relating to absolute or relative time cannot be used in a suitable way. Second, the temporal order between events is determined with respect to *causal order*, which is identical to the "happened before" relation defined by Lamport [Lam78]. In distributed debugging systems, the causal relationship between events is essential in order to locate the cause of errors. Consequently, timestamps relate to *logical time* rather than to physical time. Another characteristic of distributed debugging systems is that detected global events are displayed at a *single debugging station.* The reason for this is that the debugging process is monitored and analyzed by some user at a terminal.

1.2 Characteristics of Active Database Systems

In database systems, ECA rules present a general mechanism for supporting applications that require timely response to critical situations [DBM88].

As opposed to distributed debugging systems, active database systems utilize *physical time.* Time events express absolute, relative and periodic relationships and are employed to incorporate time constraints into composite event expressions [DBB88, GJS92, Cha89, WF90, Gat94]. Examples like "a customer debits money more than three times a day" or "the stock drops below the limit before 8pm" highlight the necessity of using time. One important implication of the use of physical time is that the temporal relationships between events must respect *physical order* rather than causal order. The characteristic of active database systems is that by using physical time, the external behaviour of the system is monitored, not its algorithm (c.f. debugging). Thus, ECA rules can be understood as "real world" rules, i.e. expressing user application concepts, not system concepts. To date, active database systems have mainly been explored in centralized environments. [JS92] and [CW92] consider ECA rules in distributed database environments. They do, however, not address general global composite events.

1.3 Objectives

In this paper, we explore the use of global composite events in distributed database environments. We present and discuss algorithms for the fully distributed detection of global composite events. The conceptual novelty in dealing with such events arises from the distributed nature of the constituent primitive events; global composite event expressions include time event expressions and are therefore constrained by physical time. The inevitable inaccuracies of local clock synchronisation, as well as the need to be able to deal with independent failures and delayed network links, are characteristic of distributed environments.

The proposed algorithms are designed to deal with these problems. Furthermore, the development of an object-oriented prototype implementation incorporates the following features:

- Two different detection algorithms are implemented, *asynchronous* and *synchronous detection*, providing either a fast and unreliable or a slow and reliable service, depending on application concerns.

- Detection algorithms are evaluated concurrently in order to avoid blocking and to improve response times.

- Network object pointers are used to make the handling of parameters and detection of global composite events transparent.

1.4 Outline of Paper

In section 2, we describe the features of distributed systems and the special assumptions underlying our investigations. The specification of global composite events is introduced in section 3. Section 4 is concerned with the detection of global composite events, namely, what timestamps look like, what data structures underlie our detection algorithms and how they are evaluated and what are the differences between asynchronous and synchronous detection. Implementation details are given in section 5. Section 6 summarizes the paper.

2 Assumptions for Distributed Systems

1. *No global time*
   Each site in a distributed system has its own *local clock*. These clocks can drift and therefore record slightly different times.

2. *Message delays between sites*
   Messages sent over a computer network can be delayed depending on the load at the sender and receiver sites and the network load.

3. *Independent failure modes*
   The sites and transmission channels of a distributed system may fail independently of each other.

In our approach, we assume that clocks can be *synchronized*, that is, the maximum time difference between any two clocks at any one physical time is bounded. Hereafter, we call this maximum time difference the *precision* Π[1]. A global time can then be approximated [Kop92, Ver93]. The "granularity condition" states that the *granularity of the global timebase* g should not be smaller than Π, $g > \Pi$. Fulfilment of this condition ensures that global ticks do not overlap. Moreover, the temporal order of two events in different sites can be determined, if their timestamps (clock ticks measured with respect to the global timebase) are two or more clock ticks apart, a fact known as $2g$*-precedence.*

Using this time model is convenient, because a temporal order between any two events can easily be decided. If two events occur at different sites, one event occurs before the other, if they are two or more clock ticks apart. Otherwise, a temporal order cannot be established and the events are said to happen in *parallel* or *concurrently* [2]. If two events occur at the same site, one event occurs before the other, if they are one or more clock ticks apart.

We also assume *FIFO network delivery*, that is, messages originating at any one site are delivered at any other site in the order they were generated. FIFO network delivery can be achieved easily using remote procedure call.

Under the assumption of FIFO network delivery events occurring at one site are signalled at corresponding global event detectors in the order of their occurrence. This is convenient, because event detectors combine event instances following certain policies. Note that there may be multiple event instances of each event class, when evaluating a composite event. The different instances are related with different sets of event parameters,

[1]The assumption on synchronized clocks is reasonable, since a synchronisation of 100 msec can be achieved even in WANs (using e.g. NTP, the Network Time Protocol [Mil91]). Telecommunication networks can deliver much greater precision.

[2]Note that the two events may have the same timestamp.

among other things their timestamp. Which of the available event instances are combined for the detection of a composite event depends on application demands. [CKAK94] introduces the *parameter contexts* chronicle, recent, continuous and cumulative. For example, the chronicle context evaluation combines the oldest event instances available and the recent context evaluation combines the newest event instances available. In this paper, we only consider global event detection in the chronicle context.

Handling independent failures is not an easy task. One major problem is to detect a failure and in particular to locate the site or the transmission channel which causes it.

The problem which arises for global event detection is, whether events are not signalled from a particular site because *there are none*, or because *the site and/or connecting channels respectively have failed.* There are two main solutions to this. First, composite events fire whenever suitable events are received, without regard to delayed or lost events. In this case, late coming events can either be ignored or consumed by other composite events. Alternatively, event detection only proceeds if all sites have signalled their current status. In this case, event detectors evaluate incoming events in temporal order or wait until a message "there are no relevant events" has been received. The first policy realizes an *asynchronous evaluation* and the second policy a *synchronous evaluation.*

3 Specification of Global Composite Events

In centralized active database systems, *primitive events* are divided into the following categories: time events, method events, value events, transaction events and abstract events [Gat94]. *Time events* occur when certain points in time are reached. Absolute, relative and periodic time specifications are allowed. *Method events* arise at the start or the completion of a method call. Events referring to the start or the completion of an update of a data value are called *value events. Transaction events* occur at the beginning, the commit or the abort of a transaction and *abstract events* appear, when external users or applications notify event occurrences to the system. For a more detailed discussion on primitive events and their specification see [Gat94].

