Composite Events for Active Databases: Semantics, Contexts and Detection

S. Chakravarthy **V. Krishnaprasad** **E. Anwar** **S.-K. Kim**

Database Systems Research and Development Center
Computer and Information Sciences Department
University of Florida, Gainesville, FL 32611
email: {sharma, vk, emsa, skk}@cis.ufl.edu

Abstract

Making a database system active entails developing an expressive event specification language with well-defined semantics, algorithms for the detection of composite events, and an architecture for an event detector along with its implementation. This paper presents the semantics of composite events using the notion of a global event history (or a global event-log). Parameter contexts are introduced and precisely defined to facilitate efficient management and detection of composite events. Finally, an architecture and the implementation of a composite event detector is analyzed in the context of an object-oriented active DBMS.

1 Introduction

This paper focuses on the event component of the ECA (event-condition-action) rules used in active databases. An ECA rule consists, primarily, of three components: an event, a condition, and an action. A significant body of work exists on rules and rule processing in a DBMS. However, the event component of rules has received attention only recently [Mis91, CM94, GJS92b, GD93] and perhaps is the least understood compared to the condition and action components. Conditions and actions correspond to side-effect free queries and transactions, respectively.

Although event specification has been addressed in the literature primarily in the context of active databases, its applicability is not limited to active databases. An expressive event specification language and its detection can be used for analyzing event histories (or event logs) [SW92] in applications, such as stock trading, trend/demographic profile computation, and auditing (either as events occur or over stored event occurrences). Some aspects of knowledge discovery (e.g., determining events that lead to the stock market crash, understanding sequences of events leading to an earthquake) involve analyzing event patterns and their effect on various recorded observations. In other words, applications that examine cause-effect relationships need to specify and detect complex event patterns.

From the above, it is evident that support for rules needs to be complemented with an expressive event specification language. As an example, management of portfolios for various customers in a stock-trading application may require a rule of the form "when the DowJones average changes by 20% in any 2 hour interval, after reaching the value 3750, execute Trump's portfolio model to determine what to buy or sell". This rule requires not only the database events, but also temporal and composite events; this rule requires an expressive event specification language for modeling its events. As another example, detecting aperiodic occurrences of a pattern of events may indicate a potential money laundering scheme.

This paper extends earlier work on Snoop [Mis91, CM94] in several significant ways. Earlier work was primarily concerned with the motivation for the event language, classification of events, need for event operators, and the set of event operators. In this paper, we introduce primitive event sequences as ordered occurrences of a primitive event (termed primitive event history/event-log), and composite event history/event-log as a partial order of the *merged* primitive (or other composite) event histories. We define the semantics of primitive and composite events over an event history. We argue that the detection of composite events over a composite event history leads to monotonically increasing storage overhead as previous occurrences of events

---

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

cannot be deleted. To overcome this problem, we introduce the notion of *parameter contexts* as a mechanism for precisely restricting the occurrences that make a composite event occur as well as for computing its parameters. We have developed complete algorithms for detecting Snoop expressions in all parameter contexts.

Ode [GJS92b, GJS92a] and Samos [GD93, GD94] address event specification and detection in the context of active databases. Although there are some differences between Snoop, Ode, and Samos in the event specification language (for example, Samos has a Times operator for defining the occurrence of n events in an interval and Ode has a complement operator), they differ primarily in the mechanism used for event detection. Ode uses a finite automaton and Samos uses a labeled Petri Net. In contrast, we use an event graph where event occurrences (both primitive and composite) flow bottom-up from the nodes to their parents. The formulation of event histories as presented in this paper is different from that of Ode [GJS92a]. In addition, parameter contexts as well as event detection in various parameter contexts are extensions of our earlier work.

The rest of this paper is structured as follows. Section 2 provides semantics for the operators of Snoop [Mis91, CM94]. In section 3, we define global, primitive, and composite event histories and present the computation of composite events using the event history. In section 4, we define parameter contexts and illustrate event detection for all contexts. Section 5 provides an architecture for composite event detection in an object-oriented DBMS and highlights implementation choices. Section 6 contains conclusions.

2 Semantics of Snoop

For the purpose of this paper, we assume an equi-distant discrete time domain having "0" as the origin and each time point represented by a non-negative integer. We distinguish between an event, an event expression, and an event modifier.

An *event* is defined to be an instantaneous, atomic (happens completely or not at all) occurrence of interest at a point in time. In database applications, the interest in events comes mostly from the state changes that are produced by data manipulation operations.[1] Similar events can be grouped into an *event type*, and a type of events can be further classified into subtypes, resulting in an event type hierarchy as the class hierarchy. For instance, events of database update can be grouped into an event type *Update*, and further grouped into *Update-IBM*, *Update-DEC*, etc. Different event types are distinguished by different event type names.[2] Event of an event type may occur zero or more times over the time line; the time of occurrence of an event is denoted by *t_occ*. In addition, for simplicity, we assume that two occurrences of the same event type are not simultaneous. Furthermore, an event may causally precede or follow another, or events may be unrelated. For example, the two events *end-of-abort T1 and begin-of-rollback T1* must follow one another and are causally related (causally dependent), whereas the events *begin-of T1* and *begin-of T2* are causally independent and are said to be unrelated. An event is *definite* if and only if it is guaranteed to occur.

An event type is expressed by an *event expression*, which shall be discussed in the following subsection. Although an event is assumed to instantaneously occur at a time point, the event might be initiated at a prior time point, thus yielding a closed time interval between the start and end points (*t_occ* is the end point by default). A transaction event is such an example. In order to explicitly specify (or modify) the occurrence time of an event spanning a time interval, event modifiers *begin-of* and *end-of* were introduced in [CM94]. For inherently instantaneous events, the two event modifiers yield the same time point.

2.1 Primitive Events

Primitive events are those that are pre-defined in the system (and using the event modifiers). A mechanism for the detection is assumed to be available (refer to [AMC93] for details). Primitive events include database events, temporal events, and explicit events. *Database events* correspond to database operations, such as data manipulation operations, transactions, or methods in object-oriented databases. Temporal events are either *absolute* or *relative*. An absolute temporal event is specified with an absolute value of time, and represented as: $<$ *time string* $>$ using the format $<$ *(hh/mm/ss)mm/dd/yy* $>$. A relative temporal event also corresponds to a unique point on the time line but in this case both the reference point and the offset are explicitly specified. The reference point may be any event that can be specified in Snoop including an absolute temporal event. The syntax for a relative event is *event* + [*time string*]. Observe that the relative event subsumes the absolute event. However, the absolute version is retained for practical reasons. *Explicit events* are those events that are detected along with their parameters by application programs (i.e., outside the DBMS) and are only managed by the DBMS. Once registered with the system, they can be used as primitive events.

2.2 Event Expressions and Event Operators

Primitive events discussed so far are useful for modeling a number of applications. However, for many other applications, it is necessary to detect certain combinations of different events as a single event, i.e., a composite event. In this paper, a composite event is defined by applying an event operator to constituent events that are primitive or other composite events. In the absence

[1] Retrieval operations may also be regarded as events, although they do not change the database state.

