EVENT DETECTION FOR SUPPORTING ACTIVE CAPABILITY IN AN OODBMS: SEMANTICS, ARCHITECTURE AND IMPLEMENTATION

By

VIDHYA KRISHNAPRASAD

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1994

Dedicated to my
Parents

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for patiently dealing with all my decision changes and for giving me an opportunity to work on this challenging project. Under his expert advice and guidance I have been able to see beyond the present requirements of the system and develop it as an easily extensible one.

I am extremely grateful to Dr. Stanley Su and Dr. Herman Lam for agreeing to serve on my committee and for their comments to improve the manuscript.

I would like to thank Sharon Grant for maintaining a well administered research environment with her indefatigable spirit and commitment to work.

I am grateful to Zia and Eman for their invaluable help and fruitful discussions during the design and implementation of this work. I will also take this opportunity to thank all the graduate students at this center for their help and friendship.

Last, but not the least, I thank my parents and brothers for their love. Without their encouragement and endurance, this work would not have been possible. Also, I will always be grateful to my aunt Shubha and my brother Tejas for the support and love they gave me to make me feel at home. And I cannot express in words the gratitude I feel for the love of my life, my husband Kumar, who has been patiently waiting for me to complete my thesis and join him.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis
Presented to the Graduate School of the University of Florida
in Partial Fulfillment of the Requirements for the
Degree of Master of Science

EVENT DETECTION FOR SUPPORTING ACTIVE CAPABILITY IN AN OODBMS:
SEMANTICS, ARCHITECTURE AND IMPLEMENTATION

By

Vidhya Krishnaprasad

April 1994

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Sciences

During the last decade, database management systems (DBMSs) have evolved considerably to meet the requirements of emerging applications. One of the requirements is to represent real-world situations as part of the database, and monitor and react to them automatically without user or application intervention. Making a database system active entails not only developing an expressive event specification language with well-defined semantics and algorithms for the detection of composite events, but also an architecture for an event detector along with its implementation. This thesis extends earlier work on event specification language and provides the semantics of composite events over a global event-history (or a global event-log). Parameter contexts are refined for meaningfully restricting the detection of composite events. In addition, algorithms for the detection of composite events in various parameter contexts are presented. We describe the implementation of the composite event detector and rule execution in the context of an object-oriented active DBMS. We propose extensions to the Open OODB Toolkit (from Texas Instruments, Dallas) for incorporating ECA rules and discuss the advantages of the proposed architecture.

The functionality supported by our architecture is compared with other existing approaches and the possible extensions to our system are also discussed.

CHAPTER 1
INTRODUCTION

During the last decade, database management systems (DBMSs) have evolved considerably to meet the diverging requirements of application domains. Most new developments in database technology aim at representing real-world situations as part of the database and monitoring and reacting to them automatically without user or application intervention. Though triggers in DBMSs, ON conditions in programming languages, and signals in operating systems have been used effectively for condition monitoring, they are not at a level of abstraction appropriate for modeling non-traditional applications and cannot be seamlessly incorporated into traditional *passive* DBMSs [Cha91]. Traditional DBMSs are referred to as passive since any situation to be monitored has to be done explicitly by the user or application by executing queries or transactions. For example, in a hospital environment if the Electro Cardiogram (ECG) readings are recorded in a database for an intensive care unit patient, it is the responsibility of the doctor or nurse to check for the change of values over a period to determine any state of emergency. An *active* DBMS can continuously monitor situations to initiate appropriate actions in response to database updates, occurrence of particular states or transition of states automatically, possibly subject to timing constraints. In the previous example, the DBMS will alert the doctor when it detects any state of emergency based on the set of rules triggered as a result of the updates.

Situation monitoring can be done by defining ECA rules on events of interest. ECA rules, in the context of an active DBMS, consist primarily of three components: an event, a condition and an action. An event is an indicator of a happening which can

be either simple or complex. In database applications, they are mostly state changes that are produced by database operations (e.g., method invocations). We can also have temporal and explicit events, which are externally detected and signalled to the DBMS by the system or the user. The condition can be a simple or complex query based on the existing database states and set of data objects, transitions between states of objects or even trends and historical data. Actions specify the operations to be performed when an event has occurred and the condition evaluates to true. Once rules are specified declaratively, it is the responsibility of the DBMS to monitor the situation and trigger the rules when the condition is satisfied. Thus active DBMS provides both modularity and timely response and prevents hard-wiring code into the application. For our example, the event might be monitoring of the pulse rate over a time interval, the condition might be a drop in the rate (say by 50%), the action might be to alert the doctors and nurses by setting off an alarm.

The process of augmenting an exiting passive DBMS to have active capability involves event/rule specification, rule management and execution. The environment/model into which ECA rules are incorporated has a bearing on some of the above. As described by Anwar et al. [Anw93], event detection is considerably complex for an object-oriented environment and furthermore, compile time and runtime issues need to be addressed. Parameter computation and its usage is also complicated as there is no single data structure such as a relation into which parameters can be stored and passed. Optimization of condition and action components (if they are not non-procedural) as well as scope of shared and program objects are also different for the object-oriented model.

This thesis attempts to address some of the design and implementation aspects of making a database system active. This thesis extends our earlier work [Mis91, Cha94a] on event specification language and proposes an architecture for composite

event detection. It also covers the implementation for integrating composite events and rules with an existing passive DBMS. This thesis is organized as follows. Section 2 discusses the semantics of composite events and extensions to our earlier work. Section 3 introduces Open OODB (the passive DBMS on which active capability is incorporated), its architecture and extensions. It also details the ECA rule processing requirements and proposes an architecture for rule execution. Section 4 describes the implementation details. Section 5 gives an overview of related efforts to incorporate ECA rules into a passive DBMS and presents a comparison between our approach and related approaches. Section 6 presents conclusions and future work.

CHAPTER 2
SEMANTICS OF COMPOSITE EVENTS

The design of any active DBMS involves a method for specifying events and composite event expressions with associated semantics. The event specification language used in this thesis is an extension of our earlier work Snoop [Mis91]. A detailed discussion of how we have refined the semantics of Snoop event expressions with respect to various parameter contexts is presented in this section.

2.1 Summary of Snoop

We start with a brief description of the primtive events and event operators proposed in Snoop [Mis91] for the object oriented environment. Here, we assume an equi-distant discrete time domain having "0" as the origin and each time point represented by a non-negative integer. Chakravarthy and Mishra [Cha94a] distinguish between an event, an event expression, and an event modifier. Briefly, an event expression defines an interval on the time line. The notion of an event expression is needed to model operations that take a finite amount of time for their execution. For example, a method, a function, or a transaction can be viewed as an event expression. The interest in event expressions comes from the fact that one needs to choose a point on the time line, within the closed interval defined by an event expression, to denote an occurrence of that event. In other words, an event expression needs to be mapped to a point that can be declared as an event occurrence corresponding to that event expression. Event modifiers were introduced in Snoop [Cha94b] to transform an event expression to one or more events, each of which corresponds to a point of

interest within the closed interval defined by that event expression. For example, *begin-of* and *end-of* are two event modifiers that transform an arbitrary interval on the time line into corresponding event occurrences.

An *event* is an instantaneous, atomic (happens completely or not at all) occurrence. In database applications, the interest in events comes mostly from the state changes that are produced by database operations. That is, state changes are concomitant with operation execution and hence event occurrences corresponding to these operations are of interest. State changes are effected by using parameters associated with the operations. Hence, events are associated with operations (i.e., event expressions) of interest and operation's parameters are viewed as the parameters of the event associated with that operation. For simplicity, we assume that two occurrences of the same event are not simultaneous.[1] Moreover, we assume that no two event types occur simultaneously. Furthermore, an event may 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.

2.1.1 Primitive Events

Events are classified into: i) primitive events – events that are pre-defined in the system (using primitive event expressions and event modifiers). A mechanism for the detection is assumed to be available [Anw93] and ii) composite events – events that are formed by applying a set of operators to primitive and composite events

[1]This may not be true in multiprocessor and distributed environments. Furthermore, we do not differentiate between the time-of-occurrence (t_occ) and the time of detection (t_det).

constructed so far. Primitive events are further classified into database, temporal, and explicit events.

Database events correspond to database operations, such as retrieve, insert, update and delete (in the relational model) and methods (in the object-oriented model). Temporal events are either *absolute* or *relative*. An absolute temporal event is specified with an absolute value of time (and is represented as: $< timestring >$). For example, 2 p.m. on March 15th, 1994 is specified as $< (14:00:00)03/15/94 >$, using the format $< (hh/mm/ss)mm/dd/yy >$. In the specification of an absolute temporal event, a field in the time string may contain a wild card notation, which is denoted by '*' which matches *any* valid value for that field. This is especially useful in the specification of events that match many points on the time line. For example, in a banking application, to print the local branch report at 5 p.m. each day, one can specify an event using the wild card notation as follows: $< (17:00:00) * / * /* >$. In addition, a wild card can be used as a method for increasing the granularity of the time line by pre-specified amounts (e.g., seconds, minutes, days). A relative 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 $event + [timestring]$. In the representation of an offset, an empty field in the time string is substituted with the minimum value for that field. Observe that a relative event subsumes an 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.

2.1.2 Composite Events

Primitive events form the basic building blocks for developing an expressive and useful event specification language. In the absence 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.[2] We define a composite event as an event obtained by the application of an event modifier to a composite event expression. By default, the *end-of* event modifier is assumed.

Composite event expression is defined recursively, as an event expression formed by using a set of primitive event expressions, event operators, and composite event expressions constructed up to that point. Below, we describe each of these operators and their semantics. We will use E (upper case alphabets) to represent an event expression as well as an event type and e (lower case alphabets) to represent an instance of the event E. We use superscripts to denote the relative time of occurrence with respect to events of the same type. Subscripts denote the event types [Cha93].

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

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

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.

[2] In fact, in production rule systems (e.g., OPS5 [For82, For87]), 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.

The Snoop event operators[3] 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,

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

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,

$$\begin{aligned}(E_1 \triangle E_2)(t) \quad &= \quad (E_1(t^1) \wedge E_2(t)) \vee ((E_1(t) \wedge E_2(t^1)) \\ &\quad \text{and } t^1 \leq t\end{aligned}$$

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

$$\begin{aligned}(E_1 \nabla E_2)(t) &= (E_2 \nabla E_1)(t) \\ (E_1 \triangle (E_2 \triangle E_3))(t) &= ((E_1 \triangle E_2) \triangle E_3)(t) \\ &= (E_1 \triangle (E_2 \triangle E_3))(t) \\ &= (E_1 \triangle E_2 \triangle E_3)(t)\end{aligned}$$

3. **ANY:** The conjunction event, denoted by Any(m, E_1, E_2, ... E_n) where m <= n, occurs when m events out of the n *distinct* events specified occur, ignoring the relative order of their occurrence. Formally,

$$\begin{aligned}ANY(m, E_1, E_2, \cdots, E_n)(t) \quad &= \quad (E_i(t^1) \wedge E_j(t^2) \wedge \cdots \wedge E_k(t^m)) \\ &\quad \text{and } t^1 \leq t^2 \leq \cdots \leq t^m \text{ and } t^m = t \\ &\quad \text{for some distinct } i, j, \cdots, k, \text{ each} \leq n\end{aligned}$$

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

$$\begin{aligned}\text{For example, } ANY(3, E_1, E_2, \cdots, E_n)(t) &= (E_i(t^1) \wedge E_j(t^2) \wedge E_k(t^m)) \\ &\quad \text{and } (t^1 \leq t) \text{ and } (t^2 \leq t) \\ &\quad \text{and } t^m = t \text{ and } (i \neq j \neq k) \\ &\quad \text{and } (i \leq n) \text{ and } (j \leq n) \\ &\quad \text{and } (k \leq n)\end{aligned}$$

To specify m distinct occurrences of an event E_1 :

$$\begin{aligned}ANY(m, E_1^*)(t) &= (E_1(t^1) \wedge E_1(t^2) \wedge \cdots \wedge E_k(t^m)) \\ &\quad \text{and } t^1 < t^2 < \cdots < t^m \text{ and } t^m = t\end{aligned}$$

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,

$$\begin{aligned}(E_1; E_2)(t) &= (E_2(t) \wedge E_1(t^1)) \\ &\quad \text{and } t^1 < t\end{aligned}$$

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 occurrence of an aperiodic event in the half-open interval formed by E_1 and E_2[4].

[4]The interval can either be (t_occ(E_1), t_occ(E_2)] or [t_occ(E_1), t_occ(E_2)).

There are two variations of this event specification. The non-cumulative variant of an aperiodic event is expressed as 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 during the half-open interval defined by E_1 and E_3. **A** can occur zero or more times (zero times either when E_2 does not occur in the interval or when no interval exists for the definitions of E_1 and E_3). Formally,

$$\begin{aligned} A(E_1, E_2, E_3)(t) &= (E_1(t^1) \wedge \sim E_3(t^2) \wedge E_2(t)) \\ &\quad \text{and } t^1 < t^2 \leq t \text{ or} \\ &\quad t^1 \leq t^2 < t \end{aligned}$$

There are situations when a given event is signaled more than once during a given interval (e.g. within a transaction), but rather than detecting the event and firing the rule every time the event occurs, the rule has to be fired only once. To meet this requirement, we provide an operator $A^*(E_1, E_2, E_3)$ that occurs only once when E_3 occurs and accumulates the occurrences of E_2 in the half-open 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 computing aggregates. As an example, highest or lowest stock price can be computed over an interval using this operator. Formally,

$$A^*(E_1, E_2, E_3)(t) = (E_1(t^1) \wedge E_3(t)) \text{ and } t^1 < t$$

In this formulation E_2 (there can be 0 or more occurrences of it) is not included because we are concerned with occurrence of the composite event A^* which coincides with the occurrence of E_3 and is not constrained by the occurrence of E_2. However, the parameters of A^* will contain the parameters of E_2.

6. **Periodic Event Operators (P, P*):**

We define a periodic event as an event E that repeats itself within a constant and finite amount of time. Only a time specification is meaningful for E. The notation used for expressing a periodic event is P(E_1, [t], E_3) where E_1 and E_3 are events and t is the time specification. P occurs for every t in the half-open interval (E_1, E_3]. t is assumed to be positive. It is important to note that t is a *constant* and preferably not contain wild card specification in *all* fields because this will result in continuous (i.e., for each point on the time line) occurrences of P. Formally,

$$\begin{aligned} P(E_1, [TI], E_3)(t) \quad &= \quad (E_1(t^1) \wedge \sim E_3(t^2)) \\ &\quad \text{and } t^1 < t^2 \text{ and } t^1 + i * TI = t \text{ for some } 0 < i < t \\ &\quad \text{and } t^2 \leq t \end{aligned}$$

where TI is a time specification.

Note that the event of interest in P is the middle event which is a time specification. To make the event more practical and meaningful for real-life applications, it may be useful to allow a parameter along with the frequency specification. To accommodate this, we define a cumulative variation of P (denoted P*) which includes a parameter for each occurrence of the periodic event. In the absence of this parameter, the cumulative operator just accumulates time of occurrences of the periodic event as the parameter object. Formally,

$$\begin{aligned} P^*(E_1, [TI]Arg, E_3)(t) \quad &= \quad (E_1(t^1) \wedge E_3(t)) \\ &\quad \text{and } t^1 \leq t \end{aligned}$$

Though TI is not mentioned in this formulation, the parameters specified are collected for each occurrence of [TI] as part of P*.

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.