Definition 3.1 (Global event expressions) A *global event expression* is built as follows:

- A primitive event expression is a global event expression.
- If E_1, E_2 and E_3 are global event expressions, then (E_1, E_2), $(E_1 \mid E_2)$, $(E_1; E_2)$, $(E_1 \| E_2)$, $(E_1; \text{NOT } E_2; E_3)$ and $(E_1^* E_2)$ are global event expressions.
- Nothing else is a global event expression.

(E_1, E_2) denotes a *conjunction* between two events and is detected when both events occur. The *disjunction* $(E_1 \mid E_2)$ is detected when either of the events occur. $(E_1; E_2)$ represents a *sequence* and is detected when E_1 occurs before E_2. A *concurrency* is depicted as $(E_1 \| E_2)$ and is detected when E_1 and E_2 occur concurrently. The *negation* $(E_1; \text{NOT } E_2; E_3)$ is detected when an event E_2 does not occur after an event E_1 and before an event E_3 (this implies that E_3 occurs after E_1) and finally, $(E_1^* E_2)$ denotes an *iteration* and is detected when E_2 occurs after zero or more occurrences of E_1.

Definition 3.1 gives a recursive definition of global event expressions. Primitive event expressions as well as global event expressions can serve as the basic building block in the construction. Therefore, the construction is nested. This enables the user to split up complex event scenarios into smaller units. Another advantage, besides increasing clarity, is that different units can be evaluated independently of each other at remote sites.

In section 2 we addressed a time model for distributed systems. This time model is general in that it applies to all distributed systems with synchronized clocks. The notion of $2g$-precedence defines our understanding of *temporal order* in distributed systems, namely the meaning of before, after and concurrently. Two non-local events E_1 and E_2 occur one before (after) the other, if the timestamp of E_1 is at least two clock ticks below (above) that of E_2, and concurrently, if the timestamps are less than two clock ticks apart. Therefore, non-local events are partially ordered. As opposed to non-local events, local events always occur one after the other. Because timestamps relate to the same local clock, the temporal order can be established, if they are one or more clock ticks apart. In this paper, we assume that no two primitive events originating at the same site will carry the same timestamp[3]. Therefore, local events are totally ordered.

The semantics of the event operators has to be investigated carefully in the light of this time model.

4 Detection of Global Composite Events

The detection of global composite events is driven by the timestamps of constituent events. Each primitive or global event is associated with a single timestamp. The timestamp of a global event is determined by the latest timestamp of events participating in its occurrence. An event bearing this timestamp is called a *terminator event.* Timestamps are used for:

- detecting a sequence, iteration, negation or concurrency between two constituent events.
- deriving the timestamp of a global event occurrence.
- determining which instances of constituent events are evaluated next (depending on the parameter context).

In the following section, we discuss the implications of the above requirements on the specification and evaluation of timestamps. The detection procedure for global events is illustrated afterwards, depending on one of two evaluation semantics, asynchronous and synchronous evaluation.

4.1 Specification and Evaluation of Timestamps

Global event expressions are built recursively from primitive and global event expressions and event operators. The following list gives an informal overview of the timestamps derived at each step, depending on the specific event operator.

Disjunction $\mathbf{E_1|E_2}$: A disjunction is detected when either E_1 or E_2 occurs. The timestamp of that occurrence becomes the timestamp of the disjunction.

[3]Note that the granularity of the global timebase is coarse in comparison with the granularity of the local clocks. Distinct events at the same local site may therefore carry identical global timestamps. The events can be distinguished by tagging them with local clock values to capture the finer granularity.

Sequence $\mathbf{E_1; E_2}$: Inherently, a sequence expresses the happened-after relation between events. Event E_2 occurs after event E_1 and hence, its timestamp is the timestamp of $E_1; E_2$.

Iteration $\mathbf{E_1^{*}E_2}$: An iteration is a special case of a sequence, in which E_2 serves as a terminator event following a number of occurrences of E_1. Consequently, the timestamp is that of E_2.

Negation $\mathbf{E_1; NOT\ E_2; E_3}$: As above, a negation is a special case of a sequence. Therefore, E_3 defines the timestamp of the negation.

Concurrency $\mathbf{E_1 \| E_2}$: It is the nature of concurrency that no temporal order between E_1 and E_2 can be inferred from their timestamps. Neither of the events can be regarded as occurring after the other, therefore both E_1 and E_2 must be considered, and their timestamps are joined.

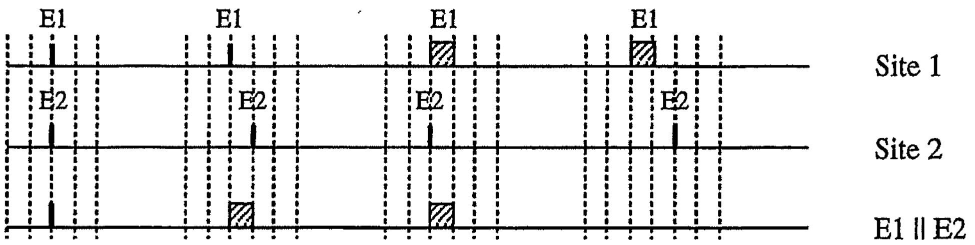

Figure 1: Examples for concurrency

Figure 1 shows four examples of occurrences of events E_1 and E_2. Each vertical line relates to a specific global clock tick. The first example shows E_1 and E_2 both having the same global clock value. This timestamp is inherited by $E_1 \| E_2$. In the second example the timestamps are one clock tick apart. Because of $2g$-precedence, it cannot be determined which event occurred last, although E_2 has a larger clock value than E_1. As a result the timestamps are joined and cover an interval of one clock tick. The third and fourth examples show how such joined timestamps are related to other events E_2. In the fourth example a concurrency cannot be detected, because the lower bounds of E_1 and E_2 are two clock ticks apart and the events are therefore sequential. The result of comparing two local clock values depends on whether they were generated at the same site, and timestamps must therefore record the originating sites as well as the associated local clock values.