[2] Or they may be distinguished by using parameters such as *Update(IBM)* and *Update(DEC)*.

of event operators, several rules are required to specify a composite event. Furthermore, some control information needs to be made a part of a rule specification.[3]

As mentioned before, an event type is denoted by an event expression. A primitive event (type) name itself is an event expression. If $E_1, E_2, \cdots, E_n$ are event expressions, an application of any event operator, described below, over the event expressions is an event expression. For an event E, *begin-of* E, *end-of* E, and (E) are all event expressions. If event modifier is omitted, *end-of* is assumed by default. The operator semantics described below assumes the *end-of* modifier.

2.2.1 Operator Semantics

An event E (either primitive or composite) is a function from the time domain onto the boolean values, True and False.

$$E : T \rightarrow \{\text{True}, \text{False}\}$$

given by

$$E(t) = \begin{cases} \text{T(rue)} & \text{if an event of type } E \text{ occurs} \\ & \text{at time point } t \\ \text{F(alse)} & \text{otherwise.} \end{cases}$$

We denote the negation of the boolean function E as $\sim E$. Given a time point, it computes the non-occurrence of an event at *that* point. The Snoop event operators[4] and the semantics of composite events formed by these event operators are as follows:

1. **OR** (∇): Disjunction of two events E_1 and E_2, denoted $E_1 \nabla E_2$, occurs when E_1 occurs or E_2 occurs. Formally,

$$(E_1 \nabla E_2)(t) = E_1(t) \vee E_2(t).$$

2. **AND** ($\triangle$): Conjunction of two events E_1 and E_2, denoted $E_1 \triangle E_2$, occurs when both E_1 and E_2 occur, irrespective of their order of occurrence. Formally,

$$\begin{aligned}(E_1 \triangle E_2)(t) &= (\exists t_1)\,(((E_1(t_1) \wedge E_2(t)) \\ &\quad \vee (E_2(t_1) \wedge E_1(t))) \wedge t_1 \leq t).\end{aligned}$$

Note that the OR and AND operators are commutative and associative:

$$(E_1 \nabla E_2)(t) = (E_2 \nabla E_1)(t)$$

$$((E_1 \triangle E_2) \triangle E_3)(t) = (E_1 \triangle (E_2 \triangle E_3))(t).$$

3. **ANY**: The conjunction event, denoted by $\text{ANY}(m, E_1, E_2, \cdots, E_n)$ where $m \leq n$, occurs when m events out of the n *distinct* events specified occur, ignoring the relative order of their occurrence. Formally,

$$\begin{aligned}&\text{ANY}(m, E_1, E_2, \cdots, E_n)(t) = \\ &\quad (\exists t_1)(\exists t_2) \cdots (\exists t_{m-1}) \\ &\quad (E_i(t_1) \wedge E_j(t_2) \wedge \cdots \wedge E_k(t_{m-1}) \wedge E_l(t) \\ &\quad \wedge (t_1 \leq t_2 \leq \cdots \leq t_{m-1} \leq t) \\ &\quad \wedge (1 \leq i, j, \cdots, k, l \leq n) \\ &\quad \wedge (i \neq j \neq \cdots \neq k \neq l)).\end{aligned}$$

For example,

$$\begin{aligned}&\text{ANY}(3, E_1, E_2, \cdots, E_n)(t) = (\exists t_1)(\exists t_2) \\ &\quad (E_i(t_1) \wedge E_j(t_2) \wedge E_k(t) \\ &\quad \wedge (t_1 \leq t_2 \leq t) \\ &\quad \wedge (1 \leq i, j, k \leq n) \wedge (i \neq j \neq k)).\end{aligned}$$

Also to specify m distinct occurrences of an event E, the following variant is provided:

$$\begin{aligned}&\text{ANY}(m, E^*)(t) = (\exists t_1)(\exists t_2) \cdots (\exists t_{m-1}) \\ &\quad (E(t_1) \wedge E(t_2) \wedge \cdots \wedge E(t_{m-1}) \wedge E(t) \\ &\quad \wedge (t_1 < t_2 < \cdots < t_{m-1} < t)).\end{aligned}$$

4. **SEQ** (;): Sequence of two events E_1 and E_2, denoted $E_1; E_2$, occurs when E_2 occurs provided E_1 has already occurred. This implies that the time of occurrence of E_1 is guaranteed to be less than the time of occurrence of E_2. Formally,

$$(E_1; E_2)(t) = (\exists t_1)\,(E_1(t_1) \wedge E_2(t) \wedge (t_1 < t)).$$

It is possible that after the occurrence of E_1, E_2 does not occur at all. To avoid this situation, it is desirable that definite events, such as end-of-transaction or an absolute temporal event, are used appropriately.

5. **Aperiodic Operators** (**A**, **A***): The Aperiodic operator **A** allows one to express the occurrences of an aperiodic event within a closed time interval. There are two versions of this event specification. The non-cumulative aperiodic event is expressed as $\text{A}(E_1, E_2, E_3)$, where E_1, E_2, and E_3 are arbitrary events. The event **A** is signaled each time E_2 occurs within the time interval started by E_1 and ended by E_3. Formally,

$$\begin{aligned}&\text{A}(E_1, E_2, E_3)(t) = (\exists t_1)(\forall t_2)\,(E_1(t_1) \wedge E_2(t) \\ &\quad \wedge (t_1 \leq t) \wedge ((t_1 \leq t_2 < t) \rightarrow \sim E_3(t_2))).\end{aligned}$$

On the other hand, the cumulative aperiodic event $\text{A}^*(E_1, E_2, E_3)$ occurs only once when E_3 occurs and accumulates the occurrences of E_2 within the open time interval formed by E_1 and E_3. This constructor is useful for integrity checking in databases and for collecting parameters of an event over an interval for

[3] In fact, in production rule systems (e.g., OPS5 [For82, FM87]), programs are written by incorporating a lot of control information as part of rules which have a form similar to an ECA rule. Specifically, in an OPS5 rule, events are not explicitly specified but are inferred for the worst case scenario.

[4] We denote the "disjunction", "conjunction", and "not" event operators as ∇, $\triangle$, and $\neg$, respectively. The symbols $\vee$, $\wedge$, and $\sim$ represent the "or", "and", and "not" boolean operators, respectively.

computing aggregates. As an example, highest or lowest stock price can be computed over an interval using this operator. Note that E_2 itself can occurs zero or more times within the interval and does not contribute to the occurrence of the composite event A*. Nonetheless, the parameters of A* will contain the parameters of E_2. Formally,

$$\mathrm{A}^*(E_1, E_2, E_3)(t) \;=\; (\exists t_1)\,(E_1(t_1) \wedge E_3(t) \wedge (t_1 < t)).$$

6. **Periodic Event Operators (P, P*)**: A periodic event is a temporal event that occurs periodically. A periodic event is denoted as $\mathrm{P}(E_1, \mathrm{TI}\,[:\mathit{parameters}], E_3)$ where E_1 and E_3 are events and $\mathrm{TI}\,[:\mathit{parameters}]$ is a time interval specification with optional parameter list. **P** occurs for every TI interval, starting after E_1 and ceasing after E_3. Parameters specified are collected each time **P** occurs. If not specified, the occurrence time of **P** is collected by default. The occurrence of **P** is formally defined as

$$\mathrm{P}(E_1, \mathrm{TI}\,[:\mathit{parameters}], E_3)(t) = (\exists t_1)(\forall t_2)\,(E_1(t_1) \wedge ((t_1 \le t_2 \le t) \rightarrow \sim E_3(t_2)) \wedge (t = t_1 + i * \mathrm{TI} \quad \text{for some integer } i \ge 1)).$$

