SUPPORT FOR TEMPORAL EVENTS IN SENTINEL: DESIGN, IMPLEMENTATION, AND PREPROCESSING

By

HYESUN LEE

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

1996

Dedicated to my
Lord and Parents

ACKNOWLEDGMENTS

Foremost, I would like to thank my adviser, Dr. Sharma Chakravarthy, for his great guidance and support, and for giving me an opportunity to work on the challenging *Sentinel* project. I am extremely grateful to Dr. Eric Hanson 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 Hyoungjin Kim and Seung-Kyum Kim 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 family for their love. Without their encouragement and endurance, this work would not have been possible.

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

SUPPORT FOR TEMPORAL EVENTS IN SENTINEL:
DESIGN, IMPLEMENTATION, AND PREPROCESSING

By

Hyesun Lee

August 1996

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Science and Engineering

During the last decade, the functionality of databases has been extended to encompass a large range of applications which tend to be very complex in nature with concomitant increase in database size requirements. Database management systems (DBMSs) have evolved considerably to meet the requirements of emerging applications. Providing monitoring and reactive capabilities to existing conventional DBMSs without application or user intervention is one of the directions taken to deal with non-traditional real world applications. In contrast to a conventional DBMS which is considered as *passive or non-active*, the self-reacting DBMS is termed *active*. The *active* DBMS typically accepts ECA (event-condition-action) production rules which specify situations that need to be monitored and reacts to those situation under certain conditions. An *active* DBMS provides three new features to a passive DBMS: i) Rule Interface to define ECA rules, ii) Event Detect Mechanism to detect the events specified through the rule interface, and iii) Action execution mechanism to execute the action of the rule whose condition evaluates to true when the event occurs.

This thesis extends Sentinel by completing the local event detector to support temporal events, and by pre-processing the Snoop event/rule specification language to generate appropriate calls to the event detector. A few Snoop operators, such as A, A*, P and P*, are implemented as part of this effort to handle the temporal events. This thesis first describes how the local event detector works (e.g., construction of event graph and event notification mechanism) followed by the description of the temporal event handler and its integrated into the event detector. We also describe the Snoop preprocessor and its integration into the preprocessor of Open OODB (from Texas Instruments, Dallas), which is used as the underlying platform for Sentinel.

CHAPTER 1
INTRODUCTION

During the last decade, the functionality of databases has been extended to encompass a large range of applications which tend to be very complex in nature with concomitant increase in database size requirements. Conventional database management systems (DBMS) perform updates and execute queries using a *demand-based* mechanism, either when applications programs are executed or when interactive users perform some operation. A demand-based mechanism is inadequate for meeting the requirements of newer applications with a clear need for monitoring and reacting to pre-defined situations automatically without any user/application intervention. As an example, consider a process control environment where there is a need to continuously monitor the valve pressures to ensure that the pressure does not exceed a particular threshold. In such situations, it is desirable to remove the human element, as a human cannot always ensure timely response to critical situations, e.g., very high pressures in the valves. DBMSs having the added functionality of asynchronously monitoring situations and reacting to these situations are called *active* DBMSs (ADBMS). In contrast, conventional DBMSs are referred to as *passive* [4]. ADBMSs typically use ECA (event-condition-action) production rules to specify situations to be monitored and how to react to these situations. Briefly, an event is an indicator of a happening which can be simple or complex, a condition is a query based on either the existing database state, a set of objects, or transition between states of objects or even trends and historical data, and lastly, actions specify the operations to be performed when an event has occurred and the condition evaluates to true. ADBMS provides three new features to a DBMS:

- Rule Interface: This allows applications to define ECA rules.
- Event Detector: This is the entity which monitors applications as well as the database to detect the occurrence of primitive events. It is important to note, that once primitive events are detected, it is possible to group these primitive events thereby detecting complex/composite events.
- Action Execution: This module is responsible for scheduling and managing the execution of rule actions in accordance with ECA rule execution semantics.

Figure 1.1 demonstrates the major functional extensions an ADBMS provides to conventional passive DBMS to support active capability at the application level.

Most of the earlier work on ADBMS has concentrated on the support of active capability in the context of relational database systems. Recently, there have been a number of attempts at incorporating ECA support into an object-oriented database management system (OODBMS). Sentinel is an ADBMS based on OODBMS. Sentinel uses Snoop [16, 7] as its event/rule specification language and provides various parameter or event-consumption contexts for detecting composite events to meet a wide range of real-world application needs.

This thesis extends the earlier work on Sentinel [16, 7, 15] by adding the following: i) the stand-alone temporal event detector is integrated into the Sentinel local event detector, ii) relative temporal event, iii) several Snoop operators (A, A*, P and P*), and iv) a pre-processor for detecting Snoop event expressions and processing them appropriately. This thesis also describes the data structures and implementation details of the event detector.

The remainder of this thesis is structured as follows. Chapter 2 briefly describes the related work on active DBMSs. Chapter 3 summarizes Snoop, and Chapter 4 describes the architecture of Sentinel. Chapter 5 describes how the event detector