   Note that this operator is different from that of !E (a unary operator in Ode [Geh92b]) which detects the occurrence of any event other than E.

$$\neg(E_2)[E_1, E_3](t) = (E_1(t^1) \wedge \sim E_2(t^2) \wedge E_3(t)) \text{ and } t^1 \leq t^2 \leq t$$

We believe that the above set of event operators define an event specification language that meets the requirements of a large class of applications. Periodic and aperiodic operators were introduced to meet the requirements of process control, network management, and CIM applications.

2.2 Histories and Event Logs

So far, we have defined the semantics of event operators over the time line in which only the time of event (primitive or composite) occurrences were recorded. However, detection of a composite event entails detecting not only the time at which the composite event occurs, but also the constituent event occurrences and associated parameters 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. A constituent event of an event are its sub-events. At some level, all constituent events are primitive events.

Recall that event occurrences are denoted by e_j^i where j denotes the event type and i denotes the relative time of occurrence with respect to events of the same type. Composite events are represented as a *set of constituent event occurrences* within which the *order* of event occurrences is *preserved for non-repeating primitive events.* That is, it is possible for the same event occurrence to be used more than once as a

participating event. For these repeating use of the same event, only one occurrence will conform to the order of event occurrences. Furthermore, the last event in the set is one whose occurrence makes the composite event occur. The time of occurrence of a composite event is the time of occurrence of the last primitive event in the set.

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 set in the log.

$$H = \{\{e_j^i\} \mid \text{for all } E_j, \text{the primitive event } e_j \text{ has occurred at instance i relative to events } E_j\}$$

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

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

Composite Event-History/Event-Log of a composite event $E_{composite}$ that has n constituent events $E_1, \cdots, E_n$ is a mapping from the global event-history H to a subset of $E_1[H] \uplus \ldots \uplus E_n[H]$ 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 set union operator.

For example, $E_{composite}[H] \uplus E_2[H]$, given the event histories,

$$E_{composite}[H] = \{\{e_1^1, e_3^4\}, \{e_1^2\}\} \quad \text{and} \quad E_2[H] = \{\{e_2^1\}, \{e_2^2\}\}$$

is given by

$$E_{composite}[H] \uplus E_2[H] = \{ \{e_1^1, e_3^4, e_2^1\}, \{e_1^1, e_3^4, e_2^2\}, \{e_1^2, e_2^1\}, \{e_1^2, e_2^2\}\}$$

Event Collection is a collection of all primitive/composite events occurrences of a particular type within a specified time interval. It is denoted by the function $\wp$.

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

Note that if E is a composite event E[H] is computed according to the definition given above.

Given a global event-history, the event-history for an arbitrary composite event formulated using the operators defined in section 2.1.2 can be easily 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.

2.2.1 The Unrestricted Context

1.

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

2.

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

3.

$$\begin{aligned}ANY(m, E_1, E_2, \cdots, E_n)[H] = \{\{e^i, e^j, \cdots, e^k\} \mid & \\ & t\_occ(e^i) < t\_occ(e^j) < \cdots < t\_occ(e^k) \text{ and} \\ & \mid \{e^i, e^j, \cdots, e^k\} \mid = m \leq n \text{ and}\end{aligned}$$

$$\{e^i, e^j, \cdots, e^k\} \in \mathcal{P}(E_1[H] \ \uplus E_2[H] \ \uplus \cdots \ \uplus E_n[H])\}$$

where $\mathcal{P}$ is the power set.

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

4.

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

5.

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

6.

$$\begin{aligned} P(E_1, [t], E_3)[H] \ = \{\{e^i, t^j\} \mid \ & t\_occ(e^i) < t^j) \text{ and} \\ & \forall \{e^i, e^k\} \in E_1[H] \ \uplus E_3[H] \\ & \text{all } t^j = i + jt \text{ and } t^j < k \text{ and } j > 0\} \end{aligned}$$

7.

$$\begin{aligned} \neg(E_2)[E_1, E_3][H] \ = \{\{e^i, e^k\} \mid \ & \{e^i, e^k\} \in E_1[H] \ \uplus E_3[H] \text{ and} \\ & \wp(E_2, t\_occ(e^i), t\_occ(e^k)) = \phi\} \end{aligned}$$

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.

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

9.

$$\begin{aligned} P^*(E_1, [t], E_3)[H] \;=\; & \{\{e^i, \wp(t, t\_occ(e^i), t\_occ(e^k)), e^k\} \mid \\ & \{e^i\} \in E_1[H], t\_occ(e^i) < t\_occ(e^k) \\ & \wp \text{ returns timepoints and } \{e^k\} \in E_3[H] \\ & \forall \{e^i, e^k\} \in E_1[H] \;\; \uplus E_3[H] \\ & \text{all } t = i + jt' \text{ and } t < k \text{ and } j > 0\} \end{aligned}$$

Below, we illustrate the computation of the composite event X[5] on the global history according to the above definitions of operators for the *unrestricted context.* 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

[5]The event X is drawn from the stock market applications. The interpretation of X is as follows: E1: Opening of stock market, E2: Change in Dow Jones average, E3: Change in the price of IBM stock, and E4: Change in a commodity which depends on IBM stock.

generated for an event.

$$H = \{\{e_1^1\},\{e_1^2\},\{e_2^1\},\{e_3^1\},\{e_2^2\},\{e_4^1\},\{e_3^2\},\{e_4^2\}\}$$

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

$$\begin{aligned} X &= ((E_1 \triangle E_2); E_3; (E_2 \triangle E_4)) \\ X[H] &= ((E_1 \triangle E_2); E_3; (E_2 \triangle E_4))[H] \\ &= ((E_1 \triangle E_2)[H]; E_3[H]; (E_2 \triangle E_4)[H])[H] \\ &= (((E_1 \triangle E_2)[H]; E_3[H])[H]; (E_2 \triangle E_4)[H])[H] \end{aligned}$$

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

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

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.

2.3 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 Snoop [Cha94a] 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.

Parameter contexts essentially delimit the events detected, parameters computed, and accommodate a wide range of application requirements. 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 type. 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 multiple times over an interval.

The occurrence of any composite event (e.g., Z) is marked by the occurrence of a constituent event that makes the composite event occur (using the *end-of* event expression semantics). Recursive application of this definition will yield a primitive event that marks the end of a given composite event. This primitive event is termed the *terminator* of the composite event Z. Several primitive events can act as terminators, but there is at least one terminator event for a given composite event. Analogously, there is always a primitive event that initiates the occurrence of a composite event. This primitive event is termed the *initiator* of the composite event. There is at least one initiator for a composite event but there could be more than one. For a primitive event the terminator is the same as the initiator. For any one occurrence of a composite event there is only a single initiator and terminator.

A sequence of primitive event occurrences (over a period of time) makes a composite event occur. Hence, the composite event detector needs to record the occurrence of each event and save its parameters so that they can be used to compute the parameter set of the composite event. We adopt the notation Z(E1, E2, $\cdots$, En) to represent an event expression Z, where Ei[6], i = 1..n are its constituent primitive events. Consider the following event expressions:

A = (E1 $\triangle$ E2) ; E3[7] B = $E1 \nabla E2 \nabla E3$ and C = E1 ; Any(2,E2, E3)

where E1, E2, and E3 are primitive events. Event A is detected when at least one instance of all three events has occurred with E3 being the last occurrence. Event B is signaled each time an instance of any of the three events E1, E2 or E3 occurs. Parameters of event A (as well as C) include parameters of all the three events E1, E2 and E3 whereas the parameters of event B include only the parameters of one of its events. Both E1 and E2 can be initiators of A and E3 is the only terminator. For

[6] E_i and Ei have been used interchangeably from this point

[7] This is a constituent event of the composite event X used in the previous section.

C, E1 is the initiator and both E2 and E3 can be terminators. Figure 2.1 shows the occurrences of different instances[8] of event E1, E2 and E3 as well as the event graph for A.

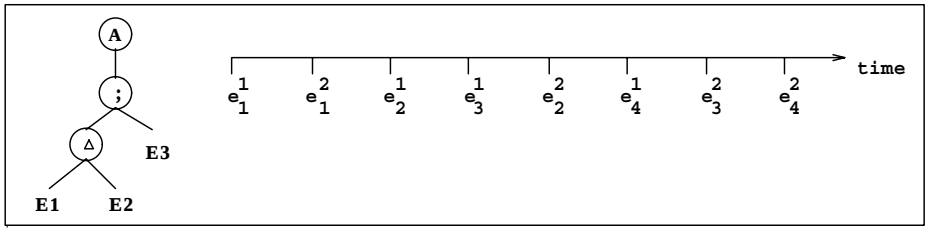

Figure 2.1. Global event history

2.3.1 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

[8]The same occurrence of event instances are used in the rest of the thesis.

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, and

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 to that occurrence of the 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 a 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 event X (example used in the previous section). We are not concerned with the primitive occurrences e_4^1 and e_4^2 as primitive event E_4 is not part of the event expression of A. Note that the semantics of a primitive event, is identical in all contexts.

- **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 X is detected, its parameters are computed by using the oldest initiator and the oldest terminator of E. However, the constituent events of an event X cannot occur in any other detection of the occurrence of E. For example, parameters of event A in the chronicle context will be computed by using event instances $\{e_1^1$, e_2^1, $e_3^1\}$. 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. The initiator and the terminator are discarded after an event is detected. This context is especially useful for tracking trends of interest on a sliding time point governed by the initiator event. In Figure 2.1, 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, all occurrences of an event type are accumulated as instances of that event until the event is detected. Whenever an event is detected, all the occurrences that are used for detecting that 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 first four event occurrences instances shown in Figure 2.1 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 X, the t_occ of the initiator is greater than the t_occ of the terminator of the immediately preceding occurrence of X. This notion of the use of non-overlapping intervals can be applied to any of the

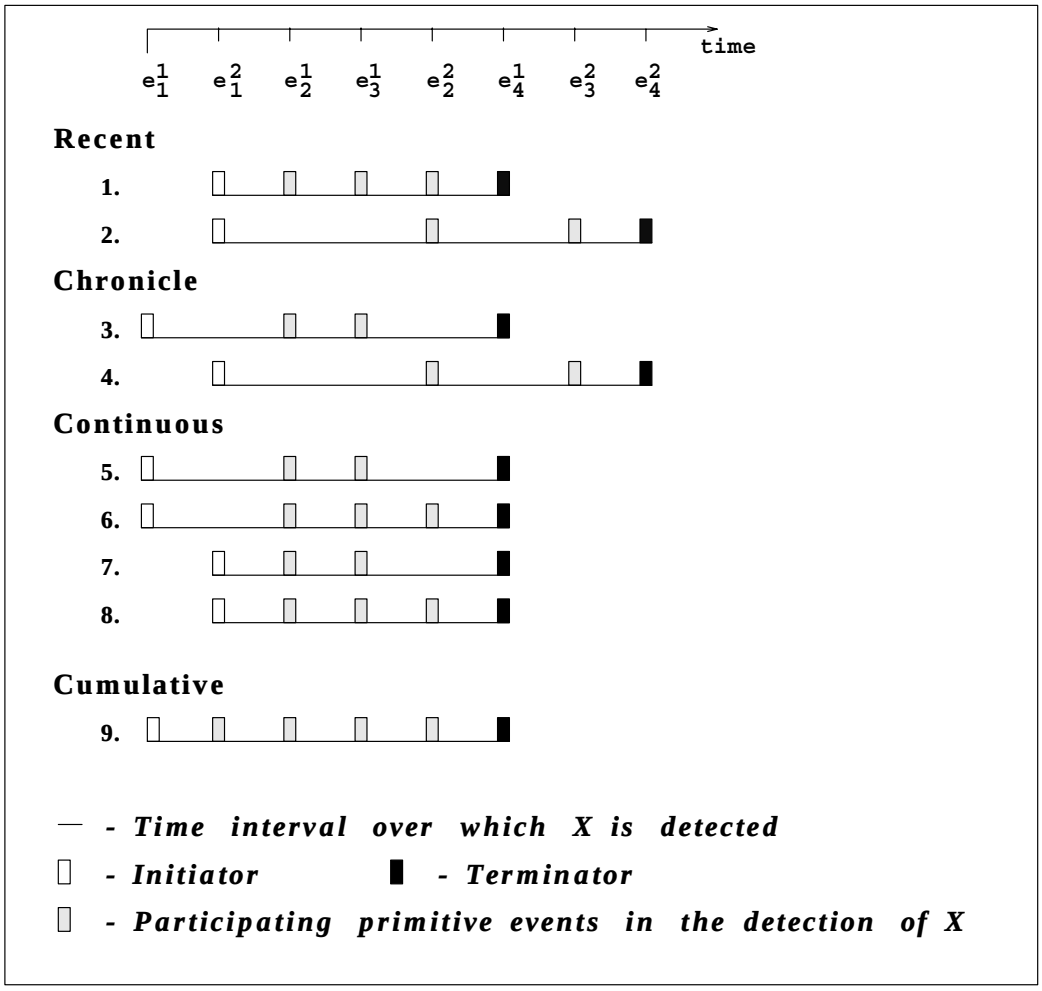

Figure 2.2. Illustration of Event detection in various contexts for the expression X = (E1 $\triangle$ E2 ; E3; E2 $\triangle$ E4)

contexts described in this section, including the unrestricted context. This can be easily seen from the Figure 2.2. 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 2.2, 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.

2.3.2 Illustration of Composite Event Detection

The approach taken for composite event detection in this thesis 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 thesis. 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 does not seem to produce the desired result. These approaches are discussed in detail in chapter 5.

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 the event graph 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. The appendix includes detailed illustration of each of the parameter contexts through an example and also the algorithm for each of the parameter context. From the example 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. 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 2.3 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. The event instances that will be deleted after this instant of time are expressed in bold letters.