P has a cumulative version **P*** expressed as $\mathrm{P}^*(E_1, \mathrm{TI}\!:\!\mathit{parameters}, E_3)$. Unlike **P**, **P*** occurs only once when E_3 occurs. Also, specified parameters are collected and *accumulated* at the end of each period and made available when **P*** occurs. Note that the parameter specification is mandatory in **P***. The occurrence of **P*** is formally defined as

$$\mathrm{P}^*(E_1, \mathrm{TI}\!:\!\mathit{parameters}, E_3)(t) = (\exists t_1)\,(E_1(t_1) \wedge E_3(t) \wedge (t \ge t_1 + \mathrm{TI}\,)).$$

7. **NOT** ($\neg$): The NOT operator, denoted $\neg(E_2)[E_1, E_3]$, detects the non-occurrence of the event E_2 in the closed interval formed by E_1 and E_3.[5] Formally,

$$\neg(E_2)[E_1, E_3](t) = (\exists t_1)(\forall t_2)\,(E_1(t_1) \wedge \sim E_2(t) \wedge E_3(t) \wedge ((t_1 \le t_2 < t) \rightarrow \sim(E_2(t_2) \vee E_3(t_2)))).$$

2.3 Examples

Below, we show some rules that entail detection of various composite events. We use a simplified syntax to make the events readable.

1. Sample IBM stock every 30 minutes from 8 a.m. to 5 p.m. each day. Event: P*(8 a.m., [30 mins]: IBM-stock-price, 5 p.m.)
2. When 4 withdrawals are made on an account in a day, do not allow further withdrawals. Event: A(8 a.m., ANY(4, withdraw-on-an-account*), 5 p.m.)
3. In a nuclear power plant if there is a change in fission rate followed by an increase in temperature, introduce moderator rods. Event: (fission-rate; temperature-increase)
4. Compute the new DowJones average when any two of IBM, DEC, or Boeing stock prices change during the day. Event: A(8 a.m., ANY(2, modify-IBM, modify-DEC, modify-Boeing), 5 p.m.)

[5] Note that this operator is different from that of !E, a unary operator in Ode [GJS92b], which detects the occurrence of any event other than E.

3 Histories and Event Logs

So far, we have defined the semantics of event operators over the time line in which only the time of (primitive or composite) event occurrences were recorded. However, detection of a composite event entails detecting not only the time at which the composite event occurs, but also the specific constituent event occurrences that make the composite event occur. In this section, we formally express the occurrence of a composite event E with respect to its constituent events that form part of the occurrence of E. At some level, the constituent events are primitive events.

We denote an occurrence of an event type E_j by e^i_j where i indicates the relative time of occurrence with respect to other occurrences of the same event. Composite events are represented as a *set of constituent event occurrences* within which the *order* of event occurrences is preserved. Note that it is possible for the same constituent event occurrence to be used for more than one occurrence of a composite event. The last event in the set is one whose occurrence made the composite event occur. The time of occurrence of a composite event is the time of occurrence of the last constituent event.

Global Event History/Event Log is a set of all primitive event occurrences and is denoted by H. Each primitive event occurrence is represented as a singleton set in the log.

$$H = \{\{e^i_j\} \mid \text{for all } j, \text{ primitive event } e^i_j \text{ has occurred at instance } i \text{ relative to events } E_j.\}$$

Primitive Event History/Event Log of the primitive event type E is a set of the occurrences of E present in the Global History H and is denoted by $E[H]$.

$$E_j[H] = \{\{e^i_j\} \mid \text{for all } i, \{e^i_j\} \in H\}.$$

Composite Event History/Event Log of a composite event E that has n constituent events $E_1, \cdots, E_n$ is a mapping from the global event history H to a subset of $E_1[H] \uplus \cdots \uplus E_n[H]$ where $\uplus$ is an operator that computes the cross product of two sets (whose elements are sets) and merges the elements of the cross product using the union operator. For example, given event histories $E_1[H] = \{\{e^1_1, e^4_3\}, \{e^2_1\}\}$ and $E_2[H] = \{\{e^1_2\}, \{e^2_2\}\}$,

$$E_1[H] \uplus E_2[H] \;=\; \{\{e^1_1, e^4_3, e^1_2\}, \{e^1_1, e^4_3, e^2_2\}, \{e^2_1, e^1_2\}, \{e^2_1, e^2_2\}\}.$$

Event Collection is a collection of all event occurrences of a particular type within a specified time interval. It is denoted by function $\wp$ as follows.

$$\wp(E, \mathit{start\_time}, \mathit{end\_time}) = \{e \mid \{e\} \in E[H] \text{ and } \mathit{start\_time} \leq \text{t\_occ}(e) \leq \mathit{end\_time}\}.$$

Given a global event history, the event history for an arbitrary composite event formulated using the operators defined in section 2.2 can be computed. Below, we define these computations formally. This formulation will compute all occurrences of a composite event (along with participating constituent event occurrences) for a finite H. This is termed the *unrestricted context*. The operators $\uplus$, ∇, $\triangle$ are all left associative.

3.1 The Unrestricted Context

1. $(E_1 \nabla E_2)[H] = \{e \mid e \in E_1[H] \cup E_2[H]\}$.

2. $(E_1 \triangle E_2)[H] = \{\{e^i, e^j\} \mid \{e^i, e^j\} \in ((E_1[H] \uplus E_2[H]) \cup (E_2[H] \uplus E_1[H]))$ and $\text{t\_occ}(e^i) \leq \text{t\_occ}(e^j)\}$.

3. $\text{ANY}(m, E_1, E_2, \cdots, E_n)[H] = \{\{e^i, e^j, \cdots, e^k\} \mid \text{t\_occ}(e^i) < \text{t\_occ}(e^j) < \cdots < \text{t\_occ}(e^k)$ and $|\{e^i, e^j, \cdots, e^k\}| = m \leq n$ and $\{e^i, e^j, \cdots, e^k\} \in \mathcal{P}$ (element) $\}$
   where $\mathcal{P}$ is the power set and element is a member of the set: $E_{\pi_1}[H] \uplus E_{\pi_2}[H] \uplus \cdots \uplus E_{\pi_m}[H]$; each π_i (permutation) can be any i from 1 to n with the restriction that each E participating in the merged-cartesian product ($\uplus$) is distinct.

   $\text{ANY}(m, E^*)[H] = \{\{e^i, e^j, \cdots, e^k\} \mid \text{t\_occ}(e^i) < \text{t\_occ}(e^j) < \cdots < \text{t\_occ}(e^k)$ and $|\{e^i, e^j, \cdots, e^k\}| = m \leq n$ and $\{e^i, e^j, \cdots, e^k\} \in \mathcal{P}(E[H])\}$.