Conjunction $\mathbf{E_1, E_2}$: The timestamp of a conjunction is the later timestamp of E_1 and E_2, if the temporal order between E_1 and E_2 can be decided. Otherwise the two timestamps are joined.

After discussing issues on deriving timestamps for composite events, the structure of timestamps and their handling is illustrated. Before defining timestamps as such, the structure of clock values is addressed.

Definition 4.1 (Local clock value) The global clock granularity g is given. A *local clock value* t represents a moment in time, reckoned as a number of clock ticks of granularity g since some epoch or starting point.

A local clock value of granularity $g = 0.01sec$ could, for example, be represented as day/month/year:hours:minutes:seconds.hundredths.

Definition 4.2 (Timestamp) Given S, the total number of sites, a *timestamp* T consists of

- a local clock value, the so-called *base*
- a duration 0 or 1, the so-called *interval*
- a partial function *offset* : $S \dashrightarrow \{0, 1\}$ defining a local clock value for each site participating in the timestamp.

The *base* of a timestamp represents the minimum value of a local clock, read at some participating site. The *interval* is set to 1, if the timestamp covers an interval of one clock tick, that is, if the timestamp refers to concurrent events having distinct local clock values. Otherwise it is set to 0. The *(site) set* of T is the domain of the partial function *offset.* For each s in the site set, *offset*(s) determines the local clock value, either the *base* of the timestamp (value 0) or *base*+1g (value 1). Hereafter, we refer to *base*+*offset*(s) as *localclock*(s) for each participating site s, and to *base*+*interval* as the *limit.* We represent a timestamp T as a triple *(base, interval, offset).*

Definition 4.3 (Timestamp of a primitive event) Given a primitive event E occurring on site X at local time t, the timestamp $T(E)$ is determined as follows:

$$\begin{aligned} &T(E).base := t \\ &T(E).interval := 0 \\ &T(E).offset := \{(X, 0)\} \end{aligned}$$

Example 4.4 The timestamp of a primitive event originating at site A may look as follows:

$$T(E_1) = (27619951703, 0, \{(A, 0)\})$$

So far we have discussed informally, whether two events are sequential or concurrent in the $2g$-precedence time model. After introducing the syntax of timestamps, we can now give a formal definition of the notions of after $<$ and concurrent $\sim$.

Definition 4.5 (Temporal relation) On the basis of the $2g$-precedence time model, the *temporal relation* between two events E_1 and E_2 is defined as follows[4]:

$$\begin{array}{llrl} T(E_1) \sim T(E_2) & \text{iff} & & limit_2 - base_1 < 2g \\ & & \textbf{and} & limit_1 - base_2 < 2g \\ & & \textbf{and} & \forall s \in set_1 \cap set_2,\ localclock_1(s) = localclock_2(s) \\ T(E_1) < T(E_2) & \text{iff} & & limit_2 - base_1 \geq 2g \\ & & \textbf{or} & base_2 = base_1 + 1g \\ & & & \textbf{and}\ \exists s \text{ s.t. } localclock_1(s) < localclock_2(s) \end{array}$$

[4]For abbreviation, the timestamp identification is written as subindex (e.g. $base_1$ instead of $T(E_1).base$).

Rationale:
Event E_1 occurs concurrently with event E_2, $T(E_1) \sim T(E_2)$, if no two component local clock values are two or more clock ticks apart, and if also the local clock values (and therefore the associated primitive events) are identical for each site participating in both E_1 and E_2. Note that $\sim$ is NOT transitive.
Event E_1 occurs before event E_2, $T(E_1) < T(E_2)$, if the limit of $T(E_2)$ is at least two clock ticks ahead of the base of $T(E_1)$ or, if there are two component local clock values read from the same site s and $T(E_2)$'s component is one clock tick ahead of $T(E_1)$'s. The first case corresponds to the basic definition of $2g$-precedence, whereas the second case expresses the fact that timestamps contain two components which are read from the same local clock and are therefore sequential. Note that $<$ IS transitive.

Proposition 4.6 If neither $T(E_1) < T(E_2)$ nor $T(E_2) < T(E_1)$, then at most two adjacent local clock values cover the timestamps $T(E_1)$ and $T(E_2)$.

Proof: Assume without loss of generality that $base_1 \leq base_2$.
Since $T(E_1) \not< T(E_2)$, $limit_2 \not\geq base_1 + 2g$. Therefore,

$$limit_2 < base_1 + 2g \Longrightarrow base_1 \leq base_2 \leq limit_2 \leq base_1 + 1g.$$

Hence, the only possible local clock values arising in $T(E_1)$ and $T(E_2)$ at any participating site s are $base_1$ and $base_1 + 1g$. □

There are pairs of events E_1 and E_2 for which none of the three cases $T(E_1) < T(E_2)$, $T(E_2) < T(E_1)$, $T(E_1) \sim T(E_2)$ applies. For example,

$$\begin{array}{ll} localclock_1(A) = 306 & localclock_2(A) = 307 \\ localclock_1(B) = 307 & localclock_2(B) = 306 \end{array}$$

Earlier in this section we argued that the timestamps of constituent events have to be joined when detecting a concurrency or a conjunction with constituent events that cannot be ordered. The following definition shows how timestamps are joined.

Definition 4.7 (Joined timestamps) Suppose given two timestamps such that neither $T(E_1) < T(E_2)$ nor $T(E_2) < T(E_1)$. The joined timestamp $\hat{T} = T(E_1) \cup T(E_2)$ is defined as follows:

$$joinset := set_1 \cup set_2$$

$$joinlocalclock(s) := \begin{cases} MAX\ \{localclock_1(s), localclock_2(s)\} & \forall s \in set_1 \cap set_2 \\ localclock_1(s) & \forall s \in set_1 \backslash set_2 \\ localclock_2(s) & \forall s \in set_2 \backslash set_1 \end{cases}$$

$$joinlimit := MAX\ \{joinlocalclock(s) \mid s \in joinset\}$$

$$\hat{T}.base := MIN\ \{joinlocalclock(s) \mid s \in joinset\}$$

$$\hat{T}.interval := \begin{cases} 0 & \text{if } joinlimit = \hat{T}.base \\ 1 & \text{otherwise} \end{cases}$$

And for all $s \in joinset$, $\hat{T}.offset(s) := \begin{cases} 0 & \text{if } joinlocalclock(s) = \hat{T}.base \\ 1 & \text{otherwise} \end{cases}$

This construction defines a valid timestamp whenever the events E_1 and E_2 cannot be placed in temporal sequence. Intuitively, the joined timestamp records the higher of the local clock values recorded for E_1 and E_2 at each site participating in either event. Note that this definition is in particular applicable whenever $T(E_1) \sim T(E_2)$.