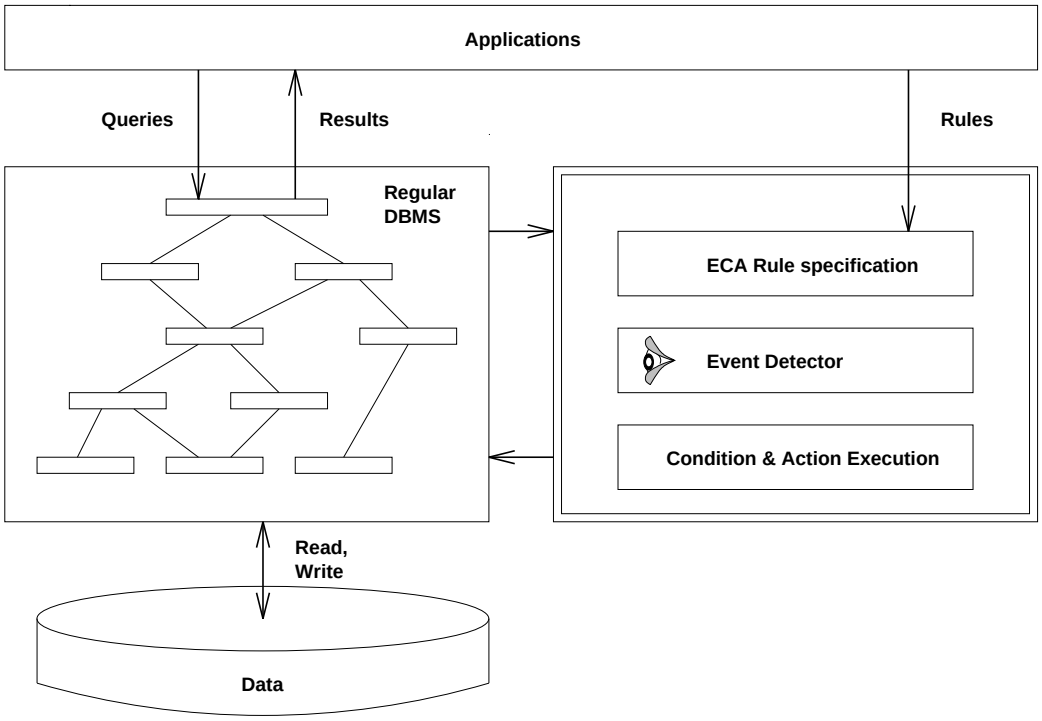


Figure 1.1. Major Extensions of a Regular DBMS to an ADBMS

works, and Chapter 6 discusses the Snoop preprocessor. Chapter 7 has conclusions and future work.

CHAPTER 2
RELATED WORK

In this chapter, a few active OODBMSs are examined to indicate how they provide the active functionalities.

2.1 Ode

Ode [14, 13] is a database system which is based on the object-oriented paradigm. The database is defined, queried and manipulated using the database programming language O++, which is an upward-compatible extension of the object-oriented programming language C++. Active behavior in Ode is presented by incorporating *constraints* and *triggers* [12] without the notion of ECA rules. Both constraints and triggers consist of a condition and an action. Constraints and triggers are defined declaratively within a class definition. Constraints are used to maintain object consistency which are applicable to *all* instances of a class as well as its subclasses. Triggers, on the other hand, are used for purposes other than object consistency and are applicable only to those instances of the class in which they are declared (along with its subclasses), specified *explicitly* by the user [12]. A trigger is activated on an object by an explicit invocation. A condition C_i is paired with an action A_i only, forming a constraint or trigger. Constraints and triggers are fired as a result of the invocation of *any* non-constant member function. Thus events in Ode are considered as the disjunction of all non-constant member functions. Events are generated as a result of the invocation of non-constant public member functions. Private and protected member functions do not generate events. All events signalled by an object of class A cause the evaluation of all constraints and triggers declared within class A. That is, event detection occurs via a method based mechanism: constraints and

triggers are precompiled into each place in the code where they might be activated, specifically, at the end of each non-constant public member function and before the commit of every transaction.

Ode [14, 13] has also proposed a language for specifying composite events. Basic (primitive) events are defined and composite events are constructed by applying operators to basic events. The basic events that are supported are *object state events* (creation, deletion, access, update, read), *method execution events* (before or after the execution of a method), *timed events* and *transaction events*. The event operators supported are *relative, prior, sequence, choose, every, fa* and *faAbs*. Basic events can be qualified with a *mask*, thus producing logical events. A mask is an optional predicate that allows users to specify more *complex* events. For instance, assume that the execution of the withdraw method constitutes a basic event.

Detection of composite events is accomplished by using finite automata. Each event expression maps an event history to another event history that contains only the events at which the event expression is satisfied and the trigger should fire [14, 13]. Each event expression has an automaton associated with it that reaches the acceptance state when the event is raised. Input to the automaton is the event history, the sequence of logical events, of the object with which the automaton is associated.

Events in Ode are treated as expressions declared within class definitions at compile time. This approach has several disadvantages. First, the treatment of events as expressions results in a dichotomy between events and other objects. Second, events cannot be created, deleted and modified dynamically. In addition, the introduction of new event types, attributes and operations requires major modifications, thus compromising the system's extensibility. The major disadvantage of this approach is the

inability to express complex events that are raised by occurrences of events in different classes. To clarify, Ode has adopted a local view of complex events; a complex event defined in class A can only be raised by events occurring in that same class A.

Ode supports two coupling modes[1], *immediate and deferred.*

2.2 ADAM

ADAM [11] is an active OODB implemented in PROLOG. It treats the rules uniformly as other objects, and rule operations are implemented as class methods. Rules are incorporated in ADAM by using an object-based mechanism. Basically, an object's definition is enlarged to indicate which rules to check when the object raises an event. Inheritance of rules from superclasses to subclasses is supported. However, the method by which inheritance is supported is specific to the PROLOG language and hence cannot be easily applied to other object-oriented programming languages. ADAM does not efficiently allow a rule to be applicable to only one instance of a class. This is accomplished by disabling the rule for all other instances. ADAM allows a rule's constituents to be modified dynamically. For example, it is possible to specify the condition and action parts of the rule at run time. Furthermore, the condition and action parts are defined dynamically rather than at compile time. The dynamic characteristics provided by ADAM are influenced by the interpretive environment in which ADAM is implemented and thus it is difficult to accomplish all of this in a language such as C++.

[1]The coupling mode denotes *when* the condition is to be evaluated with respect to the triggering transaction. HiPAC [3, 10] introduces three coupling modes, namely, immediate, deferred and detached. The immediate coupling mode specifies that a condition is to be evaluated *immediately* after the signalling of the triggering event. This mode causes the temporary suspension of the triggering transaction until the condition is evaluated. The second coupling mode, deferred, causes the condition to be evaluated at the *end* (before the commit) of the triggering transaction. The last coupling mode, detached, causes the condition to be evaluated in a *separate* transaction. This mode has two variations, specifically, causally dependent and causally independent. Causally dependent coupling mode implies a commit dependency between the triggering transaction and the rule, whereas causally independent implies that the rule is executed as a separate transaction independently of the triggering transaction.