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 E 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 E 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 E 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 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 expression 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,

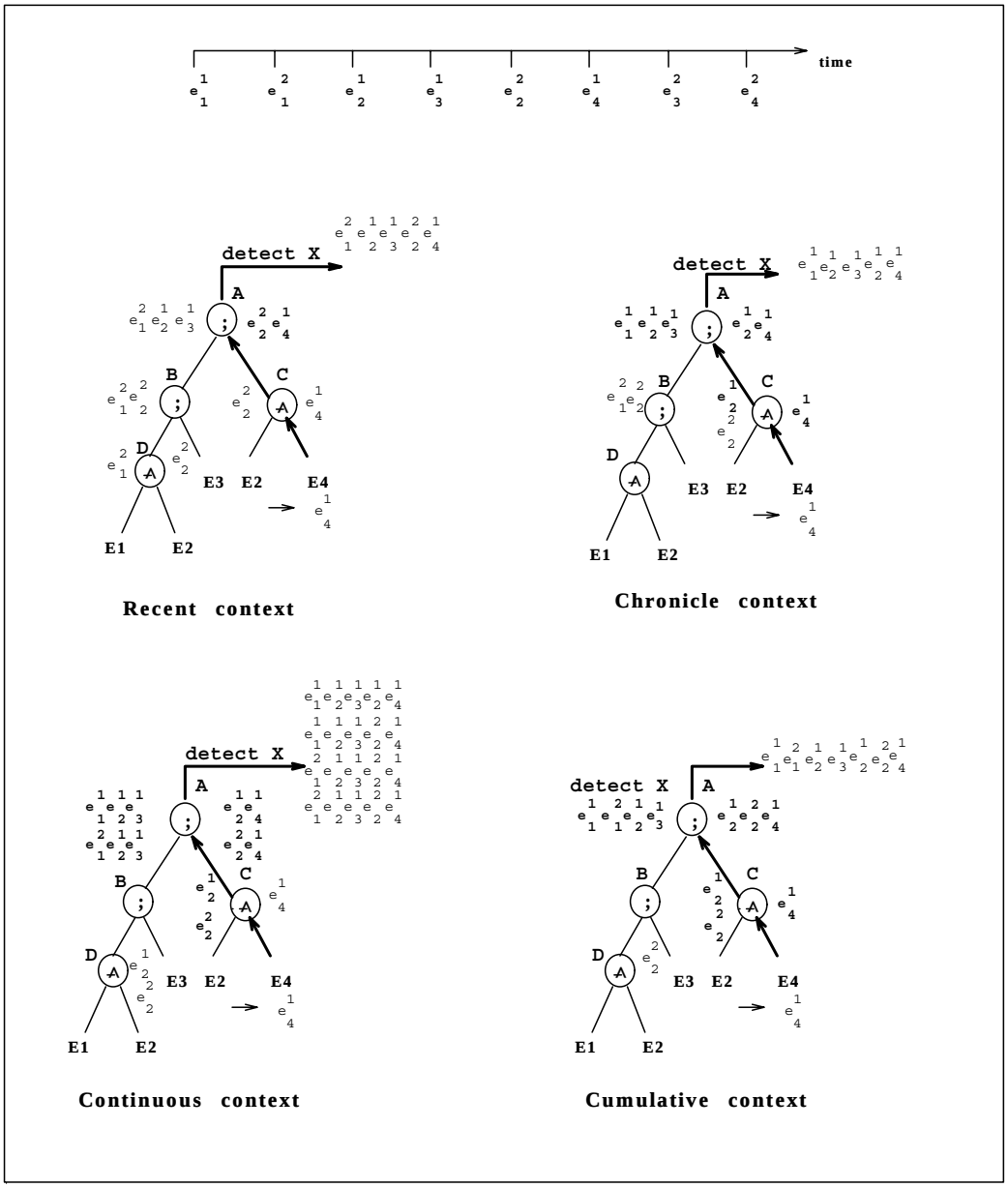

Figure 2.3. Event detection in various contexts

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 4 instances of the detection of event E with the same time of occurrence. Among the 4 contexts presented, 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.

2.4 Storage Requirements

Parameter contexts introduced in this section 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 continuous and chronicle both 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 a lot of 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.

2.5 Issues not addressed in Snoop

This thesis extends earlier work on Snoop [Mis91, Cha94a] in several significant ways. Earlier work was primarily concerned with the motivation for the event language, classification of events, need for event operators, a small set of event operators and parameter contexts. In this thesis, 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 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 cannot be deleted. To overcome this problem, we refine 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 using the initiator and terminator concepts.

Snoop gave an outline of all the contexts that are meaningful for various application domains. But the parameter contexts identified had to be detailed further. Also, the effect of the contexts on the various operators and whether the application of the context does make sense for certain operators like A, A^*, P, P^* had to be analysed. The parameter computation for these operators as constituent events had to be specified. Event expressions can be specified earlier and rules can be bound dynamically to event expressions at a later point in time. Meanwhile the event graph constructed for the event expression might have detected certain primitive/composite events. Hence there is a possibility that rules specified on these event expressions may be triggered by the constituent events whose t_occ is less than the initiation time of the rule. This led to the provision of options for triggering rules based on the point of time the rule was activated or the point of the time the composite event expression to which the rule subscribed was declared. Also, the parameters computed for any valid expression in Snoop was stipulated to be in the form of a relation so that all relational operators can be applied to it. Since we were looking at an object oriented environment this specification could not be adhered to. Moreover each method could have a variable number of parameters and values, so it is not possible to express them as a relation. Hence linked lists were chosen as a form of parameter representation.

The concept of Initiator and Terminator was introduced to define context for each operator. An analysis of storage requirements and comparison of contexts have been

presented. This will further help us to investigate whether having different contexts for subexpressions of an event expression is of interest for any application domain.

Snoop did not include the NOT operator as part of its event specification language. We have introduced the NOT operator to capture the non-occurence of primitive or composite event within a well-defined interval.

We have developed complete algorithms for detecting Snoop expressions in all the parameter contexts and presented them in the appendix.

CHAPTER 3
ARCHITECTURE

The previous chapter highlighted the semantics and contexts of composite event detection. This chapter details an architecture for incorporating rules/events into an existing passive DBMS. The details of the passive DBMS and its features are highlighted in the following section, followed by the requirements of a rule processing subsystem and our architecture.

3.1 Architecture of the Open OODB system

The Open OODB project [Wel92, TEX93] at Texas Instruments, Dallas, was an effort to build a high-performance, multi-user object oriented database management system (OODBMS) in which database functionality can be tailored by application developers for the diverse needs of demanding applications.

The system provides an incrementally improvable framework that can also serve as a common testbed for research by database, framework, environment, and system developers who want to experiment with different system architectures or components. The toolkit organization also facilitates importation of new components from smaller groups lacking resources to build an entire database system.

The Open OODB tries to describe the design space of OODB, build an architectural framework that enables configuring independently useful modules to form an Object Oriented Database Management System. Open OODB has extended the existing language (C++) in a seamless manner to incorporate persistence. The system architecture is divided into i) meta-architecture - consisting of a collection of kernel

modules and definitions providing the infrastructure for creating environments and boundaries, and regularizing interfaces among modules, and ii) an extensible collection of policy manager modules - providing the functionality for OODB. To allow for openness and modularity, the OODB Toolkit was architected as a collection of independent object services paralleling the Object Management Group (OMG) Object Services Architecture. The services can be combined to provide object-oriented database, relational database, repository, and distribution systems. Several modules are database independent and can be used in non-database applications. In essence, an extensible Open architecture is adopted for Open OODB. Since OODB is an Object-oriented front end, it uses Exodus storage manager as its underlying storage manager through an interface.

3.1.1 Features of Open OODB

Seamless Interfaces: Open OODB *seamlessly* adds functionality such as: persistence, resilience, concurrent transactions, and schema evolution to developers' existing programming environments. Open OODB does not require changes to either type (class) definitions or the way in which objects are manipulated. Rather, applications "declare" normal programming language objects to possess certain additional properties; such objects then transparently "behave properly" according to the declared extensions when manipulated in the normal fashion. For example, if an object is declared as persistent, the DBMS is responsible for moving it between the computational and long term memory as needed to ensure both its residency during computation and its preservation during program termination. This allows programmers to: i) stay within familiar programming paradigms, ii) stay within familiar programming languages, and iii) support legacy code and data. OODB extends existing languages (C++ and Common Lisp) rather than trying to invent a new "database language".

Sentry mechanism: The Open OODB computational model allows developers to define behavioral extensions of events, which is an application of an operation to a particular set of objects. In this model all objects accessible to a program exist in an "universe of objects". This universe is partitioned into "environments" by "environmental attributes". Environmental attributes include information about the address space where the object resides (e.g., persistent or transient, local or remote), replicas of object, lock status and transaction owning the lock, etc. These environments and boundaries of the environments identify where extensions may be required. For example, if we need an extension to allow objects to reside in a remote address space, we can define an environmental attribute named "address space" that defines the location of the object using the domain values which are the set of address spaces where the object could reside. To perform these extensions we must be able to interrupt or trap operations. Thus, the trapping mechanism combined with the protocol for permitting the entity performing the trapping to invoke an arbitrary extension is know as a "sentry". The primary function of sentries is to detect events which are interaction with objects, and to pass control to a policy manager which controls and performs the actual extension if it is determined that an event should be extended. The sentry manager is the used for specifying events to be extended, and is responsible for deploying sentries to detect extended events.

Extensibility: When an object is declared to Open OODB to have "extended" behavior, there are certain "invariants" associated with the extension that must be enforced. When an operation involving an extended object occurs, the sentry is called which as detailed above interrupts the operation and transfers control to a *policy manager* module responsible for ensuring that operations against extended objects "behave properly". Each semantic extension is implemented by a different

policy manager. Thus, there is a policy manager for persistence, another for index maintenance, etc. Policy managers can be added independently, and are inherited from a common root class to make them type compatible for invocation purposes. This strategy allows new extensions to be added, the semantics of a given extension to be changed, and implementation of a given policy to be changed or selected dynamically. It allows for hiding the semantic extension from applications to obtain seamlessness. Basic services used by policy managers are provided by a collections of "support modules".

Reusability: With an open system, researchers can focus on modules of interest without having to build complete systems. This reduces duplication by encouraging the reuse of system components, and increases the quality and depth of components of the system by allowing developers to focus on smaller portions of the system. To achieve this, Open OODB uses a generic framework for extensibility that allows reuse of components developed by different research groups and organizations. It should be noted that OODBs by their very nature facilitate code reuse, since stored objects contain code as well as state.

Persistence: Persistence is the ability of objects to exist beyond the lifetime of the program that created them. The Persistence Policy Manager in Open OODB provides applications with an interface through which they can create, access and manipulate persistent objects. EXODUS is used as the persistent store for objects. The interaction with EXODUS in transferring and saving objects is built into the Persistence Policy Manager and hence is transparent to the user.

Application Programming Interfaces: Open OODB provides seamless extensions to both C++ and Common Lisp. The features of each of these APIs include:

- full coverage of C++ type system and Common Lisp type system (including CLOS).
- persistence.
- recoverable, concurrency controlled transactions.
- remote access to data via a client/server model.
- SQL-like object queries in C++ API.

The various features outlined above encouraged the use of Open OODB for our project. Also the architecture of Open OODB is data model independent. Moreover, the availability of the source code for the Release 0.2 (Alpha) helped us to modify the Open OODB system to suit our requirements. The primary class OODB has been extended to have reactive capability. Also the availability of sentry mechanism helped us build wrapper functions wherever necessary. The persistent feature will be useful when the current system is extended to detect global events.

3.2 Rule Processing Requirements

Before we detail our architecture we need to identify the requirements for ECA rule management to determine the aspects that the architecture should cover.

Briefly, ECA rule management involves event detection, rule execution and rule maintenance in a manner that is consistent with the transaction concept for databases. Event detection entails not only the detection of primitive events but also of composite events in an efficient manner. The semantics of rule execution in the context of databases is based on the work done by Chakravarthy [Cha89] and Widom et al. [Wid90]. **Rule scheduling** involves ordering of rules for execution when several rules are triggered at the same time. Either a conflict resolution strategy (using the

user specified priority or precede/follows information) can be used to totally order the rules or traditional serializability theory can be applied to execute rules concurrently or a combination of both. For example, Starburst [Wid90] uses the first approach whereas HiPAC [Hsu88] uses the second approach using an extended nested transaction model.

Rule execution points have been identified in HiPAC [Hsu88] as coupling modes. Three coupling modes were introduced to support various application needs. Their semantics with respect to triggering transactions is defined as follows: in the *immediate coupling* mode a rule is executed at the point where the event occurs, in the *deferred coupling* mode a rule is executed at the end of the transaction prior to its commit, and in *detached coupling* mode a rule is executed as an independent transaction. A causally dependent variation of the detached mode was introduced in which the independent rule transaction is not committed unless the triggering transaction commits. These modes can be specified on a finer granularity (i.e., independently between event and condition as well as between condition and action).

Nested rule (or even cyclic) execution occurs when a rule action signals events triggering additional rules to an arbitrary level of nesting. Again scheduling strategies (depth-first, breadth-first etc.) for these rules need to be outlined.

Rule management involves keeping track of activated and deactivated rules. Reactivating rules involves deciding whether the rule will get triggered by events that occurred prior to its activation. Based on the given priority, one can group a set of rules (e.g., integrity rules) and assign execution semantics automatically. For example, integrity rules need to be triggered in the deferred mode as the database state can be inconsistent within a transaction. Also, if rules are treated as shared objects (like any other shared data), then modification of rules need to be supported. This entails subjecting rules to the same concurrency control mechanism used for

any other shared data. Otherwise, rules have to be treated as meta-data whose manipulation is deemed different from shared data.

The environment/model into which ECA rules are incorporated has a bearing on some of the above. As described by Anwar et al. [Anw93], event detection is considerably complex for an object-oriented environment and furthermore, compile time and runtime issues need to be addressed. Parameter computation and its usage is also complicated as there is no single data structure such as a relation into which parameters can be stored and passed. Optimization of condition and action components (if they are not non-procedural) as well as scope of shared and program objects are also different for the object-oriented model.

In addition to the above requirements of ECA rule processing, we have to analyze the requirements of a rule processing subsystem for an object-oriented active DBMS. We need to support:

Inter-application rules: 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,

Parameter computation: When a composite event is *detected*, the parameters need to be collected and recorded by the event detector. Furthermore, these parameters need to be interpreted by the rule condition and action.

Multiple rules: An event can trigger several rules. Hence, it is necessary to support a rule execution model that supports concurrent as well as prioritized rule execution, and finally

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).

The above requirements as well as the data model under consideration affect the design of both the rule processing subsystem and the event detector. Below, the issues involved in each of the above requirements are analyzed to derive the architecture presented. Using our event specification language, we can readily model the deferred coupling mode in terms of immediate coupling mode by using the A^* operator and *begin* and *pre-commit* transaction events. 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. This formulation handles the net effect variant of deferred rule execution. Hence, we need to implement only the immediate and detached coupling modes. The detached independent coupling mode requires that a new transaction be started for rule execution. This has severe ramifications in the object-oriented model where the rule's condition and action could be arbitrary functions requiring the declaration of all classes. Unlike the relational model, creating an independent transaction for a rule in an object-oriented case, may be limited by the host environment (e.g., objective C, C++, Common Lisp, SmallTalk). The causally dependent coupling mode can be modeled by using events that span applications (i.e., by using the inter-application rules mentioned above), assuming that it is possible to create a top-level transaction corresponding to that rule. Each transaction can signal a pre-commit and (possibly) an abort event which can be used by the global event detector to enforce abort dependencies between two top-level transactions.

Supporting inter-application rules requires not only the detection of global events spanning several applications, but also dealing with address space issues. In the relational model, it is easier to handle this as the data dictionary has the type information of all objects and furthermore attributes values are atomic. In the object-oriented model, interoperability across applications is extremely complicated on account of the component objects, pointers, and virtual functions. These issues are currently

being addressed by OMG and Corba [Vin93]. Given the limitations of this model, we feel that it is possible to pass only the event name, persistent oid, and atomic values (pass by value) across address spaces and the interpretation of these parameters must be left to the application executing the rule. Of course, shared database objects can be accessed as part of the rule evaluation. Parameter computation for composite objects 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 are including the identification of the object (i.e., oid) as one of the event parameters and simple types by value. 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 spans a time interval. Complete support for parameters of composite events may require versioning of objects and related concurrency control and recovery techniques.

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 can be accomplished relatively easily by separating the composite event detection from the application as detailed in the architecture described below. Finally, support for both online and batch/after-the-fact detection of composite events is also accomplished by separating the composite event detector from the application and detection of primitive events.