4. $(E_1; E_2)[H] = \{\{e^i, e^j\} \mid \text{t\_occ}(e^i) < \text{t\_occ}(e^j)$ and $\{e^i, e^j\} \in E_1[H] \uplus E_2[H]\}$.

5. $\text{A}(E_1, E_2, E_3)[H] = \{\{e^i, e^j\} \mid \text{t\_occ}(e^i) < \text{t\_occ}(e^j)$ and $\{e^i, e^j, e^k\} \in E_1[H] \uplus E_2[H] \uplus E_3[H]\}$.

6. $\text{P}(E_1, \text{TI}, E_3)[H] = \{\{e^i, t\} \mid$ for all $\{e^i, e^k\} \in E_1[H] \uplus E_3[H]$ and $\text{t\_occ}(e^i) < \text{t\_occ}(e^k)$, $t = \text{t\_occ}(e^i) + j * \text{TI}$ for integer $j \geq 1$ and $t \leq \text{t\_occ}(e^k)\}$.

7. $\neg(E_2)[E_1, E_3][H] = \{\{e^i, e^k\} \mid \{e^i, e^k\} \in E_1[H] \uplus E_3[H]$ and $\wp(E_2, \text{t\_occ}(e^i), \text{t\_occ}(e^k)) = \emptyset\}$.

The definition of the cumulative operators include the accumulation of event occurrences over an interval. This requires the function $\wp$ to collect the appropriate occurrences. A* and P* are defined below using $\wp$.

8. $\text{A}^*(E_1, E_2, E_3)[H] = \{\{e^i, \wp(E_2, \text{t\_occ}(e^i), \text{t\_occ}(e^k)), e^k\} \mid e^i \in E_1[H]$ and $e^k \in E_3[H]$ and $\text{t\_occ}(e^i) < \text{t\_occ}(e^k)\}$.

9. $\text{P}^*(E_1, \text{TI}, E_3)[H] = \{\{e^i, \tau, e^k\} \mid$ for all $\{e^i, e^k\} \in E_1[H] \uplus E_3[H]$ and $\text{t\_occ}(e^i) < \text{t\_occ}(e^k)$, $\tau = \{t \mid t = \text{t\_occ}(e^i) + j * \text{TI}$ for integer $j \geq 1$ and $t \leq \text{t\_occ}(e^k)\}\}$.

Below, we illustrate the computation of a composite event X on a global history H according to the above definitions of operators in the unrestricted context. The event X is drawn from the stock market applications. The interpretation of constituent events of X is, E_1: opening of stock market, E_2: change in Dow Jones average, E_3: change in the price of IBM stock, and E_4: change in a commodity which depends on IBM stock.

$$X = ((E_1 \triangle E_2); E_3; (E_2 \triangle E_4))$$
$$H = \{\{e_1^1\}, \{e_1^2\}, \{e_2^1\}, \{e_3^1\}, \{e_2^2\}, \{e_4^1\}, \{e_3^2\}, \{e_4^2\}\}$$

$$\begin{aligned}
E_1[H] &= \{\{e_1^1\}, \{e_1^2\}\} \\
E_2[H] &= \{\{e_2^1\}, \{e_2^2\}\} \\
E_3[H] &= \{\{e_3^1\}, \{e_3^2\}\} \\
E_4[H] &= \{\{e_4^1\}, \{e_4^2\}\}
\end{aligned}$$

$$\begin{aligned}
(E_1 \triangle E_2)[H] &= \{\{e_1^1, e_2^1\}, \{e_1^1, e_2^2\}, \{e_1^2, e_2^1\}, \{e_1^2, e_2^2\}\} \\
(E_2 \triangle E_4)[H] &= \{\{e_2^1, e_4^1\}, \{e_2^1, e_4^2\}, \{e_2^2, e_4^1\}, \{e_2^2, e_4^2\}\}
\end{aligned}$$

$$\begin{aligned}
X[H] = \{&\{e_1^1, e_2^1, e_3^1, e_2^1, e_4^1\}, \{e_1^1, e_2^1, e_3^1, e_2^1, e_4^2\}, \\
&\{e_1^1, e_2^1, e_3^1, e_2^2, e_4^1\}, \{e_1^1, e_2^1, e_3^1, e_2^2, e_4^2\}, \\
&\{e_1^1, e_2^1, e_3^2, e_2^1, e_4^2\}, \{e_1^1, e_2^1, e_3^2, e_2^2, e_4^2\}, \\
&\{e_1^1, e_2^2, e_3^2, e_2^1, e_4^2\}, \{e_1^1, e_2^2, e_3^2, e_2^2, e_4^2\}, \\
&\{e_1^2, e_2^1, e_3^1, e_2^1, e_4^1\}, \{e_1^2, e_2^1, e_3^1, e_2^1, e_4^2\}, \\
&\{e_1^2, e_2^1, e_3^1, e_2^2, e_4^1\}, \{e_1^2, e_2^1, e_3^1, e_2^2, e_4^2\}, \\
&\{e_1^2, e_2^1, e_3^2, e_2^1, e_4^2\}, \{e_1^2, e_2^1, e_3^2, e_2^2, e_4^2\}, \\
&\{e_1^2, e_2^2, e_3^2, e_2^1, e_4^2\}, \{e_1^2, e_2^2, e_3^2, e_2^2, e_4^2\}\}
\end{aligned}$$

As can be visualized, there are 16 occurrences of the event X for the given history. It is not clear whether all these occurrences will be useful in all applications. We strongly believe that an application would be interested in a subset of these events that are meaningful to the semantics of that application. Furthermore, different applications may be interested in different subsets. In the next section, we propose parameter contexts as a way of imposing meaningful restrictions of the composite event history generated for an event.

Note that the detection a composite event in the unrestricted context may warrant keeping all event occurrences (especially for ;, Any, and $\triangle$ operators) and hence poses practical problems for the management of event history and detection. In most applications, either the time interval within which the events need to be detected or the relevance of multiple occurrences of the same event is derived from the application semantics. Hence, only a subset of the events detected in the unrestricted context is likely to be meaningful.

4 Composite Event Detection

Events can always be detected and parameters computed using the *unrestricted context* presented in the previous section. However, the unrestricted context produces a large number of event occurrences and not all occurrences may be meaningful from the point of view of an application. Moreover, the computation and space overhead associated with the detection of events in this context can be substantial.

In this section, we refine parameter contexts introduced in [CM94] for the purpose of reducing the space and computation overhead associated with the detection of composite events and providing a mechanism for choosing a meaningful subset of event occurrences generated by the unrestricted context. Parameter contexts serve the purpose of detecting and computing the parameters of composite events in different ways to match the semantics of applications. The choice of a parameter context also suggests the complexity of event detection and storage requirements for a given application.

The detection of a composite event may require the detection of one or more constituent events as well as one or more occurrences of a constituent event. Events requiring multiple event occurrences (either of the same type or of different types) for the detection of a composite event, give rise to alternate ways of computing the history as well as parameters, as the events are likely to occur several times over an interval.