Example 4.8 Given $T(E_2) = (27619951704, 0, \{(B, 0)\})$. Suppose $T(E_1) \sim T(E_2)$, where $T(E_1) = (27619951703, 0, \{(A, 0)\})$. Then the joined timestamp $\hat{T} = T(E_1) \cup T(E_2)$ is

$$(27619951703, 1, \{(A, 0)(B, 1)\})$$

Example 4.9 Given two timestamps $T(E_1) = (306, 1, \{(A, 0)(B, 1)\})$ and $T(E_2) = 306, 1, \{(A, 1)(B, 0)\})$. Because neither $T(E_1) < T(E_2)$ nor $T(E_2) < T(E_1)$ the timestamps can be joined and $\hat{T} = T(E_1) \cup T(E_2)$ is

$$(307, 0, \{(A, 0)(B, 0)\})$$

The semantics of the event operators is defined next. The definition corresponds to the explanations on event operators given in the beginning of this section.

Definition 4.10 (Semantics of event operators) On the basis of the $2g$-precedence time model, the event operators are defined as follows. The first part of the right-hand side identifies under which pre-conditions a composite event of that kind is detected and the second part, indicated by $\Rightarrow$, presents the resulting timestamp:

E_1, E_2	iff	E_1 and E_2
		$\Rightarrow T(E_2)$ iff $T(E_1) < T(E_2)$
		$\Rightarrow T(E_1)$ iff $T(E_2) < T(E_1)$
		$\Rightarrow T(E_1) \cup T(E_2)$ otherwise
$E_1 \mid E_2$	iff	E_1 or E_2
		$\Rightarrow T(E_1)$ or $T(E_2)$
$E_1; E_2$	iff	E_1 and E_2 and $T(E_1) < T(E_2)$
		$\Rightarrow T(E_2)$
$E_1 \parallel E_2$	iff	E_1 and E_2 and $T(E_1) \sim T(E_2)$
		$\Rightarrow T(E_1) \cup T(E_2)$
$E_1^* E_2$	iff	E_{1_i} $(i \geq 0)$ and E_2 and $T(E_{1_i}) < T(E_2)$
		$\Rightarrow T(E_2)$
$E_1;$ NOT $E_2; E_3$	iff	E_1 and E_3 and $T(E_1) < T(E_3)$ and
		no E_2 with $T(E_1) < T(E_2) < T(E_3)$
		$\Rightarrow T(E_3)$

4.2 Detection Procedure

The detection of global events is based on the evaluation of trees. Each global event expression E is transformed into a *global event tree* $GT(E)$, corresponding to its syntactic structure. Nodes are labelled with event operators and leaves are labelled with primitive event expressions or global event expressions. Each node contains a number of lists for detected sub-events, one for each of its children. Negation-nodes have three children, all other nodes have two children. In other words, a list contains the detected event occurrences of the corresponding child node.

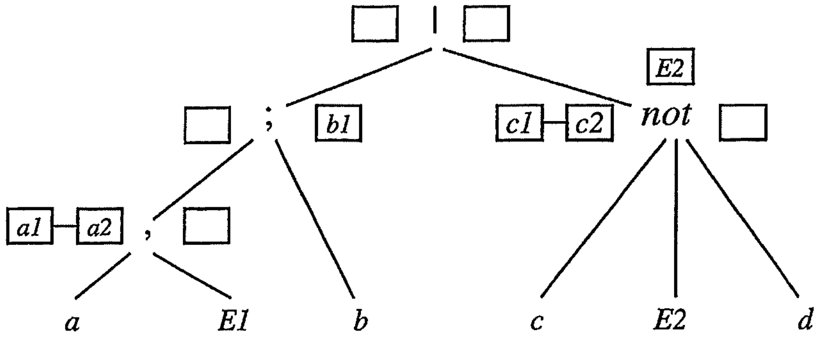

Figure 2: A snapshot of a global event tree

Example 4.11 Figure 2 shows a snapshot during evaluation of the global event tree of $((a, E_1); b) \mid (c; \text{NOT } E_2; d)$, indicating the lists of stored sub-events as linked boxes. Empty lists are represented as empty boxes. The leaves E_1 and E_2 refer to global event trees stored and evaluated elsewhere. The other leaves refer to local primitive events.

Algorithm 4.12 The evaluation of global event trees at a global event detector site proceeds as follows:

1. An event instance is signalled at the global event detector.
2. The event instance is inserted into all leaves corresponding to its event class.
3. Each inserted event instance is propagated to the parent node.
4. The parent node is evaluated, taking into account:
   (a) the event operator
   (b) the inserted event instance
   (c) the event instances stored in the child lists
   (d) the evaluation semantics, asynchronous or synchronous

   If no event is detected at the parent node, continue at 5).
   If an event is detected at the parent node, derive the new event instance, delete the consumed event instances and continue at 6).
5. The inserted event instance is stored in the corresponding child list.
   The algorithm stops.
6. If the current node has a parent node, continue at 3).
   If the current node has no parent node, a global composite event is detected and the algorithm stops.