3.3 Sentinel Architecture

The Sentinel architecture proposed in this section extends the *passive* Open OODB system [TEX93]. The Open OODB Toolkit uses Exodus as the storage manager and supports persistence of C++ objects. Concurrency control and recovery are provided by the Exodus storage manager. A full C++ pre-processor is used for transforming the user class definitions as well as the application code. Extensions incorporated for making the Open OODB active, are:

- Specification of ECA rules either as a part of the class definition or as part of an application; this is pre-processed (by using an enhanced C++ pre-processor) into appropriate code for event detection and rule execution,
- Detection of primitive events by using the sentry mechanism of the Open OODB. Sentry mechanism provides a wrapper method that permits us to invoke notification of an event to the composite event detector,
- A composite event detector for detecting composite events in various contexts [Cha93]. There is a composite event detector for each Open OODB application or client (each application of Open OODB is a client to the Exodus server),

Figure 3.1 shows how the class lattice of the Open OODB has been extended. The classes outside the dotted box have been introduced to make Open OODB active.

In order to satisfy the above requirements in an object-oriented framework, we use the architecture shown in Figure 3.2. The architecture supports the following features. i) Detection of primitive events, ii) Detection of composite events, iii) Parameter computation of composite events, and iv) Clean separation of composite event detection with application execution.

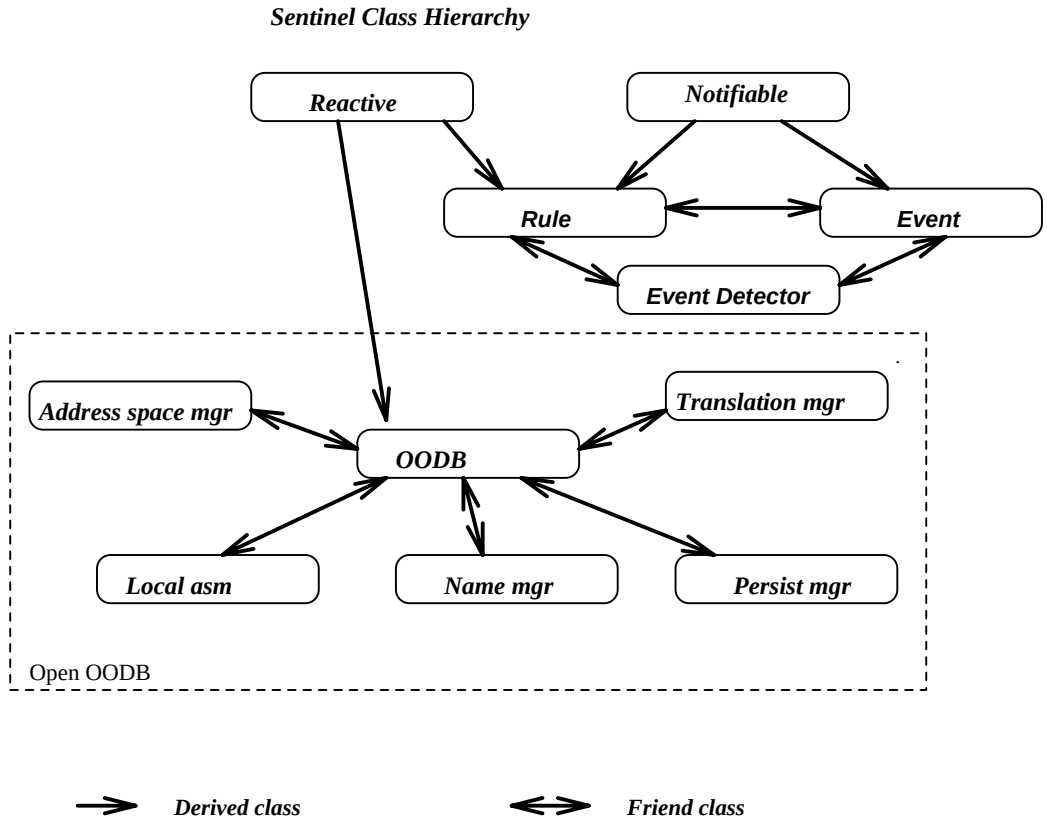

Figure 3.1. Class Lattice of Sentinel

Our primitive event detection is based on the design proposed by Anwar et al. [Anw93]. Both primitive and composite events can be 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. We have modified the Open OODB to support the detection of primitive events. 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 and ii) introduce additional event operators without having to modify the detection of primitive events.

Each application has a local event detector to which all primitive events are signaled. In addition each application will have a thread that handles the execution

of rules whose events span applications (a global event-handler thread). Our implementation uses threads (light weight processes), instead of processes, for separating composite event detection from application as: i) threads communicate via shared memory rather than a file system, thus allowing applications to share the same address space, ii) the overhead involved in creating threads and inter-task communication is low, and iii) it is easy to control the scheduling and communication of threads by assigning priorities. When a primitive event occurs it is sent to the local event detector and the application waits for the signaling of rules that are detected in the *immediate* mode. The global event detector communicates with the local event detectors for receiving events detected locally and with the application's global event handler for signaling the detection of global events for executing tasks based on global events. Again there is a clean separation between the events detected by the local event detector and the global event detector. Finally, as the local event detector and the application share the same address space and our event detection uses an event graph similar to operator trees, it is possible to combine rule evaluation with event detection (when the coupling mode permits and rules are non-procedural) and optimize the entire tree as a whole.

For multiple rule execution, a number of sub-transactions are spawned as a part of the application process. This is further elaborated in Badani's thesis [Bad93]. The order of rule execution is controlled by assigning appropriate priorities to each thread. For detached execution of rules, we are assuming that a separate application can be started with the rule as the body of a top-level transaction. With this assumption, for detached mode with causal dependency, an inter-application event is created to be detected by the global event detector. This is used to enforce the abort dependency between the two top-level transactions. These are highlighted in Figure 3.2 by indicating control and data flow. Detached mode is not yet implemented.

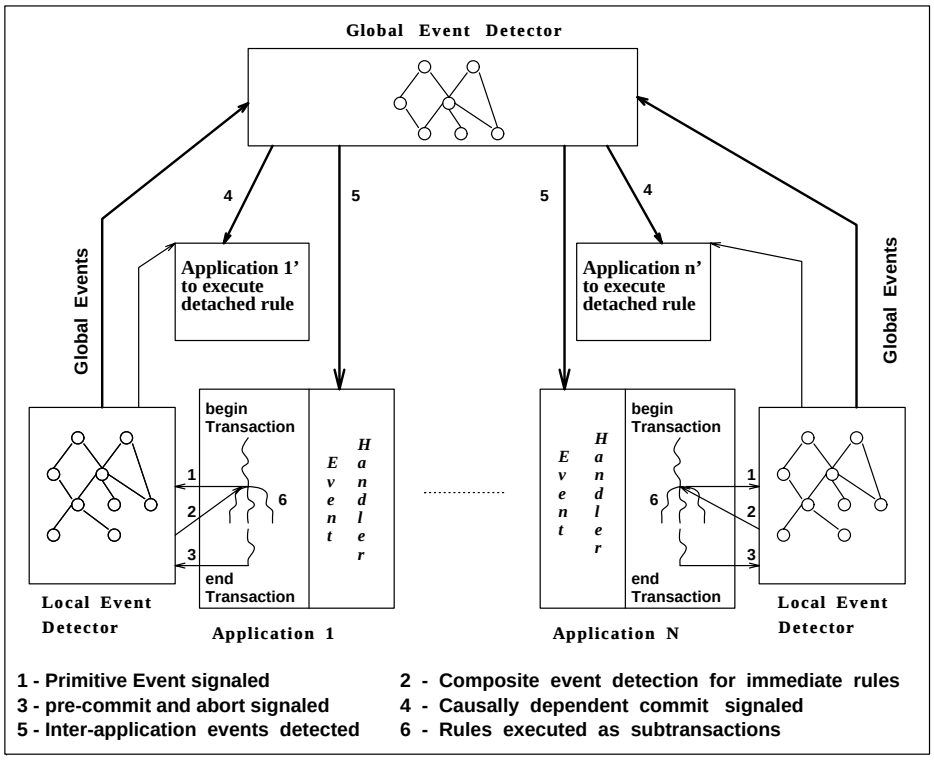

Figure 3.2. Sentinel Architecture

As the composite event detector can receive event occurrences either as they happen or from a file, it can be used for after-the-fact analysis of events (e.g., telecommunications or stock market applications) as well as a part of an active database.

CHAPTER 4
IMPLEMENTATION

This chapter details the implementation of event detection using the design proposed by Anwar et al. [Anw93] and the architecture highlighted in the previous chapter. Our implementation uses the Open OODB Toolkit from Texas Instruments, Dallas as the underlying platform. The local event detector and the rule manager have been implemented. We discuss the rule format and how we translate a high level rule specification to Sentinel system calls followed by the details of our implementation. A detailed example can be found in the appendix, parts of which are used in this chapter for explanation.

4.1 Rule Management

To allow users to specify events and rules at an abstarct level we introduce an high level event/rule format. This event/rule format is preprocessed and changed into Sentinel system calls.

The syntax of a Sentinel event/rule specification is:

$$\begin{aligned}
&\text{event } [\text{begin}(eventName)] \quad [\&\&\ \text{end}(eventName)]\ method_name \\
&\text{event } eventName = \qquad\qquad event_expression \\
&\text{rule } ruleName([eventName =] \qquad event_expression \mid eventName, \\
&\qquad\qquad condition_function, action_function \\
&\qquad\qquad [[, parameter_context][, coupling_mode] \\
&\qquad\qquad [, priority][, rule_trigger_mode]])
\end{aligned}$$

When dealing with methods as primitive events, it is necessary to specify the event interface so that the methods that generate events are clearly identified. Both begin-method (by indicating **begin**(*eventName*)) and end-method events (by indicating **end**(*eventName*) are supported. This event interface specification is pre-processed by adding wrapper methods to notify the event detector when they are invoked. The eventName specified is optional and we can have only the begin or end of a method as an event. By default end of a method is taken to be the event generator. For primitive events specified as part of the interface, the user is allowed to use them directly in the application program by prefixing the classname i.e., *className_eventName* for defining additional event expressions.

Event expressions specify primitive and composite events using event specification detailed in Snoop [Cha94a] which supports a number of event operators (e.g., and, or, sequence, aperiodic). The BNF of the event specification language can be found in [Mis91]. We allow an optional *eventName* to be specified within the event/rule definition to allow the users to name an event expression for subsequent usage.

Currently, the condition and action component of a rule are global functions.[1] The condition function returns an integer to indicate whether the condition is satisfied or not. The action function does not return any value.

The optional parameter *parameter_context* provides the context for detecting an event expression as well as for computing its parameters. Although the parameter context is meaningful only to an event (for its detection), specifying it along with the event limits the utility of an event definition. If several rules need to be defined on the same event in different parameter contexts, then the event has to be duplicated

[1]Currently, only functions are used for specifying condition/action. We plan on using the ZQL[C++] of Open OODB in the future. In the current host environment, methods cannot be used for condition/action as their invocation is tied to an object which is not known at compile time.

for each context. By allowing a late binding of parameter context (i.e., at the rule specification time instead of at event specification time), reusability of events is readily supported. Furthermore, common event sub-expressions are represented only once in the event graph (a graph representing an event expression; this is analogous to an operator graph) reducing the total number of nodes. However, this has a bearing on the event detection algorithm and the data structures employed as the same event may have to be detected in multiple contexts. The default context is assumed to be *Recent* since all the other parameter contexts have larger storage requirements. Once a context is specified it is propagated to all the nodes of the event graph associated with the rule and the parameters are collected thereafter.

Coupling_mode refers to the execution points. Currently, immediate and deferred coupling modes are supported between event and condition-action pair. Although our architecture lends itself for supporting detached mode, as we have discussed earlier there are some implementation difficulties for supporting this mode. Although [Hsu88] suggest a finer granularity for coupling modes (i.e., separating event, condition and action), we view the condition and action as a unit and use the coupling modes between event and condition_action pair.

We use *priority* classes for specifying rule priority. An arbitrary number of priority classes can be defined and totally ordered. A rule is assigned to a priority class either by indicating its number or the name of the class. As our implementation supports concurrent and nested rule execution (using light weight processes), priority of rules need to be resolved at different levels of execution. Our current approach provides a global conflict resolution mechanism among the priority classes and concurrent execution of rules that belong to the same priority class. This approach combines the advantages of both integer priority schemes and precedes/follows schemes. This

approach will also allow us to change rule priority categories based on the context or inherit priorities from users/applications.

We allow rule specification at class definition time and as part of an application. We also support rule activation and deactivation at runtime. Moreover, named events can be reused later. This implies that a number of rules may be defined on the same event expression and the event expression might have been defined prior to the rule definition time. As a result, it is possible that a rule gets triggered by event occurrences that temporally precede the rule definition time itself. As this might not be desirable in all situations, we provide an option (*rule_trigger_mode*) for specifying the time from which event occurrences to be considered for the rule. Two options, NOW (start detecting all component events starting from this time instant) and PREVIOUS (all component events since the event was detected last are acceptable) are supported as rule triggering modes, with NOW being the default.

The user-level rule specification given above is pre-processed into C++ statements that create rule and event objects. This specification helps to hide the details of rule/event implementation from the user. Furthermore, the syntax of a rule is the same for both **class level** and **instance level** rules. A class level rule satisfies the inheritance property. Even as part of the application, rules having *primitive events* can be specified as applicable to an entire class or an instance of that class as shown below.

```
REACTIVE Stock;
Stock IBM;
event any_stk_price('any_stk_price', 'Stock', 'begin',
                    'set_price(float price)');
event set_IBM_price('set_IBM_price', IBM, 'begin',
```

```
'set_price(float price)');
```

Here the character string 'Stock' is a class name and IBM is an instance of that class. The primitive event any_stk_price defines a class level primitive event. This event will be detected for all instances of this class whenever the method 'set_price' is invoked. The event 'set_IBM_price' will be invoked only when the same method 'set_price' is invoked on the IBM object. A rule defined on 'any_stk_price' will be a class level rule and a rule on 'set_IBM_price' will be an instance level rule. The specification of class/instance at the primitive event level allows us to have event expressions with class level as well as instance level events and hence rule specification which has mixed instance specification. Note that the event name is different although both the events are specified on the same method 'set_price'. A rule which contains all constituent primitive events as class level primitive events is termed a class level rule. Likewise a rule declared on only instance level primitive events is an instance level rule. Any class whose events are used in rules (either class level or instance level) need to be reactive (i.e., subclass of the REACTIVE class). When a user-defined reactive class is pre-processed, appropriate primitive events and rule declarations are generated and inserted in the application program. Since this rule will subscribe to an event expression that is specified on a class level, this rule will be notified whenever any object of this class invokes the method that are potential event generators.

4.2 Event Detection

4.2.1 Primitive Event Detection

In Sentinel external events are assumed to be explicitly signaled to the local event detector. All objects that can signal events must be derived from the REACTIVE class. The extensibility of the system is achieved by making the system class of Open OODB (namely OODB) REACTIVE. We specify an event interface to make

the methods beginTransaction and commitTransaction of this class generate events which result in actions used for deferred rule execution and flushing of all the event occurrences from the event graph, etc. Although rules are subclasses of the Notifiable class, methods of the Rule class can themselves be event generators. A rule class can be both reactive and notifiable. A runtime rule defined as

```
rule R1(e4 = (e1;e2)^e3, checkSalary, resetSalary, CHRONICLE,
           DEFERRED, 10, NOW)
```

is preprocessed by our system as

```
EVENT *e4 = new AND(SEQ(e1,e2),e3)
RULE *R1 = new RULE(``R1'', e4, checkSalary, resetSalary, CHRONICLE)
R1->set_coupling_mode(DEFERRED)
R1->set_priority(10)
R1->set_trigger_mode(NOW)
```

The methods that can generate primitive events are modified by using the wrapper class methods using the sentry feature of the Open OODB system while preprocessing the application program. This is accomplished by renaming the original method as user_method and creating a wrapper method which has the original method name. The wrapper method does the required signaling to notify the local event detector before and/or after the invocation of the user_method (according to the event interface specification). Each method which can generate an event (either at the beginning or at the end) is extended by adding code for parameter collection and notification to the event detector.