The occurrence of a composite event is marked by the occurrence of a constituent event that makes the composite event occur (using the *end-of* event modifier semantics). This constituent event is termed the *terminator* of the composite event. Several constituent events can act as terminators, but there is at least one terminator event for a given composite event. Analogously, there is always a constituent event that initiates the occurrence of a composite event. This constituent event is termed the *initiator* of the composite event. There may be more than one initiators for a composite event. For a primitive event, the primitive event itself is the terminator and initiator.

The composite event detector needs to record the occurrences of each constituent event and save its parameters so that they can be used to compute the parameter set of the composite event. Consider the following event expressions:

$A = (E_1 \triangle E_2); E_3$
$B = E_1 \nabla E_2 \nabla E_3$
$C = E_1; \mathrm{ANY}(2, E_2, E_3)$

where E_1, E_2, and E_3 are primitive events. Event A is detected when E_3 occurs provided both E_1 and E_2 have already occurred in any order. Event B is signaled each time an instance of any of the three events E_1, E_2 or E_3 occurs. Parameters of event A (as well as C) include parameters of all the three events E_1, E_2 and E_3 whereas the parameters of event B include only the parameters of one of its events. Both E_1 and E_2 can be initiators of A and E_3 is the only terminator. For C, E_1 is the initiator and both E_2 and E_3 can be terminators. Figure 1 shows a global event history in which four types of events E_1, E_2, E_3, and E_4 occur, as well as the event graph for the composite event A.

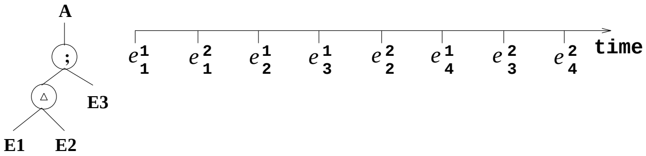

Figure 1: Global event history

4.1 Parameter Contexts

The parameter contexts proposed below are motivated by a careful analysis of several classes of applications. We have identified four parameter contexts that are useful for a wide range of applications. Below, we indicate the characteristics of the applications that motivated our choice of parameter contexts:

1. Applications where the events are happening at a fast rate and multiple occurrences of the same type of event only refine the previous data value. In other words, the effect of the occurrence of several events of the same type is subsumed by the most recent occurrence. This is typical of sensor applications (e.g., hospital monitoring, global position tracking, multiple reminders for taking an action).

2. Applications where there is a correspondence between different types of events and their occurrences and this correspondence needs to be maintained. Applications that exhibit causal dependency (e.g., between aborts, rollbacks, and other operations; between bug reports and releases; start of a transaction and its end) come under this category.

3. Trend analysis and forecasting applications (e.g., securities trading, stock market, after-the-fact diagnosis) where composite event detection along a moving time window needs to be supported. For example, computing change of more than 20% in DowJones

average in any 2 hour period requires each change to initiate a new occurrence of an event. This corresponds to the initiation of the detection of an event for each distinct occurrence.

4. Applications where multiple occurrences of a constituent event needs to be grouped and used in a meaningful way when the event occurs. This context is useful in applications where an event is terminated by a deadline-event and all occurrences of constituent events are meaningful up to the occurrence of the deadline event. For example, in a banking application we might want to keep track of the amount of withdrawals and deposits performed in a day and use it to update the account balance at the end of the day.

We introduce the following contexts for the classes of applications described above. These contexts are precisely defined using the notion of initiator and terminator events. We explain the contexts using the composite event A which is a constituent event of the composite event X in the previous section. We are not concerned with occurrences e_4^1 and e_4^2 as event E_4 is not part of the event expression of A.

- **Recent:** In this context, only the most recent occurrence of the initiator for any event that has started the detection of that event is used. When an event occurs, the event is detected and all the occurrences of events that cannot be the initiators of that event in the future are deleted (or flushed). For example, in the *recent* context, parameters of event A will include the event instances $\{e_1^2, e_2^1, e_3^1\}$ (A is detected when e_3^1 occurs) and $\{e_1^2, e_2^2, e_3^2\}$ (when A is detected again when e_3^2 occurs). In this context, *not all occurrences* of a constituent event will be used in detecting a composite event. Furthermore, an initiator of an event (primitive or composite) will continue to initiate new event occurrences until a new initiator occurs.
- **Chronicle:** In this context, for an event occurrence, the initiator, terminator pair is unique. The oldest initiator is paired with the oldest terminator for each event (i.e., in chronological order of occurrence). When a composite event is detected, its parameters are computed by using the oldest occurrence of each constituent event. However, once used occurrences of the constituent events cannot participate in any other occurrences of the composite event. For example, parameters of event A in the chronicle context will be computed by using event instances $\{e_1^1, e_2^1$ and $e_3^1\}$. When the next E3 type event occurs at e_3^2, then the A will be detected with the instances $\{e_1^2, e_2^2, e_3^2\}$.
- **Continuous:** In this context, each initiator of an event starts the detection of that event. A terminator event occurrence may detect one or more occurrences of the same event. This context is especially useful for tracking trends of interest on a sliding time point governed by the initiator event. In Figure 1, each of the occurrences e_1^1 and e_1^2 (as well as e_2^1 and e_2^2) would start the detection of the event A. The first occurrence of A will have the instances $\{e_1^1, e_2^1, e_3^1\}$. The second occurrence of A will consist of $\{e_1^2, e_2^1, e_3^1\}$. In this context, an initiator will be used *at least once* for detecting that event.

  There is a subtle difference between the chronicle and the continuous contexts. In the former, pairing of the initiator is with a unique terminator of the event whereas in the latter multiple initiators are paired with a single terminator of that event.
- **Cumulative:** In this context, for each constituent event, all occurrences of the event are accumulated until the composite event is detected. Whenever a composite event is detected, all the constituent events occurrences that are used for detecting that composite event are deleted. For example, parameters of event A will include all the instances of each event up to e_3^1 when it occurs. The entire instances shown in Figure 1 (except e_2^2, and e_3^2) is the set of occurrences that make the composite event A. Unlike the continuous context, an event occurrence does not participate in two distinct occurrences of the same event in the cumulative context.

Observe that the cumulative context described above cannot be generated as a subset of the event history generated by the unrestricted context. The notion of accumulation of event occurrences is not present in the unrestricted context. For this reason, the definitions of A* and P* used the function $\wp$ which accumulates a set of event occurrences of a specific type over a given interval.

Although contexts described above restrict the set of event occurrences generated, they are based on the use of initiator, terminator pair in different ways. In addition to the above contexts, it may be useful to detect composite events over non-overlapping time intervals. That is for any two occurrences of an event W, the t_occ of the initiator is greater than the t_occ of the terminator of the immediately preceding occurrence of W. This notion of the use of non-overlapping intervals can be applied to any of the contexts described in this paper, including the unrestricted context. This can be easily seen from the Figure 2. For instance, all events detected in recent, chronicle, and continuous contexts are not disjoint. If disjoint detection of event occurrences were to be specified for the example shown in Figure 2, only the first occurrences of events in each context (i.e., 1, 3, 5, and 9) would be detected.

Based on the above definitions of contexts, several observations can be made. Disjoint continuous context is the same as disjoint chronicle context. Also, cumulative context always generates occurrences that satisfy the disjoint specification. In other words, disjoint cumulative context is equivalent to cumulative context.