Algorithm 4.12 presents the steps involved in the evaluation of the global event trees stored at a global event detector site. At runtime, local events occur and are detected at their *local event detectors.* The local event detectors initiate messages containing the event class, the timestamp and other parameters, and send these messages to those local or remote *global event detectors* which have registered an interest in the particular event class [BBHM95]. Received event instances are evaluated by global event detectors, that is, they (i.e. their timestamp and other parameters) are inserted into all leaves corresponding to the event class. Leaves serve as entry points and propagate the event instances upwards to their parent nodes, where they are evaluated. Evaluation at a node takes into account the stored sub-events as well as the newly arrived event instance. The evaluation procedure depends on the evaluation semantics, asynchronous or synchronous. If an event occurrence is detected, it is again propagated to the parent node and the procedure recurses or, if there is no parent node, it is signalled to the event manager. The latter case means that an instance of a specified global event has been detected. In both cases, consumed sub-events are deleted. If no occurrence is detected, the event instance is inserted into the corresponding list of sub-events.

There are different policies for evaluating nodes, depending on the handling of delayed events (which comes down to the handling of site failures, network congestion and network partitioning). There are two possibilities: ignoring delayed events and waiting for delayed events. In the following two sections we discuss both policies and call them *asynchronous* and *synchronous evaluation*[5].

4.2.1 Asynchronous Evaluation

Definition 4.13 (Asynchronous evaluation) A global event tree is evaluated *asynchronously*, if each node is evaluated instantly on the arrival of an event instance from a child node.

Events affected by a failure (site failure, network partitioning or network congestion) are delayed until the failure is repaired. Asynchronous evaluation means that nodes are evaluated irrespective of failures. When events arrive at a node, there may be other events with smaller timestamps which have not yet arrived. The node is, however, evaluated instantly. This implies that events from specific child nodes are not necessarily evaluated in the order of their occurrence, that is, in the chronicle parameter context. More recent events with larger timestamps from other child nodes will be handled as soon as they become available. What is done with the delayed events is another matter for decision. They may either be accepted for event detection as soon as they arrive or they may be ignored and perhaps sent back to their site of origin.

The main advantage of asynchronous evaluation is that global event trees are evaluated and composite events are detected regardless of remote failures. Delayed events do not cause temporary blocking of the detection procedure. The simplest example is that of a disjunction $E_1 \mid E_2$. If E_2's site has failed, the disjunction can still be detected whenever E_1 is signalled. Therefore, asynchronous evaluation is characterized by immediate consumption, non-blocking detection and good response times. The main disadvantage of asynchronous evaluation is that it does not guarantee event detection in the chronicle or any other parameter context. Whether this can be accepted or not depends on the specific global composite event and the application domain.

4.2.2 Synchronous Evaluation

Definition 4.14 (Synchronous evaluation) A global event tree is evaluated *synchronously*, if each node is evaluated on the arrival of an event instance from a child node

[5]The terminology asynchronous and synchronous evaluation relates to the notion of asynchronous and synchronous communication in distributed systems [Mul93].

provided that all event instances from other child nodes which have smaller timestamps have arrived.

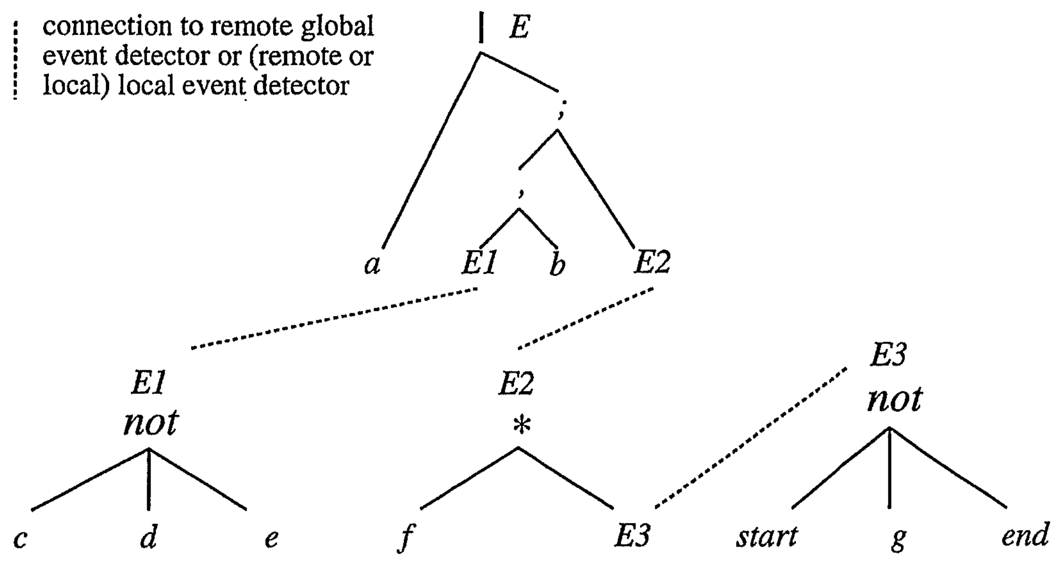
connection to remote global event detector or (remote or local) local event detector
E
;
a
,
E1
b
E2
E1
not
c
d
e
E2
*
f
E3
E3
not
start
g
end

Figure 3: A hierarchical structure of global event trees

Synchronous evaluation means that the evaluation of a node is delayed until all relevant events with smaller timestamps have arrived at the global event detector. Relevant events arrive from the sites relating to the siblings of the child node. There are no relevant events from the child node itself, because nodes are evaluated synchronously and because network delivery is FIFO (⇒ all events from one node arrive in the order of their occurrence). The siblings may have other child nodes, which again have children, and so forth. Therefore, the existence of a relevant event depends recursively on numerous event reporting sites. Figure 3 illustrates this. The right child of E depends on all nodes in its subtree. In evaluating a composite event synchronously every relevant primitive event that might participate in the evaluation of its parent node and other ancestors must be considered. Therefore, all event reporting sites in the subtree have to be checked for relevant events before an event occurrence can be detected at the parent node. In most cases, checking reveals that there are no relevant events. Occasionally there are relevant events which have been delayed because of failure.

The goal is to develop a checking procedure which is efficient and does not cause unnecessary overhead. The following methods are possible:

Dummy events For each event with a global scope, a procedure is implemented which raises a dummy event periodically if there is no "real" event, and sends it to all registered global event detectors.

Token passing Tokens are passed in a virtual ring of sites. The evaluation of a node can proceed when a token is returned to its originator.