Since conditions are assumed to be side-effect free, we have to avoid detecting events that may be generated during condition execution. This can happen if conditions invoke methods that are declared as event generators in the event interface.

To disable the signaling of a primitive event when the condition function is executed, we set an attribute of the local event detector which indicates whether the events signaled should be acknowledged or not.

4.2.2 Composite Event Detection

Composite event specifications are pre-processed and code for generating event graphs at execution time are generated. Leaf nodes of the event graph corresponds to primitive or external events. Internal nodes correspond to event sub-expressions or rules. Each node has a list of subscribers to whom it has to notify once the event denoted by that node is detected. For example, a primitive event will have a list of subscribers which may contain rules and other event expressions in which it takes part. The same mechanism is uniformly used for composite events as well. Since primitive events, composite events as well as rules are derived from a base EVENT class, the subscribers' list is implemented as a linked list by specifying it as an attribute of the EVENT class. Every node of the event graph has outgoing edges equal to the number of subscribers it has. The Event Detector is also implemented as a class and we have a single instance of this class per application (termed the local event detector). Each of the primitive events defined is maintained as a list based on the class on which it is defined. When the local event detector is notified of a method invocation for a class by the DBMS, it is propagated only to the primitive events defined for that class. The local event detector maintains separate lists for temporal and explicit events. Once a primitive event node is notified it checks the method signature with the one that has been sent. If it matches, it notifies all its subscribers. Similarly once a complex event node is notified, it is activated based on the operator semantics [Cha93], and notifies subscribers in its list. A rule node, in addition to notification, creates a thread with the condition and action function

as a unit to be executed when the thread is scheduled. The local event detector schedules these threads. Our implementation based on event graphs is demand-driven (analogous to a data-flow scheme) and does not propagate parameters to irrelevant nodes. Furthermore, this approach efficiently supports subscribe/unsubscribe of rules to events as insertion/deletion in its subscriber's list.

4.3 Rule Execution and Scheduling

All primitive event signaling is done by invoking methods of the local event detector. Since this object is visible to the entire application, the nested triggering of rules by the execution of action function is also readily accomplished. As detailed above, when a primitive event is signaled the local event detector determines which of the primitive event nodes should be notified. Once this is done, the events propagate to the root nodes of the event graph. Whenever a rule is triggered in immediate coupling mode, it gets a free thread id and transforms the function which checks the condition and performs the action to a thread with the appropriate priority. Once all the immediate rules are in the form of threads, the local event detector suspends the main application and allows the rules to execute. Once all the rules are executed it resumes the main transaction.

Since a deferred rule is executed as a transformed rule (with an A^* event) in immediate coupling mode, it is triggered only when pre_commit primitive event is signaled by the transaction manager and hence it is treated in the same way as an immediate rule. Consider

```
REACTIVE Stock;
event any_stk_price('any_stk_price','Stock','begin',
                    'set_price(float price)');
rule R1(any_stk_price, checkSalary, resetSalary, CHRONICLE,
```

```
        DEFERRED);
```

The above rule is translated internally (and rule R1 is modified to reflect immediate mode) to

```
EVENT deferred_R1 = new A*(beg_trans, any_stk_price, pre_commit);
deferred_R1->subscribe(R1);
```

The event to be monitored is changed to an A^* event and the rule subscribes to the new A^* event created. Since threads are scheduled in a priority based preemptive manner, among rules scheduling is based on their priority and in the case of **multiple rules** with the same priority, scheduling is according to their time of initiation.

To implement the detached rule execution a global event detector has to be developed. This can be done using Remote Procedure Calls. The global event detector is used to support inter-application rules (global events spanning across several applications). It communicates with the local event detector for receiving events detected locally. Though detached coupling mode is accepted in the specification, it is not yet implemented as its implementation depends on the global event detector which is currently being designed.

The **nested rule** triggering is handled by assigning priorities to threads based on their level and the priority of the rule that triggered them. We currently support depth first execution of rules using the priority class of the triggering rule and the priority class of the triggered rule to compute a new priority value. For example, if the rule triggered has a priority 9 and the nested rule triggered has a priority 5, we assign the priority 14 to the nested rule and since it has a higher priority than the currently executing rule, it is executed first before the triggering rule is completed. So our rule execution proceeds in a depth first manner.

4.4 Parameter Computation

Our implementation uses the same event graph to detect an event in different contexts. Each node of the graph maintains all the contexts in which it has to collect parameters as well as to whom it has to propagate the parameters. It also has one counter for each parameter context. Whenever a rule is defined its context is propogated to all the nodes in its event graph. The counter for that particular context is incremented. If the counter was previously 0, the set of nodes corresponding to the event expression starts detecting events in this context. Specifying PREVIOUS (for rule_trigger_mode) for this rule will not have any effect. Introduction of this mechanism for event detection in the presence of contexts helps avoid detecting events in the continuous and cumulative mode as they have significant storage requirements. Once a rule is disabled or deleted the event expressions are again notified and the respective counter is decremented. If the counter is reset to 0 events are no longer detected in that context. *Recent* is used as the default context.

Composite events pose additional problems for parameter computation. The difficulties involved in passing complex data types as parameters across applications has been detailed in the previous section. To avoid these pitfalls, currently, we have decided to pass only simple data types (e.g., integer, float, character and string by value) as parameters. Although it is possible, copying the values of complex data types will add considerable storage overhead. The parameters and component events are all maintained as linked lists at the relevant nodes and hence there is no copying of data. Only the pointers have to be adjusted thereby increasing the efficiency of event detection. The event detector and rule manager implemented lends easily for **Online** as well as **Batch detection** of events/rules.

Events crossing transaction boundaries: The logical unit of work in a DBMS is a transaction. To maintain the correctness of this concept we have to ensure that events (as well as parameters associated with the event) are not carried over across transaction boundaries. This is especially important in the presence of composite events whose detection can span an arbitrary time interval. Consider the case when Transaction1 invokes certain methods and is later aborted. These methods might have triggered certain primitive events whose parameters are recorded in the event graph. If these events are not flushed when a transaction is aborted (or committed), these events can participate in composite events for another transaction. If we allow events to span transactions, a second transaction might cause the firing of a rule which has constituent primitive events and parameters from a previously aborted transaction. This means that the condition and action functions access parameters which in the database sense does not exist at all (since the previous transaction was aborted, all its effects would have been rolled back in essence making it seem like that method was never executed). The above situation can arise for committed transactions as well although the parameter values may be consistent in this case.

We provide a flush operation that can either flush the event graph selectively for an event expression or for the entire graph. This is invoked as an action of a rule on abort and commit events. Selective flushing must be done by the application. Flushing of all event graphs are managed as rules over the primitive events begin transaction, commit and abort. However, these can be easily modified by deactivating these rules if events across transaction boundaries need to be detected.

4.5 Example Applications

Military Application A real-world military application was chosen to test our extensions to Open OODB. The military consists of various units, positioned at various locations for performing some task. Each unit has a readiness status indicating whether they are in a position to perform certain operation. For example, we could define readiness based on personnel, training, supplies etc. Readiness is maintained in terms of ratings with a value of 1 signifying Combat Ready and 5 signifying Overhaul. A readiness rating of 2 or below is desired for any unit.

As and when a crisis arises a plan has to be prepared to deal with it. To each plan a set of units have to be assigned to carry out the plan. Once this is done any change to either the plan or a unit's readiness status has to be monitored continuously. In a passive DBMS environment this task (of monitoring) is done manually by running a query. In our application we defined rules based on the reports that they get on plan changes and unit readiness status. Once the event, condition and action were defined it was easier to do tasks like *data integrity checking, situation monitoring* and *alerting*. The concept of passing parameters was found to be very useful as certain updates had to be disqualified immediately. Also developing this application helped us to tailor our system to suit real-world semantics.

Stock Application Since the Military application did not involve complex events to be monitored except 'OR', a prototype stock application was also developed. This was used to test the event expression that we have discussed before, involving the various parameter contexts.

The Stock application involved monitoring the price index of various companies and buying stocks whenever the price reached a certain level. This also involved defining rules on multiple classes. Instance level events and class level events were

also defined and tested. For example, the change in price of IBM stock was given as an instance level rule and the change in stock level of any "Stock" class object was defined as a class level event. Complex event expressions were formed on these events and rules defined on them. This helped us establish a unified approach of relating to class level and instance level event specification.

The applications described briefly have helped us in culling the initial requirements that are necessary to use our system for any real-world application. We plan to choose an application domain which uses most of the features we have provided and develop it in full scale to fine tune our system and make it functionally complete.

CHAPTER 5
OVERVIEW OF RELATED WORK

5.1 Ode

Ode [Geh92b, Geh92a] is a database system and environment based on the object paradigm. The database is defined, queried and manipulated using the database programming language O++, which is an upward compatible version of C++. Ode provides active behavior by the incorporation of *constraints* and *triggers* [Geh91]. Constraints and triggers are defined declaratively within a class definition and consist of a condition and action. Constraints are used for maintaining object consistency and are applicable to all instances of the class in which they are declared. Triggers, on the other hand, are used for other purposes and are applicable only to those instances of the class in which they are declared. Ode uses an extended finite automata for composite event detection and triggering of constraints and triggers. The extended automaton, makes a transition at the occurrence of each event in the history like a regular automaton and in addition handles attributes of the events to compute a set of relations at the transition.

Comments on Ode

- Note that the differentiation between constraints and triggers is only for convenience [Geh91] and does not contribute in any way to rule processing. Furthermore, triggers may be used to specify constraints. Hence essentially Ode maps rules to triggers.

- Ode represents an event occurrence as a tuple of the form *(primitive event, event identifier)*. An example of an event identifier is defined by Gehani et al. [Geh92a] as the time at which the primitive event occurred. An event history is defined as a finite set of event occurrences with no two event occurrence having the same event identifier. Event expression specification in the case of 'AND' operator is specified as an intersection of two event histories. The field on which this intersection is performed is not specified. If it is on the event identifier then the 'AND' operator recognizes only simultaneous occurrence of events. It is stated that two event occurrences e1 and e2 refer to the same event occurrence if their eids are identical which rules out the possibility of simultaneous occurrence of events. The other alternative is to perform the intersection with respect to the primitive event or the entire tuple of an event occurrence in which case the result will always be an empty set. Most of the operators in Ode are defined in terms of the 'AND' operator and since this definition itself is questionable these operator semantics are also unclear.

- The automaton for the 'AND' operator constructed according to the specification given [Geh92a] does not seem to reach an accepting state.

- In the case of automata construction for the expression employing the pipe operator E|F according to the specification given [Geh92a], if E and F are primitive all the states will be non-accepting states which is not the desired result.

- In the case of an event occurrence each constraint and trigger has to be evaluated, i.e., each finite automaton constructed has to be checked to see if there are any transitions. This leads to excessive checking.

- Also there is no specification of priority in the case of constraints and triggers and hence they seem to be activated in an arbitrary manner.

- In the case of implementation a suite of finite automatons are generated if an attribute is specified, for each different value of the attribute. So further detection should satisfy all the automatons generated and is a potential bottleneck. We have overcome this problem by shifting the burden of checking the attributes to the condition function written for a rule.

5.2 SAMOS

The combination of active and object-oriented characteristics within one, coherent system is the overall goal of SAMOS (**S**wiss **A**ctive **M**echanism Based **O**bject-Oriented Database **S**ystem). Samos [Gat93, Gat94] addresses event specification and detection in the context of active databases. Although there are some differences between Snoop and Samos in the event specification language (for example, Samos has a Times operator for defining the occurrence of n events in an interval which can be specified as Any(n, E^*) in our event specification), they differ primarily in the mechanism used for event detection. Samos uses modified colored Petri nets called SAMOS Petri Nets to allow flow of information about the event parameters in addition to the occurrence of an event.

Comments on SAMOS

- When an event participates in more than one composition, (e.g., in E=(E1;E2) and in EE=(E1,E3)) to combine the Petri Nets for the two composite events, E1 has to be duplicated into E1' and E1". This results in duplicating Petri nets equal to the number of common event expressions that E1 participates in. Since all duplicates must also be represented in the data structure this might

lead to excessive storage requirements. Our implementation uses linked lists for the subscribers list and hence overcomes the need for duplication.

- In Samos only the chronicle context is supported. As we have highlighted before, the other contexts are also useful in various application domains. The semantics of contexts is built into our operator nodes. Hence it is easy for us to have a single instance of the event graph and detect all the contexts. In the case of Petri nets they have to generate a different Petri net for each context. Also, we can generate the event graph as and when the event expression is specified even if the context information is not specified. If contexts are introduced in Petri nets then they cannot be built unless the context information is specified beforehand.

- In the case of implementation, since ObjectStore is a blackbox Samos uses the layered approach for providing active capability. In a layered approach the underlying DBMS is augmented with a layer that is responsible for providing active capability. The architecture shown permits access to the augmented system either through a user interface tool that transforms user active database design to underlying system constructs or through a stand-alone interface. All applications that require active capability have to interact with the system through this layer; otherwise, active capability will not be available. Although full active capability cannot be obtained in this approach, a number of techniques can be used and some optimizations can be performed by the situation monitor layer. For instance, the layer can decide whether to rewrite a transaction to include the condition monitoring code (similar to the application-based architecture but the rewrite is done by the situation monitor layer) or use either the polling or aperiodic checking approach depending upon the meta-data used by

the system. The layer is responsible for monitoring the situations and executing appropriate rules which also means that all transactions are routed through the layer (although eventually processed by the underlying system's transaction manager). There may be some limitations on the class of ECA rules that can be supported using this approach. For example, immediate mode coupling may not be possible as the layer may not be able to suspend a transaction that is being executed by the underlying DBMS (even when the rewrite technique is used). Also, explicit and other temporal events cannot be supported in this approach without resorting to polling.

- They have addressed the issue of composite event detection and rule management but do not discuss the issues of rule execution.

5.3 ADAM

ADAM [Dia91] is an active OODB implemented in PROLOG. ADAM's main focus is to provide an uniform approach for defining and treating rules in the same way as other objects in the system. It adopts the ECA format for rules. Events and rules in ADAM are first class objects.

ADAM supports database events, clock events and application events. Events in ADAM are generated either *before* or *after* the execution of a method. Rules are incorporated in ADAM by using an object based mechanism. The attributes of a rule include an *event*, *condition*, *action*, *is_it_enabled*, *active_class* and *disabled_for*. Rule operations are implemented as methods. The attribute *active_class* in rules indicate which class this rule is monitoring. Correspondingly each class structure has a *class_rules* attribute that indicates which rules to check when the object raises an event. In order for ADAM to support inheritance of rules, each class definition is enlarged with an *activated_by* attribute. When an update is done to the class_rules

attribute of any class, the update is propogated to the *activated_by* attribute of all its subclasses. This process is performed automatically by the system.

Comments on ADAM

- Since complex events are not supported in ADAM attaching the rules as attributes to a class leads to efficient rule detection. In our case since rules span multiple classes it is not possible to use this approach. Our event detector keeps track of the primitive events defined on classes and the event graph maintains the pointers to rule objects wherever necessary. When ADAM is extended to include complex events it might have to use a similar approach where the class_rules attribute keeps track of events defined on this class.