Unlike Ode, ADAM adopts the ECA rule format. Events in ADAM are also treated as objects which are created, modified and deleted in the same fashion as other objects. An *event-class* is defined having three subclasses: *db-event, clock-event and application-event*. Each event is an instance of one of these three subclasses. Events in ADAM are basically generated either *before* or *after* the execution of a method. When an event is raised, all the methods' arguments are passed by the system to the condition and action part of the rule. Thus, the condition and action code may refer to the method's input or output parameters during evaluation. In order to create an event, the user must specify the *name of the method* generating the event and *when* the event should be raised. Although ADAM does not support complex events, the system is extensible enough to support them. This is due to their treatment of events as objects.

ADAM only supports the immediate coupling mode and does not support the other coupling modes proposed in HiPAC [3, 10].

2.3 Sentinel

Sentinel is an integrated active DBMS incorporating ECA rules using the Open OODB Toolkit (from Texas Instruments). Event and rule specifications are seamlessly incorporated into the C++ language. Any method of an object class is a potential primitive event. Furthermore, *before*- and *after*-variants of method invocation are allowed as events. Composite events are formed by applying a set of operators to primitive events and composite events. Events and rules are specified in a class definition. In addition, Sentinel supports events and rules which are applicable to a specific object instance. In that case, events and rules are specified outside of class definitions within the program where instance variables are declared. Supporting

this instance-level events and rules removes the significant drawback of ADAM's local view of composite events and allows composite events which are combinations of events from different classes or different applications (global events).

The parameters of a primitive event correspond to the parameters of the method declared as the primitive event and other attributes, such as the time of occurrence. The processing of a composite event entails not only its detection, but also the computation of the parameters associated with a composite event. The parameters of a composite event are collected, recorded and passed on to condition and action portions of a rule. Furthermore, these parameters are stored and reordered to meet various parameter contexts which are used to capture the application semantics. *Recent, Chronicle, Continuous,* and *Cumulative* parameter contexts are supported in Sentinel.

An event (primitive as well as composite) can trigger several rules, and rule actions may raise events which can trigger other rules. Sentinel supports multiple rule executions, nested rules executions as well as prioritized rule executions. The three coupling modes (immediate, deferred and detached) discussed in HiPAC[3, 10] were introduced to support application needs. Currently immediate and deferred modes are implemented.

CHAPTER 3
SUMMARY OF SNOOP

This chapter provides an overview [1] of the event specification language Snoop used in Sentinel for specifying ECA rules. Snoop uses an even hierarchy to classify events, defines the notion of events and event expressions, and describes a set of event operators [2] for constructing composite events.

3.1 Event, Event Expression, and Condition

An *event* is an atomic (happens completely or not at all) occurrence, and an *event expression* is an expression which defines an interval on the time line. A database transaction, operation, or a function can be regarded as an event expression since typically it takes a finite amount of time for its execution. Since an absolute point on the time line (which is corresponding to an absolute time) can be viewed as a degenerate event expression where a point is a special case of an interval, choosing a point that can be declared as an event corresponding to that event expression is necessary. Event modifiers [8] are used to transform an event expression to one or more events which correspond to various points of interest on the time line for that event expression. Snoop provides two event modifiers, *begin-of* and *end-of* to transform an arbitrary interval on the time line into an event.

A condition is a boolean function of data values, such as 'the salary of Jane is greater than 40K'. Evaluation of condition does not produce any side effects, i.e., change the database state. A condition may be valid over an interval of time. For example, 'the salary remains the same during the academic year' is an assertion that

[1] The implementation and syntax of Snoop [16, 7] are discussed in Chapter 5 and Chapter 6 respectively.

[2] The Snoop operators are discussed in Chapter 5 with implementation details

can be translated into an ECA rule. A database state of interest can be defined in terms of a condition; conversely, being in a state can be checked using a condition. Conditions define 'states' and hence are used in ECA rules as guards on transitions. A guarded transition fires when its event occurs but only if the guard condition is also true. For example, when one withdraws from an account (event), if the balance minus the amount being withdrawn (a parameter of the event) is below the minimum amount required (condition), then indicate the amount that can be withdrawn (action which may lead to another state). Conditions are likely to be formed on the parameters that are computed for a particular event instance (in addition to other shared data).

3.2 Event Classification

Events can be organized into a hierarchy of event classes. Each event class which has a unique event type [5] and instances of a class are identified by their class type and time of occurrence. For example, in a relational database, `end-of-delete` is an event class and each `delete` operation is an event instance of this class, which may have parameters such as the relation name and the tuples inserted in addition to other class parameters.

Events can be broadly classified as follows:

1. Primitive events : Events that are pre-defined in the system by using primitive event expressions and event modifiers. A mechanism for the detection is assumed to be available [1]

2. Composite events : Events that are formed by applying a set of operators [15] to primitive and composite events.

3.2.1 Primitive Events

Primitive events are further classified into database events, explicit events, and temporal events. Each event (primitive or otherwise) has a well-defined set of parameters that are instantiated for each occurrence of that event.

Database events are related to database operations such as a transaction, and in the relational model, they can be retrieve, insert, update and delete.

Explicit events are those events that are detected and signalled along with their parameters by application programs and are only managed by the system. Prior to their usage, explicit events and their formal parameters need to be registered with the system.