Requests Requests are sent recursively to all sites which report event occurrences and from them to their children. This procedure continues until it is confirmed that all relevant event occurrences have already been signalled.

Our approach investigates the last possibility, namely, sending requests to corresponding sites. This decision is justified, because used for this purpose dummy events cause a big overhead on network traffic and event processing, and token passing often delays event detection unnecessarily. For a detailed discussion refer to [Sch95].

Algorithm 4.15 The following algorithm checks a subtree starting with its root node *child* for event occurrences until time *checktime*. Each node has a variable *logicaltime* which represents the minimum timestamp of the next event occurrence[6]:

```
Check_Subtree(child, checktime)
IF child is leaf THEN
   CASE leaf's event class originates at
      local site: logicaltime := Time.Now+1
                  RETURN
      remote site: logicaltime := Check_Remote_Tree(eventclass, checktime)
                   RETURN
   END
ELSE /* child is node */
   IF logicaltime ≥ checktime THEN
      RETURN
   ELSE
      CASE event_operator is
         Conjunction,
         Sequence,
         Concurrency =>
            Fork a thread(Check_Subtree(leftchild, checktime))
            Fork a thread(Check_Subtree(rightchild, checktime))
            IF one thread returns THEN
               IF there are no event instances in the corresponding child list
                  with timestamps smaller than checktime THEN
                  logicaltime := MAX{logicaltime, checktime}
                  RETURN
               ELSE
                  Wait for other thread
                  logicaltime := MAX{logicaltime, checktime}
                  RETURN
               END
            END
         Disjunction =>
            Fork a thread(Check_Subtree(leftchild, checktime))
            Fork a thread(Check_Subtree(rightchild, checktime))
            Wait for both threads
            logicaltime := MAX{logicaltime, checktime}
            RETURN
      END
   END
END
```

[6]The event operators iteration and negation have not been included: they are similar.

Algorithm 4.15 checks the nodes of a subtree recursively. The recursion stops, if the child is a leaf corresponding to the local site (no network delays and therefore no unknown primitive event occurrences up to the current point in time), or if the child is a node which is already checked up to *checktime*. If the child is a leaf corresponding to a global event tree at a remote site, the request is forwarded (*Check_Remote_Tree*[7]). If the child is a node, the checking procedure depends on its event operator. Conjunction, sequence and concurrency operators have two children and need exactly one event instance from each child in order to detect an occurrence. Therefore the checking procedure can return as soon as it is certain that there is no event occurrence from *one* of the children. In the case of a disjunction, both subtrees have to be examined.

For synchronous evaluation, step (4d) in algorithm 4.12 involves applying algorithm 4.15 before evaluating a disjunction, iteration, or negation node. The algorithm is applied to all siblings of the child node propagating the new event occurrence. The evaluation of the parent node blocks until the algorithm returns or until relevant event instances arrive. In the case of a conjunction, sequence, or concurrency node, the application of algorithm 4.15 is redundant, because an event occurrence cannot be detected before a suitable event instance has arrived from the sibling node. Blocking the evaluation until all relevant event instances have arrived from the sibling node is therefore inherent.

In contrast to asynchronous evaluation, synchronous evaluation of a node blocks until all relevant sites have signalled relevant event occurrences to that node. That can cause considerable delays and therefore lead to bad response times. On the other hand, events are always evaluated in the order of their occurrence. This guarantees chronicle parameter context.

5 System Architecture

The mechanisms for specifying and detecting global composite events in distributed systems are currently being implemented using *Modula-3 for Network Objects* [BNOW94]. Modula-3 for network objects is a distributed programming system, where communication over a network is done using network objects. A network object is an object whose methods can be invoked over a network. The program containing a network object is called the *owner* of the network object and the program using it is called the *client*. An important feature of Modula-3 for Network Objects is that a *network object pointer* can be passed as an argument or result between network nodes. This provides a more powerful mechanism than ordinary RPC.

The implementation includes two processes running independently of each other on each site in the distributed test environment: an *event simulator process* and an *event detector process*. In a test environment of N sites, there are $2 \times N$ processes, N event simulators and N event detectors.

An event simulator simulates primitive event occurrences. Different simulator setups can be installed at different sites, defining which primitive events occur, and when. The occurrence can be random and/or predetermined. Simulated events of different event classes are tagged with the timestamp containing the current local clock value and the originator site. Information on which detector sites have registered an interest in occurrences of specific event classes is kept in a class/site table. Event occurrences are signalled to

[7] *Check_Remote_Tree(eventclass, checktime)* locates the detector site of *eventclass* and applies the *Check_Subtree* algorithm to the root node of the corresponding global event tree.

specific detector sites by calling the Signal_Primitive_Event-methods of particular Port network objects at these sites, containing the event class and the timestamp as arguments.

A global event detector receives event instances from multiple sites via Port network objects. There is one port for each event reporting site. Since network delivery is FIFO point-to-point, the event instances arrive at a single port in the order in which they were sent. Also, network objects can be invoked concurrently. We exploit the fact that events are signalled concurrently at ports and allow the concurrent evaluation of global event trees. The threads corresponding to different ports synchronize at single nodes. This means, if two events arrive at a node simultaneously, one thread is evaluated while the other one is waiting. Different nodes can, however, be evaluated concurrently. This is especially useful in synchronous evaluation, because the evaluation of a node can block for a long time if an event reporting site has failed. Other nodes, which are not influenced by the failed site, can be evaluated in the meanwhile.