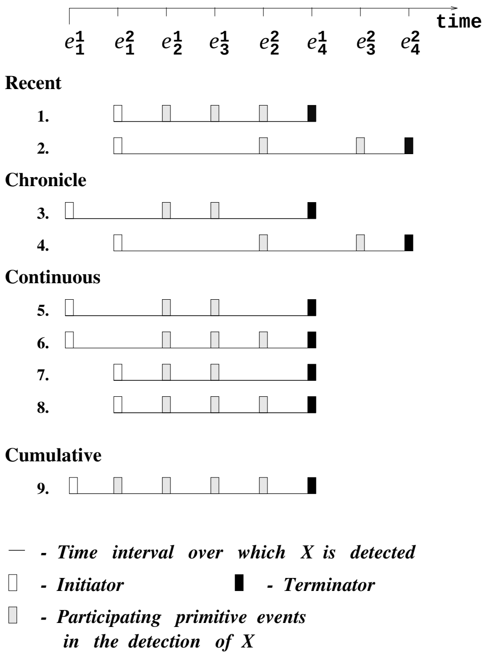

Figure 2: Illustration of event detection in various contexts for the expression $X = (E_1 \triangle E_2; E_3; E_2 \triangle E_4)$

4.2 Illustration of Composite Event Detection

The approach taken for composite event detection in this paper is different from the approaches taken in Ode and Samos. Samos defines a mechanism based on Petri Nets for modeling and detection of composite events for an OODBMS. They use modified colored Petri nets called SAMOS Petri Nets to allow flow of information about the event parameters in addition to occurrence of an event. It appears that common subexpressions are represented separately leading to duplication of Petri Nets. Furthermore, although not stated explicitly, Samos detects events only in the chronicle context described in this paper. Ode uses an extended finite automata for composite event detection. Their extended automaton, makes a transition at the occurrence of each event in the history like a regular automaton and in addition to that it looks at the attributes of the events, and also computes a set of relations at the transition. The definitions of And and Pipe operators on event histories do not seem to produce the desired result.

We use an event tree for each composite event and these trees are merged to form an event graph for detecting a set of composite events. This will avoid the detection of common sub-events multiple times thereby reducing storage requirements. Primitive event occurrences are injected at the leaves and flow upwards analogous to a data-flow computation. Furthermore, the commonality of representation between event detection and query optimization using operator trees allow us to combine both, and optimize a situation (event-condition pair) as a unit. This is certainly possible in the relational model as transformations can be applied to push predicates from conditions to and apply them during event detection as part of the optimization (in contrast, event masks are specified in Ode by the user). Finally, the combination of event-condition trees will allow conditions to be evaluated on a demand basis avoiding unnecessary computations. In summary, our formulation of event detection readily lends itself to optimization techniques used in databases.

The introduction of parameter contexts adds another perspective to the detection of composite events. From Figure 2 it is easier to understand how each parameter context detects different instances of the same composite event for a given sequence of primitive event occurrences. In this section we will use one event graph and discuss how we compute the constituent events of a composite event for each of the parameter contexts. Algorithms for detecting composite events in different contexts and their implementation are detailed in [Kri94]. The time line indicates the relative order of the primitive events with respect to their time of occurrences. All event propagations are done in a bottom-up fashion. The leaves of the graphs have no storage and hence pass the primitive events directly to their parent nodes. The operator nodes have separate storage for each of their children. The graphs shown in Figure 3 for the various contexts are at a time point when primitive event e_4^1 is detected. The different instances of the same event are stored as separate entries and are shown in separate lines in the figure. Since the leaves do not have any storage, the primitive event e_4^1 is passed to the parent of leaf E_4. The arrows pointing from the child node to its parent in the graph indicates the detection and flow of the events.

In the recent context $\{e_2^2, e_4^1\}$ is sent to node A since e_2^2 and e_4^1 are the most recent initiator and terminator of the AND operator (node C). Since the terminator e_4^1 can serve as an initiator for node C (according to the semantics of AND), it is not discarded. At node A the initiator is already present and $\{e_2^2, e_4^1\}$ serves as the terminator. So event X is detected with $\{e_1^2, e_2^1, e_3^1, e_2^2, e_4^1\}$. Here since the terminator cannot serve as the initiator it is discarded and only $\{e_1^2, e_2^1, e_3^1\}$ which is the most recent initiator of X is retained at node A.

In the case of Chronicle context, e_2^1 is the oldest initiator of node C and it is at the head of the initiator list. Hence e_4^1 is paired with e_2^1 and $\{e_2^1, e_4^1\}$ is passed to node A. Once they are passed, unlike the recent context, both the initiator and the terminator are discarded. Hence node C retains only e_2^2 after AND is detected. Event X is detected with $\{e_1^2, e_2^1, e_3^1, e_2^1, e_4^1\}$ at node A and both $\{e_1^2, e_2^1, e_3^1\}$ and $\{e_2^1, e_4^1\}$ are deleted.

Continuous context involves lot of storage overhead for event detection. As in the chronicle context we retain all the initiators signalled so far in each of the nodes. But unlike chronicle context, the terminator is paired with each of the initiators present and all the

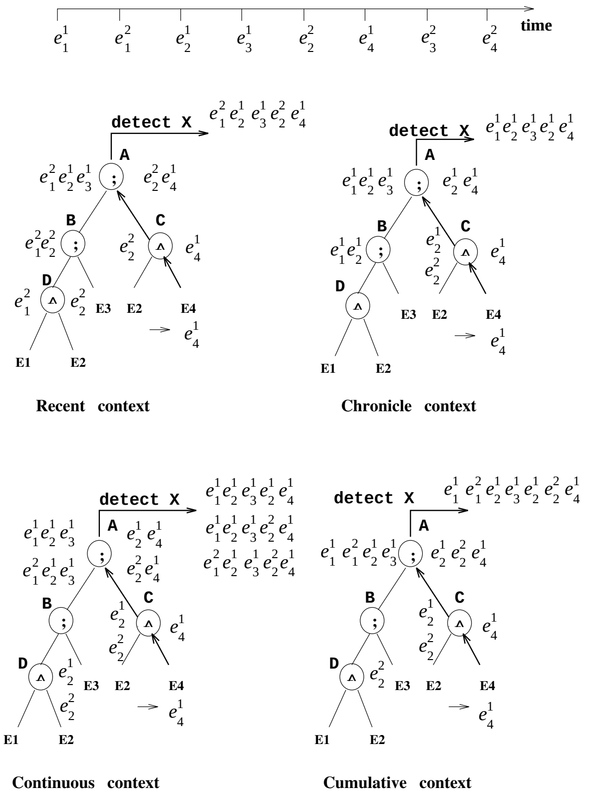

Figure 3: Event detection in various contexts

initiators are deleted after the detection of the composite event. We retain the terminator only if it can serve as an initiator for future detection of the composite event. At any point of time, the terminator of the composite event X in all the other contexts will signal only one occurrence of event X, whereas in the continuous context it will generate multiple occurrences of X. In our example e_2^1, e_2^2 are the initiators at node C. Both of them are paired with e_4^1 to generate two occurrences of the AND at the same point of time, namely $\{e_2^1, e_4^1\}$, $\{e_2^2, e_4^1\}$. Since e_4^1 can serve as an initiator for node C in the detection of a new occurrence of the constituent event, we retain it and both the initiators e_2^1, e_2^2 that have been paired are deleted. At node A, there are two initiators already present and the two terminators signalled from node C lead to four instances of the detection of event X with the same time of occurrence. Among the four contexts presented, the continuous context generates a larger subset of the event occurrences identified by the unrestricted case.