Temporal events are related to time and are of two types: absolute and relative. Absolute events map to discrete points on the time line (e.g., at 10 a.m.), whereas relative events are defined with respect to an explicit reference point (e.g., one hour after an event e occurs, where e is either a primitive or a composite event). An absolute time is composed of the following six fields: second (ss), minute (mm), hour (hh), day (DD), month (MM), and year (YY), and it is specified in the form of [hh:mm:ss/MM/DD/YY]. A relative time is a concatenation of one or more time units and has the form: [2 months + 10 days + 5 hrs + 10 mins + 49 secs]. Both absolute and relative can have wildcard ('?', or '*') in their expressions to represents multiple times.

3.2.2 Composite Events

A composite event expression is defined as an event expression formed by using a set of primitive event expressions, event operators (described in section 3.2.1), and composite event expressions constructed up to that point. A composite event is an event obtained by the application of an event modifier to a composite event expression. By default, the *end-of* event modifier is assumed.

The event classification is shown in Figure 3.1.

Figure 3.1. Event Classification

3.3 Snoop Event Operators

Snoop provides a number of operators for constructing composite events. Below, we summarize the operators supported in Snoop with brief explanations:

1. 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 (the order of occurrence of E_1 and E_2 is irrelevant).

2. OR (∇): Disjunction of two events E_1 and E_2, denoted $E_1 \nabla E_2$, occurs when either E_1 or E_2 occurs (simultaneous occurrence is currently excluded).

3. SEQ ($\gg$): Sequence of two events E_1 and E_2, denoted $E_1 \gg E_2$, 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.

4. 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 [14]) which detects the occurrence of any event other than E (the complement set semantics is used). It is rather similar to the SEQ operator except that E_2 should not occur between E_1 and E_3.

5. 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 occurrences.

6. A: One can express the occurrence of an aperiodic event in the half-open interval formed by E_1 and E_3[3]. An aperiodic event is denoted as A(E_1, E_2, E_3), where E_1, E_2 and E_3 are arbitrary events. The event A is signalled each time E_2 occurs during the half-open interval defined by E_1 and E_3.

7. A^*: This is a cumulative variant of A expressed as $A^*(E_1, E_2, E_3)$. It is useful when a given event is signalled more than once during a given interval, but rather than detecting the event and firing the rule every time the event

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

occurs, the rule has to be fired only once. A^* is detected when E_3 occurs and accumulates the occurrences of E_2 in the half-open interval formed by E_1 and E_3.

8. P: A periodic event is defined as an event E that repeats itself within a constant and finite amount of time. It is denoted as P(E_1, E_2, E_3), where E_1 and E_3 are any types of events and E_2 is a relative temporal event. P occurs for every amount of time specified with the time string of E_2 in the half-open interval (E_1, E_3]. The time string should be positive and should not have any wild card to prohibit continuous occurrences of P.

9. P^*: P^* is a cumulative variant of P and is denoted by $P^*(E_1, E_2, E_3)$. P^* occurs only once when E_3 occurs and accumulates the time of occurrences of the periodic event whenever E_2 occurs.

10. PLUS (+): Sequence of an event E_1 after a time interval TI, denoted E_1 + *[TI]* occurs when TI time units are elapsed after E_2 occurs.

3.4 Parameter Contexts

The notion of parameter contexts is introduced in Snoop to capture application semantics for computing the parameters or consuming event occurrences (of composite events) when they are not unique. Snoop identifies four parameter contexts that are useful for a wide range of applications. These contexts are precisely defined using the notion of initiator and terminator events. An initiator of a composite event is a constituent event which can start one detection of the composite event, and a terminator is a constituent event which can detect an occurrence of the composite event.

- **Recent:** In this context, *not all occurrences/instances* of a constituent event will be used in detecting a composite event, 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). 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 (after a detection, the initiator and the terminator are flushed). The oldest initiator is paired with the oldest terminator for each event (i.e., in chronological order of occurrence).

- **Continuous:** In this context, each initiator of an event starts the detection of that event and is saved until a terminator occurs. A terminator is paired with every initiator. Thus the terminator 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.

- **Cumulative:** In this context, all occurrences of an event type are accumulated as instances of that event until a terminator occurs (that is, the event is detected). Thus all the occurrences of the event detection are packaged in timely order. Whenever an event is detected, all the occurrences that are used for detecting that event are deleted.

CHAPTER 4
ARCHITECTURE

This chapter discusses the architecture of Sentinel, an active OODBMS being implemented at the UNiversity of Florida. Below, we briefly introduce the Open OODB and the Sentinel architectures.

4.1 Open OODB

The Open OODB project [19, 18], was initiated by Texas Instruments to build a high performance, multi-user object-oriented database management system (OODBMS) in which the database functionality can be tailored for the diverse needs of applications. The system provides an expandable framework that can also serve as a common testbed for research by database, framework, environment and system developers who intend to experiment with different system architectures or components. It facilitates incorporation of new components from smaller groups lacking resources to build an entire database system. The Open OODB describes the design space of OODB and builds an architectural framework that enables configuring independently useful modules to form an Object Oriented Database Management System. The Open OODB system architecture is divided into:

- A meta-architecture consisting of a collection of *kernel modules* and definitions providing the infrastructure for creating environments and boundaries, specifying and implementing event extensions and regularizing interfaces among modules.

- An extensible collection of *policy manager modules* which provide functionality to the system.

Since Open OODB is an object-oriented front end, it uses Exodus as its underlying storage manager through an interface. Open OODB supports multiple application programming languages (C++ and Lisp) and has extended these languages to support: persistence, concurrent transactions, and schema evolution to developers' existing programming environments. These extensions allow programmers to stay within familiar programming paradigms and languages. Open OODB allows developers to define a behavioral extension of events, which is an application of an operation to a particular set of objects. 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 known as a *sentry*. The primary function of *sentries* is to detect events interacting 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 used for specifying events to be extended and is responsible for deploying sentries to detect extended events.

Sentinel uses Open OODB as its platform. The primary class *OODB* has been extended to have reactive capability and the sentry mechanism is used to build wrapper functions wherever necessary.

4.2 Sentinel

Sentinel is based on the Open OODB [19, 18]. Sentinel's ECA rule support enhances the Open OODB from a passive OODB to an active one.