- Since rules are treated as objects they can be created, deleted and modified like any other object. In this sense ADAM provides a uniform treatment of rules in an object oriented context.

- Since complex events are not part of the system the concept of parameter contexts is not applicable to this system.

- ADAM supports only the immediate coupling mode.

- Inheritance of rules is supported. But the way in which it is supported is specific to Prolog in which the system has been implemented and hence cannot be adopted to other environments.

- In ADAM all the method's arguments are passed as parameters to the rule. It is the same as our case except that we restrict our parameters to only simple data types and oids.

- Supporting instance level rule is not possible in this system. This requires naming all the other instances in the 'disable_for' attribute of the rule and is cumbersome.

5.4 Alert

Alert [Sch91] is an extension architecture, implemented in the Starburst extensible DBMS at the IBM Almaden Research Center, for experimentation with active databases. Alert uses the layered approach and the inherent disadvantages that we have discussed in section 5.2 with respect to a layered approach applies to it as well.

Alert introduces the concept of active tables and active queries. Active tables are append-only tables. Active queries are queries that range over active tables. An SQL-like syntax is used to specify queries. Alert highlights that the active queries differ from the usual SQL-like passive queries only in their cursor behavior. The standard SQL does a non-blocking read: if no more tuples are available in the answer set of the query, the process doing a fetch is not blocked but is simply returned an EOF. In the case of active queries a *fetch-wait* is employed which is a blocking read i.e., if the current answer set is exhausted, the process doing a fetch-wait is blocked until one becomes available.

Alert has transaction coupling modes which specify whether a rule executes in the same transaction that triggered the rule or as a separate transaction. Rule execution coupling modes are used to specify whether a rule should be executed synchronously (rule execution is completed before triggering transaction continues) or asynchronously (rule and triggering transaction execute in parallel) with the triggering transaction. Immediate and deferred coupling modes with the semantics equivalent to ours are also present.

Comments on Alert

- By having a rule language similar to SQL, Alert reuses almost all of the existing semantic checking, optimization and execution implementations.
- Schreier et al. [Sch91] note that the fetch-wait process returns an EOF only if it is guaranteed that no more answer tuples will be generated. However the point at which it is decided that there are no more answer tuples is not clarified.
- In Alert the DBMS creates an active table for every user-defined passive table. But the details regarding what is maintain in the active tables are not specified. The queries given as examples by Schreier at al. [Sch91] indicate that the system-defined active tables have the same fields as the passive table. If it is so, then there is no justification for having a system-defined active table for each passive table.
- The users have to truncate the old tuples to limit the size of an active table. This shifts the burden of maintaining active tables to the user and might lead to non-detection of certain events if the user deletes the tuples arbitrarily.
- The issue of updating active tables when some tuples are deleted in the passive table is not addressed.
- The type of events that Alert detects is not discussed, leading us to believe that they do not support temporal and external events.
- The monitoring viewpoint of Alert is similar to busy waiting. Hence condition monitoring is not efficient.

- The system is more geared towards the Relational Model. In relational databases, events are generally restricted to database updates, but an OO environment allows any message to raise an event. Thus the efficiency requirements for rule support in OODBs are even greater than in relational databases. Hence extending the Alert system to an OO environment will involve significant extensions.

5.5 UBILAB system

The integration of an existing Smalltalk-based OODBMS - Gemstone and active DB functionality was the goal of the UBILAB system [Kot93]. This system also uses a layered approach but the limitations of this approach have been realized and identified in the paper.

Since the OODBMS is a black box, all additional capabilities are implemented using the officially accessible interfaces of the DBMS. Internal events (changes and method execution on specific objects), external events (events not related to any object or class), and time-related events (signal absolute and relative points in time) are supported. UBILAB system also supports the concept of simple events and complex events, the latter being defined by means of an event algebra. The complex events occur totally within the scope of one transaction.

In this system all ECA rules are mapped onto triggers. The trigger mechanism is just a pair of event and action without the notion of a condition. Event, action and triggers are classes to which the ECA rule specifications are mapped. Parameter lists are also associated with each trigger with an indication as to how they should be collected.

The implementation uses the concept of daemon processes to execute asynchronous actions. Separate daemons for each of the action types have been identified namely, DML actions, general UNIX actions and notification actions operating on the window

system. A high level interface for rule specification and debugging involving a rule designer, browser, simulator and tracer is currently under development.

Comments on UBILAB system

- Only *and*, *or*, *not* and *sequence (;)* operators are supported in this system. We support A, A^*, P and P^* operators in addition to these operators for defining complex event expressions.

- The UBILAB system does not have the concept of parameter contexts. From the discussion [Kot93] it appears that events are detected in the Chronicle context.

- The event detection features like the use of wrapper methods to signal events are similar to our implementation. Method wrappers are created dynamically as soon as rules refer to a method. It should be noted that the runtime creation of wrappers is specific to the implementation environment of the UBILAB system and is difficult for adopting to other environments.

- The concept of grouped events is expressed as a relationship between two subsequent events that belong to each other. A grouped event can be either a start event or a following event. For example,
  OP_A_begin[1]; OP_A_begin[2]; OP_A_END[2]; OP_A_END[1]
  is a complex grouped event which will be raised if during the execution of OP_A the same operation is once more executed completely. This is a useful mechanism and we are exploring the ways of providing this feature with our system.

- Only one event per expression is raised in this system even if multiple occurrences of the same event are detected. This simplifies the process of parameter collection and association. In our case since we have parameter contexts the way users might want to deal with parameters is not known. We provide all the parameters detected so far and let the user decide which is necessary for evaluating the condition and performing the action.

- There is no discussion on how this system deals with complex data types in case of parameter passing for event detection.

- Implicit grouping of objects to separate events relevant to one user group and central events visible to everybody is present in this system, which is highly desirable for our system.

5.6 K

K is a high-level knowledge base programming language developed at the University of Florida. It is used for doing general computation as well as for defining, querying, and manipulating databases in nontraditional application domains [Shy91]. The underlying semantic model of K is $OSAM^*$ [Su89] an extensible object-oriented semantic association model which provides a rich class system to support modeling, persistence, knowledge abstraction, encapsulation and multiple inheritance.

K provides declarative and expressive constructs to specify rules [Arr92]. It also takes care of rule management and execution. Here rules are associated with class definitions. Each rule of class X is specified as:

`rule` *rule_name* is

`triggered` *trigger_time operation_spec*

[`condition` [*guard_condition*] *rule_condition*]

[`action` *statements*]

[`otherwise` *statements*]

`end;`

The semantics of a rule is given by: If the *guard_condition* is false the whole rule is skipped. If the *guard_condition* is true and the *rule_condition* is also true then `action` statements are executed else the `otherwise` statements are executed. The condition parts can contain any boolean expression, including a database query. Both `action` and `otherwise` are optional, but atleast one of them should be specified. *Operation_spec* is the event, which can be either a database operation or user-defined method. The *trigger_time* specification can be *before*, *after*, or *immediate_after*. In the case of *before*, the rule is executed before the specified method/operation is executed. In the case of *immediate_after*, the rule is executed immediately after the method/operation is performed. *After* is similar to our deferred rule specification.

Comments on K

- K provides a uniform treatment of rules as objects.
- Inheritance of rules are supported.
- Temporal and external events are not supported.
- K supports only the disjunction of complex events. Other types of complex events are not supported.
- Index mechanism for selecting rules for execution is performed by using OQL queries.
- Rules cannot be explicitly specified to span multiple classes, but they are done indirectly by using parameterized rule declarations.

- Rules are declared at class definition time. They are stored in the database along with the class definition and are compiled as part of the schema definition. Since rules are created declaratively, modification of the event it monitors at run time is not possible. In our case rules are created dynamically enabling the modification of their attributes at run time.

From all the systems discussed so far we can conclude that the concept of parameter contexts is unique to our system. We also address an extensible rule management and execution system. We are aware of certain desirable features like grouping of events etc., and are in the process of incorporating these extensions in our system.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

6.1 Contributions and Conclusion

This thesis significantly extends our earlier work [Mis91, Cha94a] on an expressive event specification language. 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 thesis, 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 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 cannot be deleted. We define 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 and refine it to using initiator and terminator concepts. We have developed complete algorithms for detecting Snoop expressions in all parameter contexts. We then propose extensions to an object-oriented DBMS (Open OODB) and indicate the functionality supported by the architecture. The implementation – of event detection – using the design proposed by Anwar et al. [Anw93] and the architecture highlighted is the main contribution of this thesis. We

have presented an architecture for an active OODBMS and described its implementation. We have discussed the implementation details of ECA rule transformation, composite event detection and rule execution.

To summarize, the contributions of this thesis are:

- Introducing the notion of primitive event history and composite event history and defining the semantics of primitive and composite events over an event-history.
- Algorithms for detecting Snoop expressions in all parameter contexts.
- An extensible architecture for event detection and rule execution.
- Implementation of the local event detector involved
  - Seamless incorporation of ECA rules into a passive OODBMS Open OODB.
  - Supporting the immediate and deferred coupling modes proposed in HiPAC.
  - Supporting the specification and detection of complex and primitive events.
  - Allowing class level and instance level events/rules.
  - Supporting online as well as batch mode of rule execution.
  - Supporting prioritized rule scheduling.

6.2 Future Work

The underlying concepts behind our architecture and implementation can be easily adapted to the relational model as well. For example, the implementation of composite event detection can be easily tailored to a relational model since the individual linked lists maintained by the composite event detector can be viewed as tuples. Currently we are extending the preprocessor of Open OODB to convert our high level specification to low level object definitions and function calls.

Our future work includes

- Expanding the rule management support to public, private, and protected rules.
- Investigating efficient ways of providing the semantics of detached rule execution considering the limitations inherently present in an object oriented system. The implementation of the detached coupling mode entails generating an entire application with all the class definitions in the triggering application as the condition and action functions might refer to both program and database objects. The problems being addressed by OMG and Corba need to be resolved for the implementation of detached mode. An alternative is to extend the nested transactions semantics to include detached execution of rules.
- Implementation of a global event detector satisfying all the functionality highlighted in our architecture.
- Integrating the nested subtransaction model into the rule execution model.
- In this thesis, 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. All the event detection algorithms presented in this thesis extend readily when the oid is allowed as an explicit parameter of a primitive event.

The task of making a passive DBMS active with all the functionality highlighted in this thesis involves a tremendous amount of design as well as implementation. In this thesis we have addressed a subset of these issues and provided solutions to them. The implementation of the local event detector can be identified as a stepping stone for building the global event detector, which needs to be implemented to attain complete active capability.

APPENDIX A
COMPOSITE EVENT DETECTION ALGORITHMS

In this appendix, we present algorithms for event graph construction and detection in all the parameter contexts. Algorithms for all the contexts have been implemented and has been integrated with an object-oriented DBMS (the Open OODB) from Texas Instruments, Dallas.

Figures A.1, A.2, A.3, A.4 illustrate the recent, chronicle, continuous and cumulative event detection algorithms for the event expression X = ((E1 $\triangle$ E2) ; E3 ; (E2 $\triangle$ E4)) shown in the body of the paper. The time line indicates the relative order of the primitive events with respect to their time of occurrences. Events are propagated in a bottom-up fashion. The sequence of the graphs are from left to right and top to bottom. Leaf nodes of the graph correspond to primitive events and pass the events as they occur to their parent nodes. The operator nodes have separate storage for each of their children. The different instances of the same event are stored as separate entries and are given in separate lines in the figure. A small arrow indicates the primitive event detected at that point of time. The arrows pointing from the child to its parent in the graph indicates the detection of a composite event and flow of the detected events. The event instances that are deleted after a composite event is detected and propagated are indicated in bold letters. A walk-through example of each context on a single graph instance has been discussed in the body of the paper.

ALGORITHM Composite Event Detection

Construct an event graph for each rule with nodes as operators and leaves

as primitive events. The primitive event nodes are the source and the rule nodes are sinks. Edges are from constituent events to composite event.

Initialize counters (e.g., num_events) and flags.

For each occurrence of a primitive event

store its parameter in the corresponding terminal node 't';

activate_terminal_node(t);

PROCEDURE activate_terminal_node(n)

For all rule-ids attached to the node 'n'

signal event;

For all outgoing edges i from 'n'

propagate parameters in node 'n' to the $node_i$ connected by edge i

activate_operator_node($node_i$);

Delete propagated entries in the parameter list at 'n'

Algorithm for the Recent Context

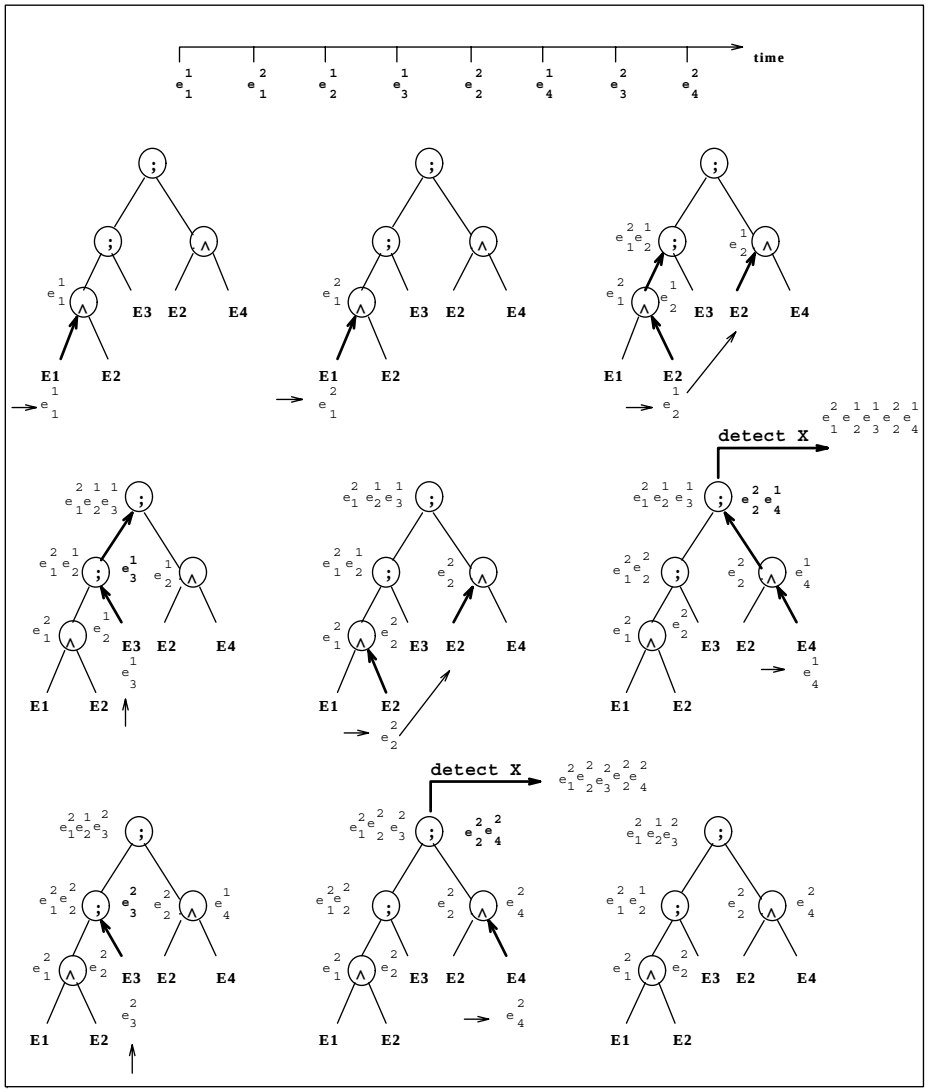

Figure A.1. Detection of X in recent mode

PROCEDURE activate_operator_node($node_i$) /* **Recent Context** */

`CASE` $node_i$ `is of type`

/* a primitive or composite event has been signalled to $node_i$ */

AND(E1, E2): if left event e1 is signalled

if E2's list is not empty

Pass <e2, e1> to the parent

Replace e1 in E1's list.