In the cumulative context, unlike the continuous context, all the initiator occurrences available so far are combined with the terminator and only one occurrence of X is detected. In our example, e_2^1, e_2^2 are combined together as one initiator and $\{e_2^1, e_2^2, e_4^1\}$ is sent to parent node A. Similarly, node A detects X with $\{e_1^1, e_1^2, e_2^1, e_3^1, e_2^1, e_1^2, e_4^1\}$. Once detected the unified initiator and terminator is discarded.

4.3 Storage Requirements

Parameter contexts described in this paper simplify the event detection as well as the computation of parameters as compared to the unrestricted context.

Some of the parameter contexts, such as continuous and chronicle, impose more storage requirements than the recent and cumulative contexts. The recent parameter context can be implemented using a fixed size buffer for each event (i.e., at each node of the event graph). This is because only the parameters for the most recent occurrence of an event is stored and hence requires the least amount of storage. For the chronicle context, a queue is required and the amount of storage needed is dependent upon the duration of the interval of the composite event and the frequency of event occurrences within that interval. Similarly, for the continuous context, the storage requirements can be excessive, implying that the choice of the parameter context for each rule needs to be made judiciously. The cumulative context, unlike the continuous and chronicle contexts, combines all initiators and hence at each node there is only one whole initiator combination. Though both continuous and chronicle maintain a list of initiators, only continuous can signal more than one occurrence of a composite event for a single terminator. Since this composite event might be a constituent event of another larger expression, the continuous parameter context requires considerable storage compared to any other parameter context. The storage requirements can be excessive for the cumulative context also. However, based on the semantics of the parameter contexts, the storage requirement increases monotonically from recent to cumulative to chronicle to continuous to unrestricted. This is because all the event occurrences used in the detection of a composite event are deleted when the event is detected in the cumulative context whereas in the chronicle context, initiator and terminator event occurrences are paired in the order of occurrences and hence more events are stored for longer duration. Application of the disjoint modifier, on any context (except the cumulative), further reduces the storage requirements by allowing events to be discarded earlier.

5 Active OODBMS Architecture

It is useful to examine the requirements of rule processing in active databases before presenting an architecture. Broadly, the requirements are:

5.1 Support for Events

- *Primitive and Composite event detection:* Any method of any object class is a potential primitive event. Further we permit before- and after-variants of method invocation as events. Composite events are formed by applying a set of operators to primitive events and composite events. Both primitive and composite events need to be detected by the system. The detection of composite events entails not only the time at which the composite event occurs, but also keeping track of the constituent event occurrences.

- *Parameter computation:* The parameters of a primitive event corresponds to the parameters of the method declared as a primitive event. The processing of composite events entails not only its detection, but also the computation of the parameters associated with a composite event. The parameters of a composite event need to be collected, recorded and passed on to condition and action portions of a rule by the event detector. Furthermore, these parameters need to be recorded in such a way that they can be interpreted by the condition and action components of a rule.

- *Online and batch detection of composite events:* The composite event detector needs to support detection of events as they happen (online) when it is coupled to an application or over a stored event-log (in batch mode).

- *Inter-application (global) events:* In addition to rules based on events from within an application, it is useful to allow composite events whose constituent events come from different applications. This is especially useful for cooperative transactions and workflow applications. This entails detection of events that span several applications.

5.2 Support for Rules

- *Multiple rules:* An event (primitive as well as composite) can trigger several rules. Hence, it is necessary to support a rule execution model that supports concurrent as well as prioritized rule execution.

- *Nested rules:* When rule actions raise events which trigger other rules there is nested execution of rules. Rules can be nested to arbitrary levels.

- *Coupling modes:* The three coupling modes (immediate, deferred and detached) discussed in HiPAC were introduced to support application needs. Sentinel architecture should be able to support all of them.

- *Rule scheduling:* In the presence of multiple rules and nested execution, the architecture need to support prioritized serial execution of rules, concurrent execution of all rules, or a combination of the two. Further, the system, should allow the application designer to choose from among the above alternatives.

The above requirements as well as the OO model into which active capability is being incorporated affect the design of both the rule processing subsystem and the event detector. Below, we present the Sentinel architecture in terms of extensions to the Open OODB system and discuss how the above requirements are supported in our current implementation.

5.3 Sentinel Architecture

The Sentinel architecture proposed in this section extends the *passive* Open OODB system [Ins93].

In order to satisfy the above requirements in an object-oriented framework, we propose the architecture shown in Figure 4 which is being implemented as an extension to the Open OODB Toolkit developed at Texas Instruments. Our proposed architecture relies on the use of threads (or light weight processes) for separating event detection from application execution in a transparent manner.

Our primitive event detection is based on the design proposed in [AMC93]. Primitive events are signaled by adding a notify procedure call in the wrapper method by Sentinel. Also, appropriate calls for the parameter collection are added at this stage. Both primitive and local composite events are signaled as soon as they are detected. However, the detection of a composite event may span a time interval as it involves the detection and grouping of its constituent events in accordance with the parameter context specified. A clean separation of the detection of primitive events (as an integral part of the database) from that of composite events allows one to: i) implement a composite event detector as a separate module (as has been done) and ii) introduce additional event operators without having to modify the detection of primitive events.

Each application has a local composite event detector (Figure 4) to which all primitive events are signaled. Our implementation uses threads (light weight processes), instead of processes, for separating composite event detection (as well as for the execution of rules) from application. When a primitive event occurs it is sent to the local composite event detector and the application waits for the signaling of a composite event that is detected in the *immediate* mode. The local composite event detector and the application share the same address space and our event detector uses an event graph similar to operator trees.

Parameter computation for composite events raises additional problems in the object-oriented framework. The lack of a single data structure (such as a relation) makes it extremely difficult to identify and manage parameter computation even within an application. As a first cut, we include the identification of the object (i.e., oid) as one of the event parameters and other parameters which have atomic values. However, no assumptions are made about the state of the object (when the oid is passed as part of a composite event) as the detection of a composite event is over a time interval. A linked list that contains the parameters of each primitive event (as a list) that participates in the detection of the composite event is computed and passed to the rule associated with that event. Complete support for parameters of composite events may require versioning of objects and related concurrency control and recovery techniques.