Figure 4.1 indicates the functional modules of the open OODB and the extensions for Sentinel. These extensions include:

- Primitive event detection: A method can be specified as a primitive event, and the occurrences of the primitive events are notified to the local event detector when the method is invoked. We have modified the Open OODB preprocessor

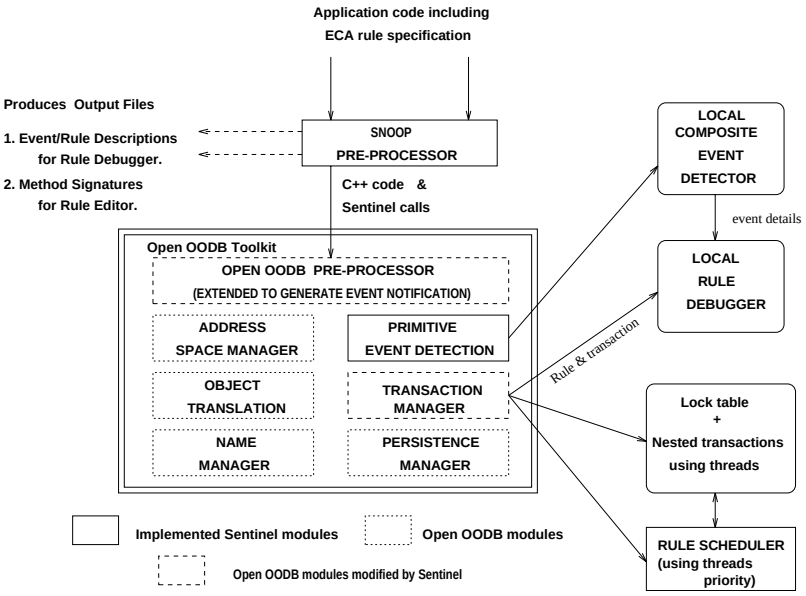


Figure 4.1. Sentinel Architecture

to wrap the method invocation with the notifications to the local event detector in order to detect primitive events.

- Composite event detection: Composite events defined within an application is detected by using a sequence of primitive events detected according to the specified parameter context of the composited event [15, 6]. Each Open OODB application has its own local event detector.

- Global event detection: The events of inter-applications is detected by the global event detector. The global event detector communicates with the local event detectors through RPC and socket-basec communication to detect global events (This is currently being implemented).

- Nested transactions: The transaction manager in the client address space supports nested transactions [17, 2] for concurrent execution of rules. Light weight processes are used both for prioritized and concurrent rule execution.

- Rule debugger: The rule debugger [20, 9] allows visualization of interactions among rules, rules and events, and rules and database objects.

- Rule editor: The rule editor allows the user to add external rules at run time. It communicates with the local event detector to edit rules and events. A graphical user interface for the rule editor is currently under development.

- Snoop preprocessor: The Snoop preprocessor transforms the ECA rules specified either as part of a class definition or as part of an application. The preprocessor converts the high-level user specification of ECA rules specified in Snoop language [7] into appropriate code for event detection, parameter computation, and rule execution.

Figure 4.2 shows how the class lattice of the Open OODB has been extended by Sentinel. The classes outside the dotted box have been introduced for providing active capability. This figure also shows the kernel-level enhancements to the Open OODB modules to accommodate nested subtransactions.

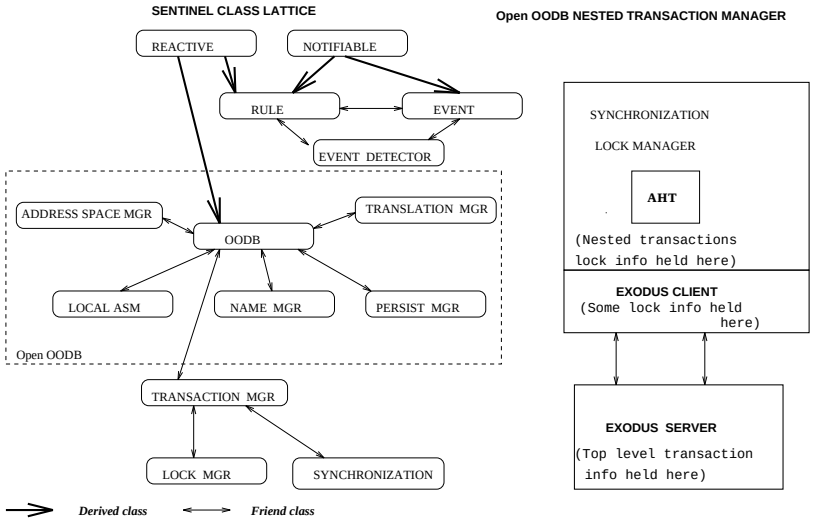


Figure 4.2. Class lattice and transaction manager of Sentinel

Figure 4.3 shows the control flow for supporting event detections and rule executions. The Sentinel primitive event detection mechanism is based on the design

proposed in [1]. The occurrences of the primitive events are signalled with the appropriate parameter collections by wrapping notifications of the primitive event method. Both primitive and local composite events are signalled as soon as they are detected.

Each application has a local event detector. 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. There is a clean separation between the events detected by the local event detector and the global event detector.

Figure 4.3. Local and Global Event Detector Architecture

CHAPTER 5
DESIGN AND IMPLEMENTATION OF LOCAL EVENT DETECTOR

Events are detected by a local event detector (LED) in Sentinel. In this chapter we discuss the implementation details of LED, including the constructing an event graph, handling of temporal events, and implementation of Snoop operators A, A^*, P, and P^*.

5.1 Event Graph

An event graph is a graph constructed to reflect the primitive and composite events declared in an application. Each event is represented as an event node in the graph, and the event nodes are connected by their subscription [1] relationships. An internal node of the event graph represents a composite event, and a leaf node represents a primitive event.