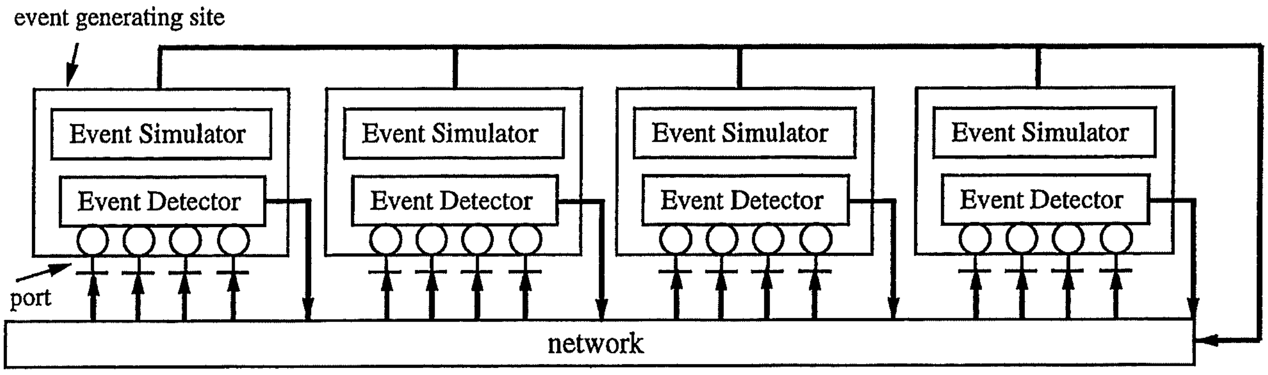

Figure 4: The underlying system architecture (4 sites)

Figure 4 illustrates the system architecture with four sites. Event simulators signal primitive events to sites via the corresponding ports. Each Port network object receives the primitive and detected composite events from a single generating site. The event detectors evaluate incoming events concurrently. Detected global composite events can be resignalled to any event detector site.

The global event trees in an event detector site are implemented as dynamic data structures. Figure 5 shows an instance of a global event tree. The dynamic data structure corresponds to an inverse tree, where the leaves serve as entry points pointing to their parent node and so forth. The root node represents a corresponding global composite event expression. An incoming event instance is inserted into the leaves corresponding to its event class and directly propagated to the parent node. The parent node is evaluated, depending on the specified evaluation semantics, synchronous or asynchronous. If no event is detected, the event instance is stored in the corresponding child list (e.g. $E1_1$, $E1_2$, b_1, c_1, c_2 and $E2_1$). If an event is detected, a new event instance is derived which contains a timestamp and parameters, namely, pointers to the constituent event instances (e.g. the event relating to the left operand of the root node).

The implementation of asynchronous evaluation is straightforward. An event arrives at a node and is evaluated with respect to the events already available. If an event is detected, a new event instance is derived, the consumed events are deleted and the new instance is propagated. An important point to consider is the *garbage collection* of obsolete events. During evaluation of sequence, concurrency and negation nodes, events accumulate at the node which cannot be used for event detection. For example, if there

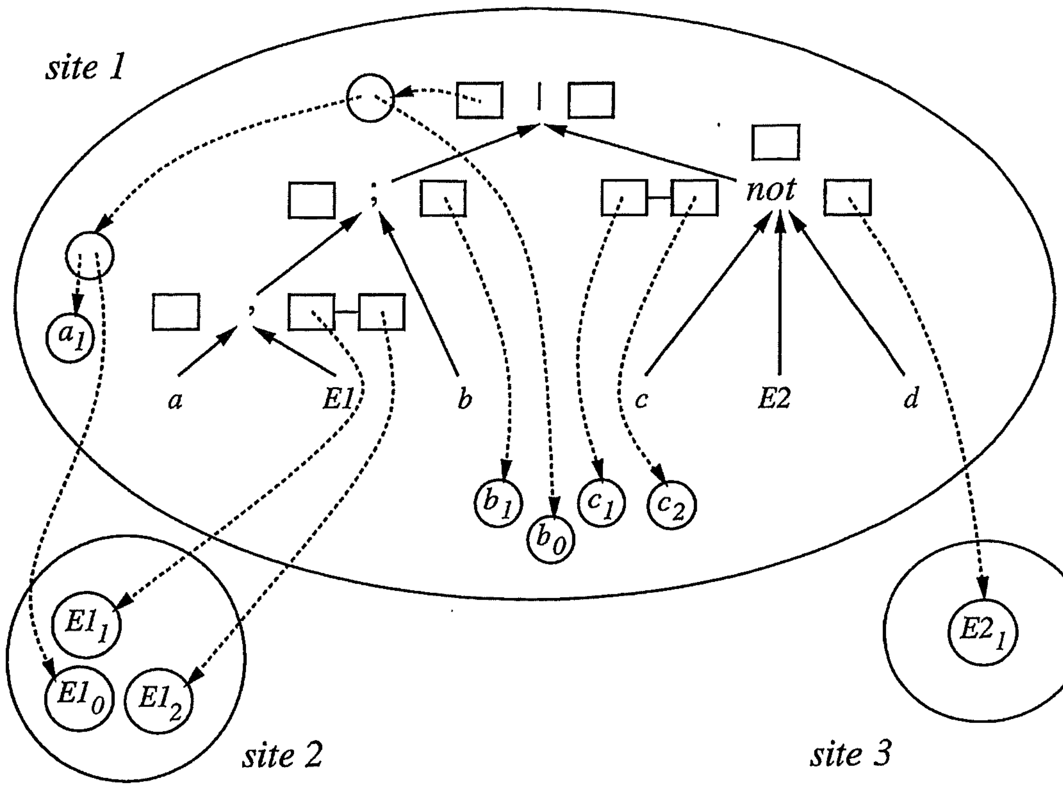

Figure 5: An instance of a global event tree

are multiple E_2 events for $E_1; E_2$, but no E_1 events with smaller timestamps, the E_2's can be deleted. It seems reasonable to apply garbage collection independently at each node, when the information regarding sibling events (e.g. what is the minimum timestamp of E_1) is available.

The implementation of synchronous evaluation involves algorithm 4.15. Regarding the data structures, the variable *logicaltime* has to be defined at each node in addition to the data structures used in asynchronous evaluation. Also, the algorithm works in the inverse direction to the detection algorithm. Therefore, each node contains pointers to its children. When an event arrives at a disjunction, iteration or negation node, the node can only be evaluated if all events up to the timestamp are available. If the sibling(s) contain no events or only events with smaller timestamps, the *Check_Subtree* procedure is initiated and evaluation blocks until it returns, or until relevant events arrive. In comparison with asynchronous evaluation, the garbage collection of obsolete events is simplified, because events always arrive in the order of their occurrence at each node.