A rule specified to be executed in the deferred mode

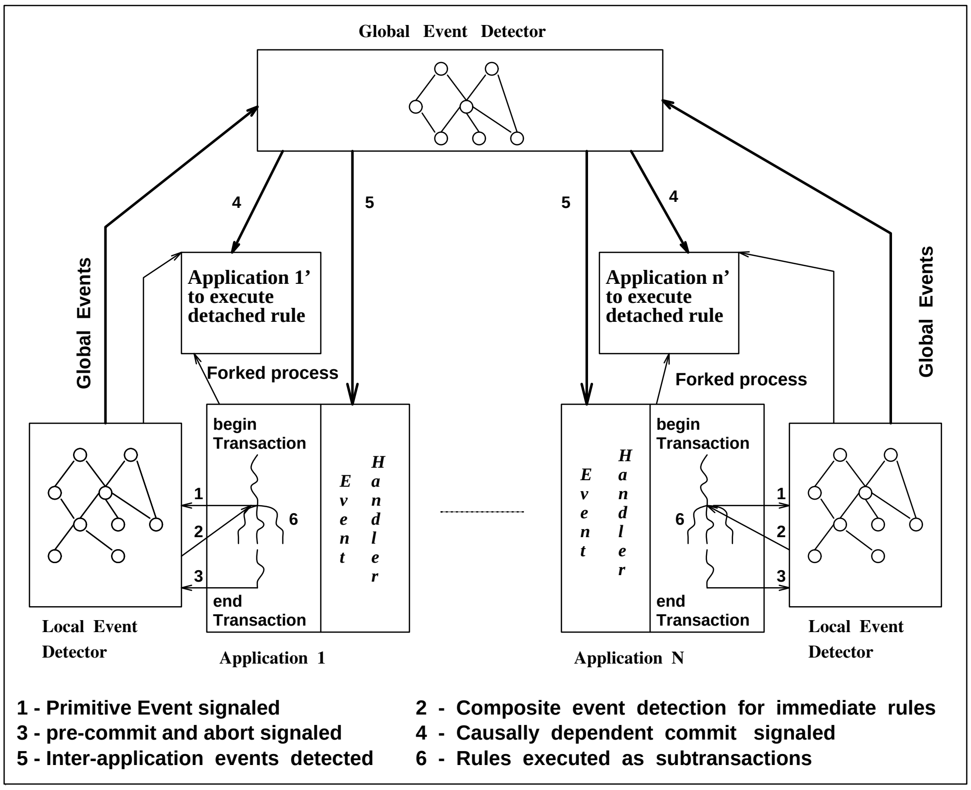

Figure 4: Sentinel architecture

is rewritten at the source code level into a rule in immediate mode by the Sentinel pre-processor. Our event specification language Snoop [CM94] supports a number of operators of which A^* monitors the cumulative effect of an event occurrence within a specified interval. For example, if we need to accumulate all insert events in a transaction, we can specify the event as A^*(begin_transaction, insert, end_transaction). Using this operator, we model the deferred coupling mode in terms of the immediate coupling mode by using *begin* and *pre-commit* transaction events and postpone the execution of the rule to the end of the transaction. In Sentinel, the begin transaction event is always signalled at the beginning of a transaction and the pre-commit is signalled before the commit of a transaction. Using the A^* operator, a rule in deferred mode with an (arbitrary) event E is transformed by the Sentinel pre-processor to A^*(begin-transaction, E, pre-commit-transaction). This causes a deferred rule to be executed exactly once even though its event may be triggered a number of times in the course of that transaction execution.

For rule execution, a nested transaction manager is implemented with its own lock manager. This is in addition to the concurrency control and recovery provided by the Exodus for top-level transactions. Each rule (i.e., condition and action portions of a rule) is packaged into a subtransaction. A number of subtransactions are spawned as a part of the application process. This is further elaborated in [CKTB94]. Support for multiple rule execution and nested rule execution entails that the event detector be able to receive events detected within a rule's execution in the same manner it receives events detected in a top level transaction. This is accomplished relatively easily by separating the local composite event detection from the application as shown in Figure 4. This separation also readily supports both online and batch (or after-the-fact) detection of composite events.

Finally, in the presence of composite events, it is possible for the events to cross transaction boundaries (within the same application). Currently, we provide a mechanism to flush all events generated by a transaction when it commits. More work is required to understand the semantics of rule execution whose events span transaction boundaries.

6 Conclusions

This paper significantly extends our earlier work on an expressive event specification language. We have provided a declarative semantics of each operator of Snoop. We introduced the notion of global event history and local event history for defining the computation of participating events for an arbitrary composite event expression. We refined the parameter contexts introduced earlier using initiator and terminator events. We have illustrated the detection of composite events in various contexts and proposed an architecture for its implementation in an object-oriented framework along with a discussion of the various issues involved. Finally, algorithms for all parameter contexts have been developed and implemented using the architecture shown in this paper.

In this paper, we are assuming that the parameters of an event can be computed once the event occurrences are known. It is useful, however, to explicitly introduce (as a minimum) the identification of the object (i.e., oid) for which the primitive event is applicable. This can be done by specifying, for each primitive event, a parameter which is either a constant or a variable representing the oid. For example, the primitive event Change_price(IBM) indicates that the event occurs when the method Change_price is executed for the IBM object. As another example, Change_price(X);Change_price(X) refers to the sequence of events on the same oid X. And Change_price(X);Change_price(Y) refers to the sequence of events on two different oid's. Detailed discussion of parameter computations are beyond the scope of this paper. Some of these issues have been discussed in section 5. All the event detection algorithms we have developed extend readily when the oid is allowed as an explicit parameter of a primitive event.

Acknowledgements

This work is supported, in part, by the National Science Foundation IRI-9011216, by the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division, and by the Rome Laboratory.

References

[AMC93] E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *Proceedings, International Conference on Management of Data*, pages 99–108, Washington, D.C., May 1993.

[CKTB94] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. Technical Report UF-CIS-TR-93-039, University of Florida, E470-CSE, Gainesville, FL 32611, Feb. 1994. (Submitted for publication.).

[CM94] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 1994. (To appear).

[FM87] C. L. Forgy and J. McDermott. Domain-Independent Production System Language. In *Proceedings Fifth International Conference on Artificial Intelligence*, Cambridge, MA, 1987.

[For82] C. L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem. *Artificial Intelligence 19*, pages 17–37, 1982.

[GD93] S. Gatziu and K. R. Dittrich. Events in an Object-Oriented Database System. In *Proc. of the 1st International Conference on Rules in Database Systems*, September 1993.

[GD94] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Databases Using Petri Nets. In *Proc. of the 4th International Workshop on Research Issues in data Engineering: Active Database Systems*, pages 2–9, February 1994.

[GJS92a] N. H. Gehani, H. V. Jagadish, and O. Shmueli. COMPOSE A System For Composite Event Specification and Detection. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, December 1992.

[GJS92b] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event Specification in an Object-Oriented Database. In *Proceedings, International Conference on Management of Data*, pages 81–90, San Diego, CA, June 1992.

[Ins93] Texas Instruments. Open OODB Toolkit, Release 0.2 (Alpha) Document, September 1993.

[Kri94] V. Krishnaprasad. Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, March 1994.

[Mis91] D. Mishra. SNOOP: An Event Specification Language for Active Databases. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, August 1991.

[SW92] A. Prasad Sistla and O. Wolfson. Temporal Triggers in Active Databases. Technical report, University of Illinois at Chicago, Chicago, IL, July 1992.