if right event e2 is signalled

if E1's list is not empty

```
                        Pass <e1, e2> to the parent
                      Replace e2 in E2's list

OR(E1, E2):           For any event <e> signalled
                        Pass <e> to the parent

ANY(m,E1,E2,...,En):When an event ek is signalled
                      Replace ek in it's event list Ek.
                      Increment the counter num_events only
                      if Ek list was empty previously
                      if num_events >= m
                        Find all event tuples (taken from their
                        respective event lists)
                        <ei, ej, ek ...> such that
                        they are the most recent m distinct
                        occurrences of events.
                        Pass the tuple to the parent
                        change num_events appropriately

SEQ(E1, E2):          if left event e1 is signalled
                        Replace e1 in E1's list

                      if right event e2 is signalled
                        if E1 list is not empty
                          Pass <e1, e2> to parent

A(E1,E2,E3) |
  P(E1,[t],E3):       if left event e1 is signalled
                        Replace e1 (only t_occ & event name) in E1's list

                      if middle event e2 is signalled
                        if E1's list is not empty
                          Pass <e1, e2> to parent

                      if right event e3 is signalled
                        FLUSH E1's buffer

A*(E1,E2,E3)|
  P*(E1,[t],E3):      if left event e1 is signalled
                        Replace e1 (t_occ & event name) in E1's list
                        FLUSH E2's buffer

                      if middle event e2 is signalled
                        if E1's list is not empty
                          Append e2 to E2's list
```

```
                    if right event e3 is signalled
                      if E2's list is not empty
                        Pass <e1, e2, e3> to parent
                        FLUSH E1 and E2's buffers
                      else
                        Pass <e1, e3> to parent
                        FLUSH E1's buffer

NOT(E2)[E1, E3]:    if left event e1 is signalled
                      Replace e1 (only t_occ & event name) in E1's list

                    if middle event e2 is signalled
                      if E1's list is not empty
                        FLUSH E1's buffer

                    if right event e3 is signalled
                      if E1's list is not empty
                        Pass <e1, e3> to parent
                        FLUSH E1's buffer
```

Algorithm for the Chronicle Context

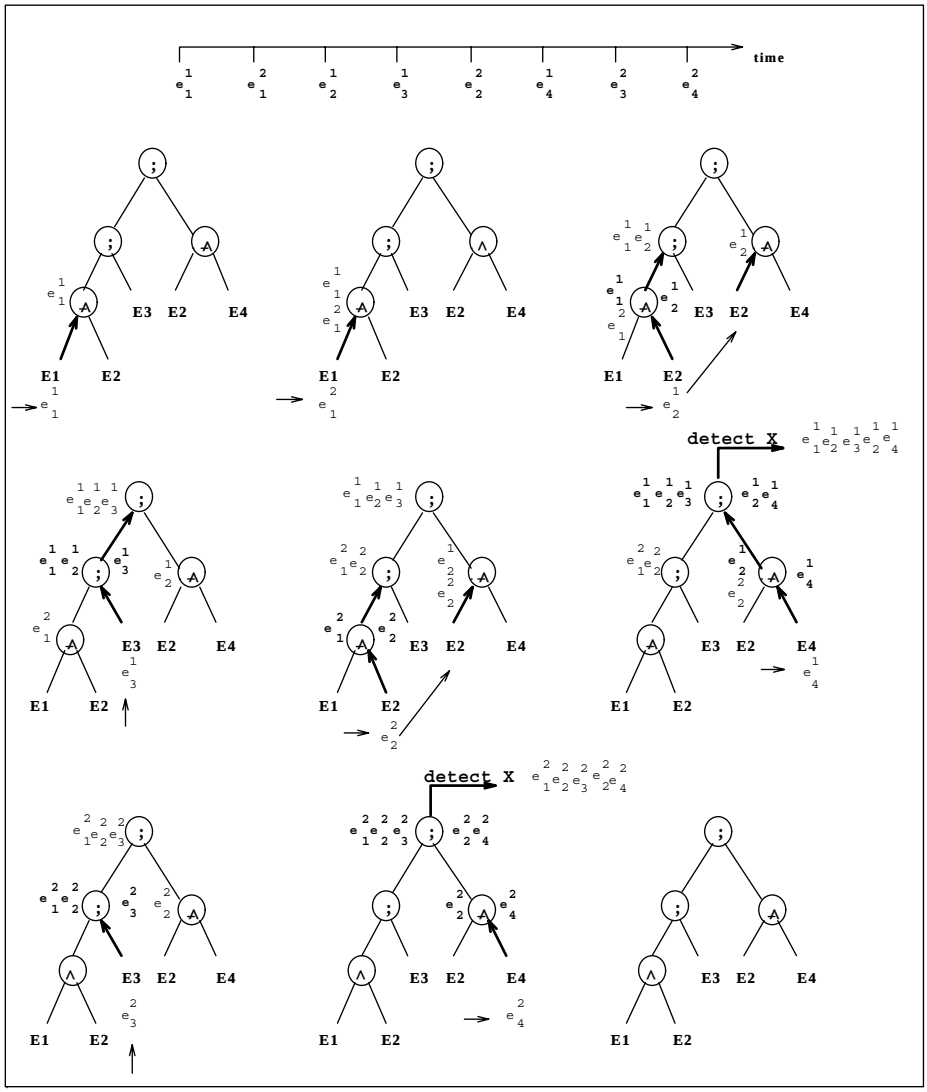

Figure A.2. Detection of X in chronicle mode

PROCEDURE activate_operator_node($node_i$) /* **Chronicle Context** */
`CASE` $node_i$ `is of type`
/* a primitive or composite event has been signalled to $node_i$ */
AND(E1, E2): if left event e1 is signalled
if E2's list is not empty
Pass <E2's head, e1> to the parent
Delete head of E2's list
else Append e1 to E1's list

if right event e2 is signalled

```
                        if E1's list is not empty
                          Pass <E1's head, e2> to the parent
                          Delete head of E1's list
                        else Append e2 to E2's list

OR(E1, E2):             For any event <e> signalled
                          Pass <e> to the parent

ANY(m,E1,E2,...,En):When an event ek is signalled
                    Append ek in it's event list Ek.
                    Increment the counter num_events only
                    if Ek list was empty previously
                    if num_events >= m
                      Find all event tuples (taken from their
                      respective event lists)
                      <ei, ej, ek ...> such that
                      they are the oldest (head) m distinct
                      occurrences of events
                      Pass the tuple to the parent and delete them from
                      their respective event lists
                      change num_events appropriately

SEQ(E1, E2):            if left event e1 is signalled
                          Append e1 to E1's list

                        if right event e2 is signalled
                          if E1 list is not empty
                            Pass <E1's head, e2> to parent
                            Delete head of E1's list

A(E1,E2,E3)|
 P(E1,[t],E3):          if left event e1 is signalled
                          Append e1 (only t_occ & event name) to E1's list

                        if middle event e2 is signalled
                          if E1's list is not empty
                            Pass <E1's head, e2> to parent
                            Delete head of E1's list

                        if right event e3 is signalled FLUSH E1's buffer

A*(E1, E2, E3)|
 P*(E1,[t],E3):         if left event e1 is signalled
                          Append e1 (t_occ & event name) to E1's list
```

if middle event e2 is signalled
  if E1's list is not empty Append e2 to E2's list

if right event e3 is signalled
  if E2's list is not empty
    Pass <E1's head, All e2's in E2's list, e3> to parent
    Delete head of E1's list
    Delete all e2's in E2's list whose t_occ
    is less than the E1's head t_occ
    If E1's list is empty then FLUSH E2's buffer
  else Pass <E1's head, e3> to parent
    Delete head of E1's list

NOT(E2)[E1, E3]: if left event e1 is signalled
  Append e1 (only t_occ & event name) to E1's list

if middle event e2 is signalled
  if E1's list is not empty FLUSH E1's buffer

if right event e3 is signalled
  if E1's list is not empty
    pass <E1's head, e3> to parent
    Delete head of E1's list

Algorithm for the Continuous Context

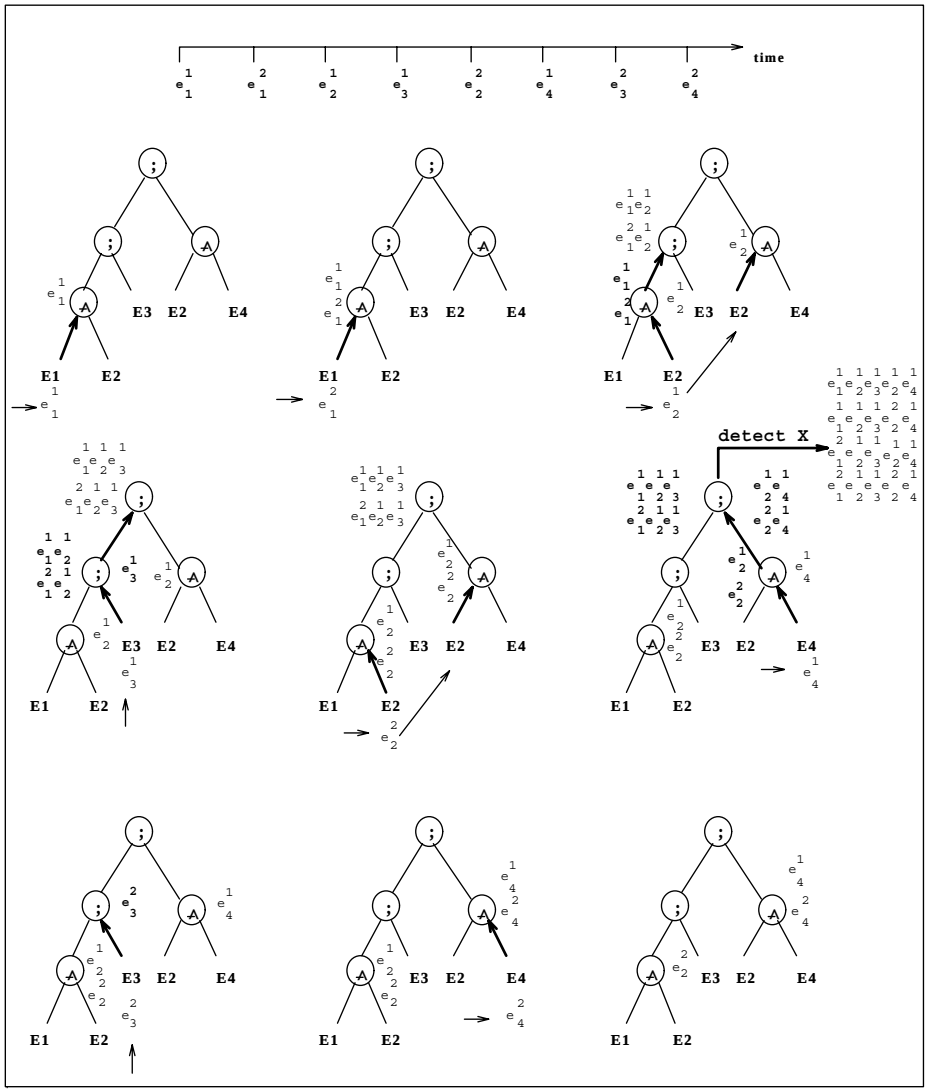

Figure A.3. Detection of X in continuous mode

PROCEDURE activate_operator_node($node_i$) /* **Continuous Context** */
    `CASE` $node_i$ `is of type`
    /* a primitive or composite event has been signalled to $node_i$ */
        AND(E1, E2): if left event e1 is signalled
                        if E2's list is not empty
                          For every event e2 in E2's list
                              Pass <e2, e1> to the parent
                          FLUSH E2's list
                        else
                          Append e1 to E1's list

```
                    if right event e2 is signalled
                      if E1's list is not empty
                        For every event e1 in E1's list
                            Pass <e1, e2> to the parent
                        FLUSH E1's list
                      else
                        Append e2 to E2's list

OR(E1, E2):         For any event <e> signalled
                    Pass<e> to the parent

ANY(m,E1,E2,...,En):When an event ek is signalled
                    Append ek in it's event list Ek.
                    Increment the counter num_events only
                    if Ek list was empty previously
                    if num_events = m
                      Find all event tuples (taken from their
                      respective event lists)
                      <ei, ej, ek ...> such that
                      they form m distinct occurrences of events
                      Pass the tuple to the parent
                      FLUSH all E1, E2, ..., En buffers
                      Set num_events to 0

SEQ(E1, E2):        if left event e1 is signalled
                      Append e1 to E1's list

                    if right event e2 is signalled
                      if E1 list is not empty
                        For every event e1 in E1's list
                            Pass <e1, e2> to parent
                        FLUSH E1's list

A(E1,E2,E3)|
 P(E1,[t],E3):      if left event e1 is signalled
                      Append e1 (only t_occ & event name) to E1's list

                    if middle event e2 is signalled
                      if E1's list is not empty
                        For every event e1 in E1's list
                            Pass <e1, e2> to parent

                    if right event e3 is signalled
                      FLUSH E1's list
```

A^*(E1, E2, E3)|
P^*(E1,[t],E3):

```
if left event e1 is signalled
  Append e1 (t_occ & event name) to E1's list

if middle event e2 is signalled
  if E1's list is not empty
    Append e2 to E2's list

if right event e3 is signalled
  if E2's list is not empty
    For each e1 in E1's list
        Pass <e1, All e2's whose t_occ is greater
        than t_occ(e1), e3> to parent
    FLUSH E1's and E2's buffers
  else
    For each e1 in E1's list
        Pass <E1's head, e3> to parent
    FLUSH E1's buffers
```

NOT(E2)[E1, E3]:

```
if left event e1 is signalled
  Append e1 (only t_occ & event name) to E1's list

if middle event e2 is signalled
  if E1's list is not empty
    FLUSH E1's buffer

if right event e3 is signalled
  if E1's list is not empty
    For each e1 in E1's list
    Pass <e1, e3> to parent
    FLUSH E1's buffer
```