5.1.1 Primitive Event Node

A primitive event is a leaf node of the event graph and is a subscriber of a method. A primitive event node has four attributes: *method signature, event modifier, instance number*, and *object address*. *Method signature* stores the unique signature (return type, name and arguments of the method) of a method declared as a potential event. If a primitive event is a temporal event, it keeps the ascii time expression of the temporal event instead of a method signature. *Event modifier* (*begin* or *end*)[7] tells when the primitive event gets a notification from its associated method. It can be at the beginning or at the end of method invocation. The primitive event occurs at the notification time. Each occurrence of the primitive event has a unique instance

[1] An object can get notifications when methods of other object's are executed. We call this notification relationship *subscription*. The notifiable object becomes a subscriber of the notifying object.

number. If the primitive event is declared for a specific object of a reactive class, the attribute *object address* has the address of the specific object, otherwise it has a null pointer. The binding between a primitive event and a method is specified with an event modifier by a user in the Snoop language. In the example,

```
event begin (e1) int sell_stock(int number);
```

the primitive event *e1* is bound to a method named sell_stock and the method notifies its occurrence at the beginning of its invocation. Composite events and rules can subscribe any number of primitive events. These events and rules are kept either in *event-list* or in *rule-list* of the primitive event. Leaf nodes pass the primitive events immediately to their parent nodes as the semantics of all contexts are identical for primitive events.

5.1.2 Composite Event Node

A composite event is defined using one or more Snoop operators and primitive events using the BNF. The local event detector generates an event tree whose root node represents the composite event. Event trees in an application are merged to form an event graph for detecting a set of composite events. This prevents detecting common sub-events multiple times, thereby reducing storage requirements. A sequence of constituent event occurrences makes a composite event occur. Since event occurrences happen over a period of time, it is necessary to store the occurrence of each event and save its parameter lists to detect composite event occurrences. Each operator node which is the root node of an event tree has storage to save these event occurrences. The information is saved separately according to the parameter contexts which are applicable to composite events. Multiple contexts can be set to an operator node, and these contexts do not interfere with each other while the composite event is detected. The stored information is flushed or remains after an occurrence of the composite event according to parameter context. Similar to a primitive event,

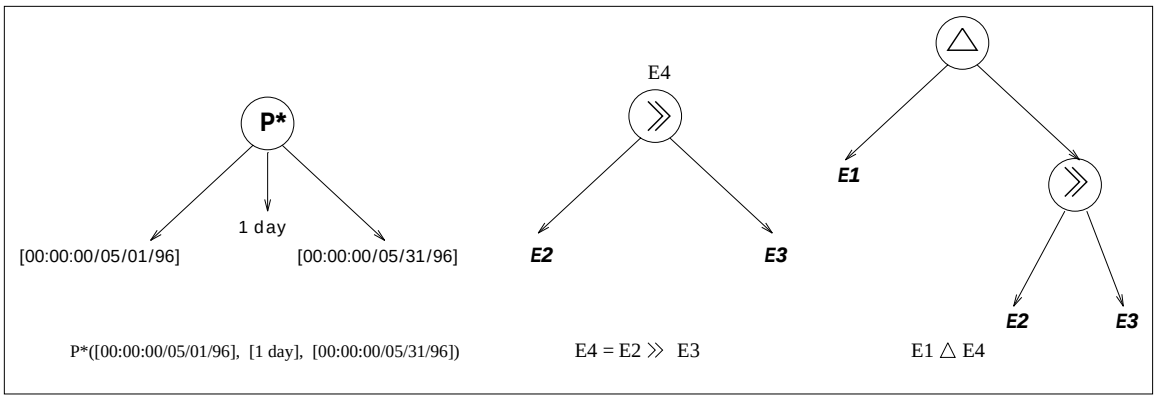


Figure 5.1. Event trees

a composite event (node) has two linked lists, one for its event subscribers and the other for its rule subscribers. Figure 5.1 shows a few event trees.

5.1.3 Local Event Detector

An event detector is implemented as a class, and we have a single instance of this class per application (termed local event detector). A local event detector has a linked list whose nodes hold one reactive class of an application. Each node, in turn, has two linked lists, *begin_list* and *end_list*. The lists have the subscribers (here, they are primitive events which are bound to one of methods in the class) to be notified at the beginning or the end of these methods's invocations. By default a subscriber is inserted in the end-list if it does not specify when to be notified. This organization reduces the search which is based on the class. However, search for the class is sequential. An event graph is constructed while the event trees of composite events in the application are built by their subscribing relations. The event graph is connected to the reactive class list through primitive-leaf nodes. A primitive-leaf node is pointed by a method which is belong to one of the reactive class nodes. This connected event graph and the reactive list constitute a local event detector for the application. A primitive event constructor registers itself to the reactive class list as a subscriber of a method in the list. Figure 5.2 shows how primitive leaf nodes