Figure 5 shows that some pointers refer to objects in remote sites. These pointers are called *network object pointers*. Network object pointers are employed for the communication of objects, rather than copying. This has numerous advantages:

- It minimizes the copying of data. Event instances often have extensive parameters. Also, event instances of a certain class are often used in more than one event detector. Instead of copying the event instance with all its parameters each time and sending it to remote sites, we simply send its reference.
- Abstraction: it allows different local representations of objects and, in particular, of event instances at different sites, and can therefore be applied in heterogeneous

systems.

- Audit logging: each site can keep an audit trail of events structured for local purposes. The global event trail will be a structured thread through these local logs.
- Traceability: the full structure can be traversed by any observer. This gives the possibility of global logging and of alternative semantics.

6 Conclusions

In this paper, we have shown how to apply ideas originating in centralized active database systems to distributed systems, i.e. how to realize global composite events in a general way. This means that the event detectors of global composite events are distributed to a number of sites and that constituent events occur at arbitrary sites. A number of the problems involved have already been studied in the field of distributed debugging. However, active database systems monitor the external behaviour of the system and not its algorithm. This means that physical time is relevant. Hence, one major problem was to investigate the meaning of physical time in distributed systems and to fix a semantics for the notions of *after* and *concurrent*. It seemed convenient to apply the $2g$-precedence time model [Kop92], which is applicable in all distributed systems with synchronized clocks.

We have developed two detection algorithms for global composite events realizing two distinct evaluation semantics: an asynchronous and a synchronous semantics. Asynchronous evaluation offers fast detection. The detected events do, however, not necessarily reflect the temporal order of event occurrences. Consequently, parameters can be combined with respect to parameter contexts in an unpredictable way. Unlike asynchronous evaluation, synchronous evaluation reflects the temporal order of event occurrences. Composite events are evaluated under full knowledge of system behaviour. Detection is, however, possibly blocked for long periods in the case of site and network failures. Both detection algorithms are evaluated concurrently, synchronizing at single nodes of global event trees. Which detection algorithm to apply in a particular case depends on application demands. The algorithms are currently being implemented using Modula-3 for Network Objects. This distributed programming system seems to be well suited for the implementation, because network objects provide a clean high-level semantics.

Future work aims

- to investigate applications, i.e. to design a "real" environment and to test it under different circumstances (e.g. number of sites, distribution of global event detectors, applied detection algorithm, network load).
- to consider the handling of event parameters with respect to the condition-part of ECA rules.
- to examine the design of ECA rules in distributed database systems, i.e. to show how ECA rules specified with respect to a global conceptual schema are mapped onto internal conceptual schemata, are distributed and are finally implemented.

Acknowledgements

The authors wish to thank Jean Bacon for helpful comments on earlier versions of this paper.

References

[Bac92] J. Bacon. *Concurrent Systems.* Addison-Wesley Publishing Company, 1992.

[Bat88a] P. Bates. Debugging heterogeneous distributed systems using event-based models of behaviour. *ACM SIGPLAN/SIGOPS*, 24(1):11–22, 1988.

[Bat88b] P. Bates. Distributed debugging tools for heterogeneous distributed systems. In *Proceedings of the 8th International Conference on Distributed Computing Systems, Washington, D.C.*, pages 308–315, June 1988.

[BBHM95] J.M. Bacon, J. Bates, R.J. Hayton, and K. Moody. Using events to build distributed applications. In *Proc IEEE Second International Workshop on Services in Distributed and Networked Environments (SDNE), Whistler, British Columbia*, pages 148–155, June 1995.

[BNOW94] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. Technical Report 115, Systems Research Center, Digital Equipment Corp., 1994.

[Cha89] S. Chakravarthy. Rule management and evaluation: An active dbms perspective. *SIGMOD RECORD*, 18(3):20–28, September 1989.

[CKAK94] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. Technical report, University of Florida, February 1994.

[CW92] S. Ceri and J. Widom. Production rules in parallel and distributed database environments. In *Proceedings of the 18th International Conference on Very Large Databases, Vancouver, British Columbia, Canada*, pages 339–351, August 1992.

[DBB88] U. Dayal, B. Blaustein, and A.P. Buchmann. The HiPAC project: Combining active databases and timing constraints. *SIGMOD RECORD*, 17(1), March 1988.

[DBM88] U. Dayal, A.P. Buchmann, and D.R. McCarthy. Rules are objects too: A knowledge model for an active, object-oriented database system. In *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Ebernburg, FRG.* Lecture Notes in Computer Science, volume 334, September 1988.

[Gat94] S. Gatziu. *Events in an Active Object-Oriented Database System.* PhD thesis, University of Zurich, Switzerland, 1994.

[GJS92] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Compose: A system for composite event specification and detection. In N.R. Adam and B. Bhargava, editors, *Advanced Database Concepts and Research Issues, LNCS*, 1992.

[HW88] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 166–175, 1988.

[HZMW91] D. Haban, S. Zhou, D. Maurer, and R. Wilhelm. Specification and detection of global breakpoints in distributed systems. Technical Report SFB 124-08/1991, Univ. Saarbrücken, Univ. Kaiserslautern, Germany, 1991.

[JS92] H.V. Jagadish and O. Shmueli. Composite events in a distributed object-oriented database. Technical Report att-db-92-11, AT& T Bell Laboratories, 1992.

[Kop92] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems, Yokohama, Japan*, pages 460–467, June 1992.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Mil91] D.L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1492, October 1991.

[Mul93] S. Mullender. *Distributed Systems.* Addison-Wesley Publishing Company, 1993.

[Sch95] S. Schwiderski. *Global Composite Events in Distributed Systems.* PhD thesis, University of Cambridge Computer Laboratory, Uk, 1995. in preparation.

[Spe91] M. Spezialetti. An approach to reducing delays in recognizing distributed event occurrences. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices*, 26(12):155–166, May 1991.

[Ver93] P. Verissimo. Real-time communication. In S. Mullender, editor, *Distributed Systems*, chapter 17, pages 447–490. Addison-Wesley Publishing Company, 1993.

[WF90] J. Widom and S.J. Finkelstein. Set-oriented production rules in relational database systems. *SIGMOD RECORD*, 19(2):259–270, June 1990.