Algorithm for the Cumulative Context

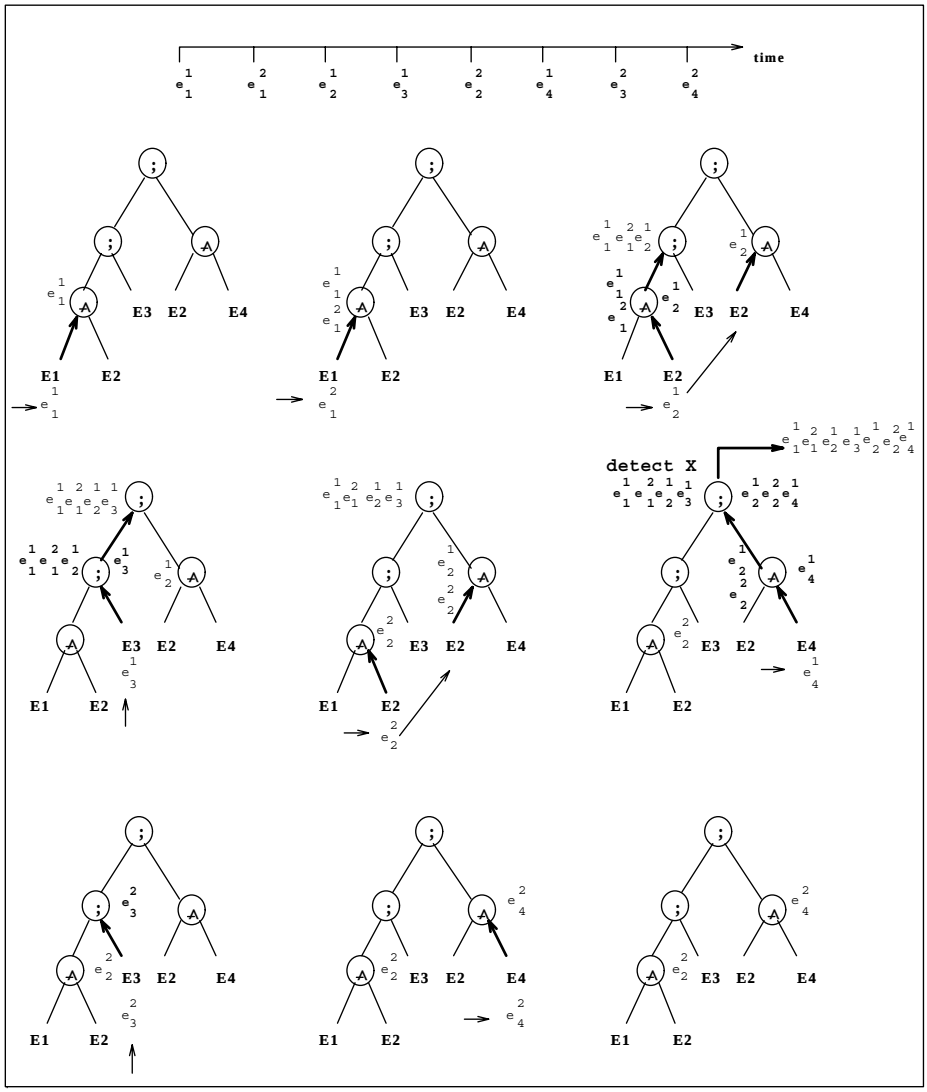

Figure A.4. Detection of X in cumulative mode

PROCEDURE activate_operator_node($node_i$) /* **Cumulative Context** */
    `CASE` $node_i$ `is of type`
    /* a primitive or composite event has been signalled to $node_i$ */
        AND(E1, E2): if left event e1 is signalled
                        if E2's list is not empty
                          Pass <all e2's, e1> to the parent
                          FLUSH E2's buffer
                        else
                          Append e1 to E1's list

```
                    if right event e2 is signalled
                      if E1's list is not empty
                        Pass <all e1's, e2> to the parent
                        FLUSH E1's buffer
                      else
                        Append e2 to E2's list

OR(E1, E2):         For any event <e> signalled
                    Pass <e> to the parent

ANY(m,E1,E2,...,En):When an event ek is signalled
                    Append ek in it's event list Ek.
                    Increment the counter num_events only
                    if Ek list was empty previously
                    if num_events = m
                      Find all event tuples (taken from their respective
                      event lists) <all ei, all ej, all ek ...> such that
                      they are the m distinct occurrences of events
                      Pass the tuple to the parent and
                      FLUSH all E1, E2, ... En buffers
                      Set num_events to 0

SEQ(E1, E2):        if left event e1 is signalled
                      Append e1 to E1's list

                    if right event e2 is signalled
                      if E1 list is not empty
                        Pass<all e1's, e2>to parent
                        FLUSH E1's buffer

A(E1, E2, E3)|
 P(E1,[t],E3):      if left event e1 is signalled
                      if E1's list is empty
                        Store e1 (only t_occ & event name) in E1's list

                    if middle event e2 is signalled
                      if E1's list is not empty
                        Pass <e1, e2> to parent
                        FLUSH E1's buffer

                    if right event e3 is signalled
                      FLUSH E1's buffer

A*(E1,E2,E3)|
 P*(E1,[t],E3):     if left event e1 is signalled
```

```
                    if E1's list is not empty
                      Store e1 (t_occ & event name) in E1's list

                  if middle event e2 is signalled
                    if E1's list is not empty
                      Append e2 to E2's list

                  if right event e3 is signalled
                    if E2's list is not empty
                      Pass <e1, All e2's in E2's list, e3> to parent
                      FLUSH E1 and E2's buffer
                    else
                      Pass <e1, e3>to parent
                      FLUSH E1's buffer

NOT(E2)[E1, E3]:  if left event e1 is signalled
                    if E1's list is not empty
                      Store e1 (t_occ & event name) in E1's list

                  if middle event e2 is signalled
                    if E1's list is not empty
                      FLUSH E1's buffer

                  if right event e3 is signalled
                    if E1's list is not empty
                      Pass <e1, e3> to parent
                      FLUSH E1's buffer
```

APPENDIX B
A DETAILED EXAMPLE

Original program

```
class STOCK : public REACTIVE
{
        private:
        ......
        public:
        ......
        event end(e1) int sell_stock(int qty);
        event begin(e2) && end(e3) void set_price(float price);
        int get_price();
        event e4 = e1 ^ e2; /* AND operator */
        /* class level rule */
        rule R1[e4, cond1, action1, CUMULATIVE, DEFERRED];
};

int STOCK::sell_stock(int qty) { ..... }
void STOCK::set_price(float price) { ..... }
int STOCK::get_price() { ..... }
/* Main program */
STOCK IBM, DEC, Microsoft;
```

```
main()
{
        .................
        /* Creating instance level primitive event */
        event set_IBM_price("set_IBM_price",IBM,
                                "begin","void set_price(float price)");
        /* SEQUENCE operator */
        event seq_event = STOCK_e4 << set_IBM_price;

        /* Creating a rule which contains both class level
        and instance level events */
        rule R2[seq_event, cond2, action2,,,20, PREVIOUS];
        ..................

        OpenOOBD->beginTransaction();
                IBM.set_price(115.00);
                DEC.set_price(100.00);
                Microsoft.sell_stock(200);
                DEC.get_price();
                IBM.set_price(75.95);
        OpenOODB->commitTransaction();
}
```

Preprocessed program

```
class STOCK : public REACTIVE
{
```

```
        private:
        ..........
        int user_sell_stock(int qty);
        void user_set_price(float price);
        public:
        ..........
        int sell_stock(int qty);
        void set_price(float price);
        int get_price();
};
int STOCK::sell_stock(int qty)
{
        int ret_value;
        /* Parameters are collected in a linked list */
        PARA_LIST *sell_stock_list = new PARA_LIST();
        sell_stock_list->insert("qty", INT, qty);

        /* The original sell stock method is invoked here */
        ret_value = user_sell_stock(qty);

        /* Notify end of method */
        Notify(this, "STOCK", "int sell_stock(int qty)",
                                "end",sell_stock_list);
        return(ret_value);
}
int STOCK::user_sell_stock(int qty)
```

```
{
    /* original sell_stock method */
}
void STOCK::set_price(float price)
{
    /* Parameters are collected in a linked list */
    PARA_LIST *set_price_list = new PARA_LIST();
    set_price_list->insert("price", FLOAT, price);

    /* Notify begin of method */
    Notify(this, "STOCK", "void set_price(float price)",
                          "begin",set_price_list);

    /* The original set price method is invoked here */
    user_set_price(price);

    /* Notify end of method */
    Notify(this, "STOCK", "void set_price(float price)",
                          "end", set_price_list);
}
int STOCK::user_set_price(float price)
{
    /* original set_price method */
}
int STOCK::get_price(char *n1) { ..... }
```

```
/* Main program */
STOCK IBM, DEC, Microsoft;
LOCAL_EVENT_DETECTOR *Event_detector

main()
{
    .....
    /* Creating the local event detector */
    Event_detector = new LOCAL_EVENT_DETECTOR();

    /* Creating primitive events */
    EVENT *STOCK_e1 = new PRIMITIVE("STOCK_e1", "STOCK",
                                    "end", "int sell_stock(int qty)");
    EVENT *STOCK_e2 = new PRIMITIVE("STOCK_e2", "STOCK",
                                    "begin", "void set_price(float price)");
    EVENT *STOCK_e3 = new PRIMITIVE("STOCK_e3", "STOCK",
                                    "end", "void set_price(float price)");

    /*Composite event AND */
    EVENT *STOCK_e4 = new AND(STOCK_e1, STOCK_e2);

    /* Creating Rule R1 */
    RULE *R1 = new RULE("R1", STOCK_e4, cond1, action1, CUMULATIVE);
    R1->set_mode(DEFERRED);

    /* Creating instance level primitive event */
```

```
    PRIMITIVE *set_IBM_price = new PRIMITIVE("set_IBM_price",
                                        IBM, "end", "void set_price(float price)");

    /* Composite event SEQUENCE */
    EVENT *seq_event = new SEQ(STOCK_e4, set_IBM_price);

    /* Creating Rule R2 */
    RULE *R2 = new RULE("R2", seq_event, cond2, action2, RECENT);
    R2->set_priority(20);
    R2->set_trigger_mode(PREVIOUS);

    OpenOODB->beginTransaction();
        IBM.set_price(115.00);
        DEC.set_price(100.00);
        Microsoft.sell_stock(200);
        DEC.get_price();
        IBM.set_price(75.95);
    OpenOODB->commit();
}
```

This example illustrates the wrapper methods introduced and conversion of application level event specification to system calls during preprocessing stage. It also illustrates the use of class level and instance level events/rules. Three class level primitive events, e1 as end-method event of sell_stock(), e2 as begin-method and e3 as end-method event of set_price() are declared. A class level composite event e4

is defined which is an AND of e1 and e2. A class level rule R1 is defined on event e4. Instance level primitive event *set_IBM_price* is defined for Stock object IBM. A composite sequence event is defined which is a combination of an instance level and class level event and finally rule R2 is defined on the sequence event(seq_event). Notice that after preprocessing the user defined methods 'sell_stock' and 'set_price' are renamed as 'user_sell_stock' and 'user_set_price' and wrapper methods 'sell_stock' and 'set_price' are introduced. As seen from the example appropriate code is introduced in the wrapper methods to notify the events. Also the application level rule and event specification are preprocessed to appropriate code for generation of event and rule objects along with the relevant parameters.

Regarding the detection of events Rule R2 will be fired first because it is in *immediate* mode with parameters {{DEC, price, FLOAT, 100.00}, {Microsoft, qty, INT, 200}, {IBM, price, FLOAT, 75.95}}. Rule R1 will be fired later since it is in *deferred* mode with parameters {{IBM, price, FLOAT, 115.00}, {DEC, price, FLOAT, 100.00}, {Microsoft, qty, INT, 200}}. Both DEC and IBM prices will be parameters to Rule R1 since its context is specified to be CUMULATIVE. Refer to [Cha93] for details on parameter computation for various contexts.

REFERENCES

[Anw93] 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.

[Arr92] J. A. Arroyo. The Design and Implementation of K.1 : A third-generation database programming language. Master's thesis, University of Florida, 1992.

[Bad93] R. Badani. Nested transactions for concurrent execution of rules: Design and implementation. Master's thesis, University of Florida, 1993.

[Cha89] S. Chakravarthy. Rule management and Evaluation: An Active DBMS Perspective. *Special issue of ACM Sigmod Record on rule processing in databases*, 18(3):20–28, 1989.

[Cha91] S. Chakravarthy. Active Database Management Systems: Requirements, State-Of-The-Art, and an Evaluation. In H. Kangassalo, editor, *Entity-Relationship Approach: The Core of Conceptual Modeling*, pages 461–473. Elsevier Science Publishers, North-Holland, 1991.

[Cha93] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K Kim. Anatomy of a Composite Event Detector. Technical Report UF-CIS-TR-93-039, University of Florida, E470-CSE, Gainesville, FL, December 1993. (Submitted for publication.).

[Cha94a] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data and Knowledge Engineering*, 1994. (To appear).

[Cha94b] S. Chakravarthy and D. Mishra. Towards an expressive event specification language for active databases. In *Proceedings of the 5th International Hong Kong Computer Society Database Workshop on Next generation Database Systems*, Kowloon Shangri-La, Hong Kong, February 1994. (Invited Paper).

[Dia91] O. Diaz, N. Paton, and P. Gray. Rule Management in Object-Oriented Databases: A Unified Approach. In *Proceedings, 17th International Conference on Very Large Data Bases*, Barcelona, September 1991.

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

[For87] C. L. Forgy and J. McDermott. Domain-Independent Production System Language. In *Proceedings, Fifth International Conference on Artificial Intelligence*, Cambridge, MA, 1987.

[Gat93] S. Gatziu and K. R. Dittrich. Events in an Object-Oriented Database System. In *Proceedings of the 1st International Conference on Rules in Database Systems*, September 1993.

[Gat94] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Databases using Petri nets. In *Proceedings of the 4th International Workshop on Research Issues in Data Engineering: Active Database Systems*, February 1994.

[Geh91] N. H. Gehani and H. V. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proceedings, 17th International Conference on Very Large Data Bases*, pages 327–336, Barcelona, September 1991.

[Geh92a] 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.

[Geh92b] 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.

[Hsu88] M. Hsu, R. Ladin, and D. McCarthy. An Execution Model for Active Data Base Management Systems. In *Proceedings, 3rd International Conference on Data and Knowledge Bases*, Washington, D.C., June 1988.

[Kot93] Angelika Kotz-Dittrich. Adding Active Functionality to an Object-Oriented Database System - a Layered Approach. In *Proceedings of the Conference on Database Systems in Office, Technique and Science*, Braunschweig, 1993.

[Mis91] D. Mishra. Snoop: An event specification language for active databases. Master's thesis, University of Florida, 1991.

[Sch91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive dbms into an active dbms. In *Proceedings, 17th International Conference on Very Large Data Bases*, pages 469–478, Barcelona, September 1991.

[Shy91] Y-M. Shyy and S. Y. W. Su. K: A High-Level Knowledge Base Programming Language. In *Proceedings, International Conference on Management of Data*, pages 29–31, Denver, CO, May 1991.

[Su89] S. Y. W. Su, V. Krishnamurthy, and H. Lam. An Object-oriented Semantic Association Model (OSAM*). In *Artificial Intelligence: Manufacturing Theory and Practice*, pages 464–494. The Institute of Industrial Engineers, Norcross, GA, 1989.

[TEX93] Texas Instruments. Open OODB Toolkit, Release 0.2 (Alpha) Document, September 1993.

[Vin93] S. Vinoski. Distributed object computing with corba. *C++ Report*, pages 33–38, July-August 1993.

[Wel92] D. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an Open Object-Oriented Database Management System. *IEEE Computer*, 25(10):74–81, 1992.

[Wid90] J. Widom and S. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proceedings of ACM-SIGMOD*, pages 259–270, May 1990.

BIOGRAPHICAL SKETCH

Vidhya Krishnaprasad was born on September 28, 1968, at Madras, India. She received her undergraduate degree in mathematics from University of Madras, India, in June 1989. She also received her graduate degree in computer applications from Anna University, India, in May 1992. She will receive her Master of Science degree in computer and information sciences from the University of Florida, Gainesville, in April 1994. Her research interests include active, object-oriented and heterogeneous databases, and implementation of graphical user interfaces.