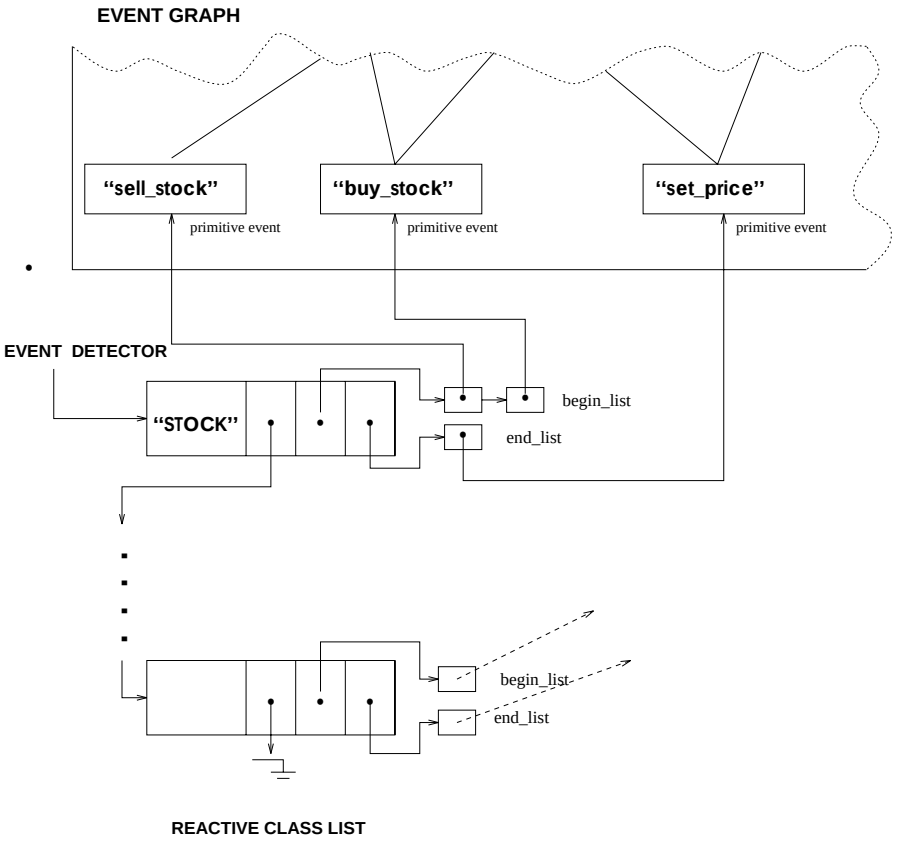


Figure 5.2. An Event Graph connected to a reactive class list

of an event graph are connected to a reactive class list. The reactive class name is "STOCK" and it has three methods: "sell_stock", "buy_stock",and "set_price". Figure 5.3 shows an event detector structure.

5.1.4 Detecting Events

The methods that can generate primitive events are modified by the wrapper class methods using the sentry feature of the Open OODB system while preprocessing the application program. The Open OODB preprocessor was modified by Sentinel to add code for parameter collection and notification to the event detector while preprocessing the application program. When a method registered to the local event detector is invoked, notifications of this invocation are signalled both at the beginning and at end of the method to the local event detector with method signature, class name,

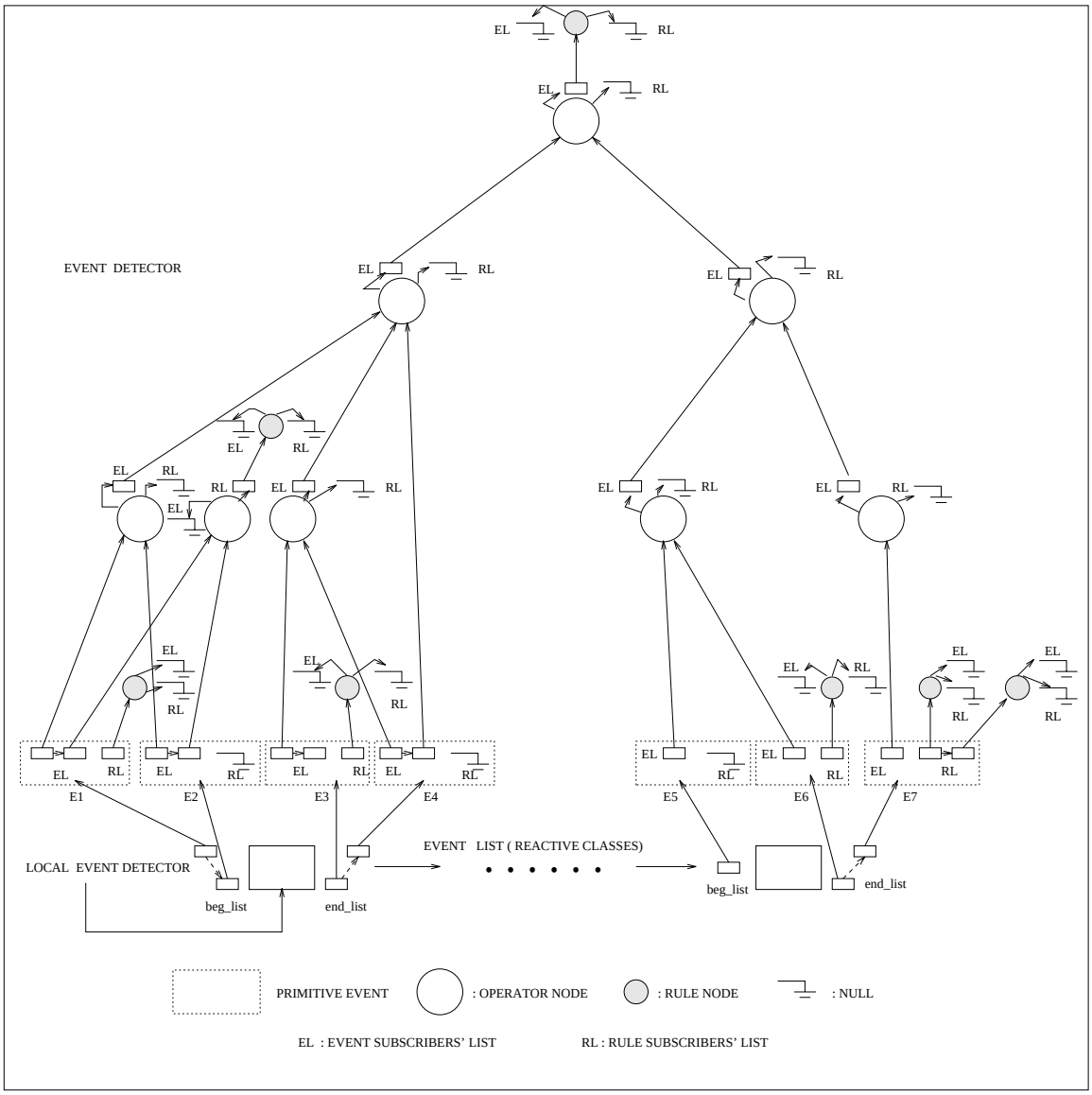


Figure 5.3. An Event Detector

event modifier, parameter list of the method, and a reactive object if this event is declared as an instance-level event. A parameter list is formed right before notification with the parameters of the method. The parameter list keeps names, types, and values of the parameters. With the class name, the local event detector searches the list of the reactive classes. If there exists no node with the same name, it causes an event-raising error. If there is a node with the same name, then it traverses either begin-list or end-list according to the event modifier that came with the notification. Up to this point, every primitive event in any one of the lists is notified regardless of the event method to which the primitive event is bound. A primitive event node keeps the method signature and the reactive object if any. If the notification that came from the event method has a reactive object, the primitive event node compares these two reactive objects as well as their method signatures. Blank spaces are ignored when the comparison takes place. Only when both are matched does it propagate its occurrence to its subscribers, which can be any composite events or rules. Every time the primitive event occurs, its unique instance number is given, its occurrence time is set to the parameter list, and further notifications go to its subscribers. As defined earlier, a composite event is an event tree and the root of the event tree is one of the Snoop operators. The node maintains the occurrence of its constituent event occurrences with their parameter lists which are stored separately for each context set to the node. Whenever it is notified by one of its constituents, the node checks the status of its constituents' occurrences. We call the constituent event instance that causes the start of an occurrence of the composite event an *initiator*, and the constituent event instance that causes the detection of the occurrence a *terminator*. If the composite event occurs by the last notification, it is detected, and further notifications are sent to its subscribers. The parameter list is recomputed to hold event traces from method invocation to this new occurrence and gets the

current time as a new occurrence time. After the detection and the notifications, the parameter list held in the operator storage will be flushed or maintained for the next detection of an event according to the operator semantics. Any events or rules can unsubscribe their constituent events even after the event graph is constructed. This unsubscription does not change the event graph; only deletes the events or rules from the subscriber lists. Resubscribing is as easy as unsubscribing: just insert the new subscriber in the list. No reshaping of the event graph is requested.

5.2 Temporal Event

5.2.1 Handling Temporal Event

Unlike the other types of primitive events, temporal events need a temporal event handler as well as a local event detector. One temporal event handler is generated per application. The temporal event handler consists of classes, *Time_queue_item, Time_queue, and Time_queue_handler. Time_queue_item* [21] is responsible for converting a time expression into a numerical representation. Time_queue handles these numerical temporal representations in timely order and Time_queue_handler provides an interface between a local event detector and temporal event handler and manages time queues and the timer. Two *Time_queues* are created when *Time_queue_handler* is constructed.

Converting a timestring into a numeric time expression

A temporal event is created by giving a unique time expression and its identification number. *add_item*, one of the functions of *Time_queue_handler*, carries the time expression and the identification number to the temporal event handler. Examples of declaring temporal events are below:

```
Time_queue_handler tqh;
...
tqh.add_item(``12:00:00/04/18/96'', evnt_id ); /* absolute event */
```

```
tqh.add_item(``3 hrs 30 mins'', evnt_id ); /* relative event */
```

add_item creates an instance of *Time_queue_item* with the time string and the identification number. The time string is converted into a numerical time expression. This conversion is directed by the function called *conv_ts* if it is an absolute temporal string, or by the function called *conv_rel* if it is a relative temporal string. Once the time string is broken into time units from second to year, these broken character values are converted into integer values. If a time unit includes a wildcard, the wildcard is replaced with the lowest value according to its position when the conversion takes place. The current time is added to the converted values if it is a relative case. The results of the conversion are held in the array *tm_hold* where each element represents a time unit. These broken integer values are copied to a *tm* structure which is declared in <*time.h*> for representing real time, and the values in the *tm* are converted into a time value that represents the number of seconds since Jan. 1, 1970, 00:00, *Greenwich Mean Time*. This final converted result is stored into the *conv_time* data member, which is a type of *time_t* (long integer).

Managing Time Queues

The temporal event handler has two time queues–*timeq* and *pres_items*– generated when the handler is created. A time queue is semaphore-protected; that is, the time queue constructor requests a semaphore for the queue from the operating system, and the semaphore is used to serialize additions to and deletions from the queue. The *timeq* is an ordered queue of time queue items. Time queue items are queued in timely order before they are set to the timer. If the item has an obsolete time, it is not queued. The *pres_items* queue is used to handle multiple temporal events which have the same expiration times. The front item of timeq is moved into the pres_items before its time is set to the timer and stays there until the time expires: the item in pres_items is currently set to the timer and dequeued when the time expires. Multiple

items with the same expiration time are moved together into pres_items. If a new item is queued into timeq with an earlier time than that of the item in the pres_items, then they are swapped and the timer is reset with the earlier time.

Setting a Timer

The time of the item in *pres_items* is set to the timer. Even though there is more than one item with the same time in the queue, the time is set to the timer only once. If the time becomes obsolete while it stays in the timeq, the item is not set and the next time is set to the timer. Setting the timer is done by calling the system function *setitimer*.

Processing of Alarm Clock Signal

When the temporal event handler is created, *Time_queue_handler* constructor installs a signal handler to give an alarm clock signal by calling a system function, *signal*. A signal is sent when the time on the timer expires to the signal handler of the temporal event handler. The signal handler calls *process*, which is a method of *Time_queue_handler*. This function dequeues all items in the *pres_items* and gives them notifications with the expiration time. Repetitive items are updated with the next higher value and queued into *timeq* unless updating is no longer necessary. The new front item in *timeq* is moved into *pres_items* and is set to the timer. Figure 5.4 shows how a time item is handled inside of the temporal event handler.

5.2.2 Integration of a Temporal Event Handler into a Local Event Detector

As other types of primitive events, temporal events are detected through an event graph. The temporal event handler and the event graph is connected through primitive temporal events. When the first temporal event is declared, a reactive class node is inserted into the reactive class list of the local event detector with the name *"TEMPORAL"* instead of a reactive class name. A primitive event is declared with

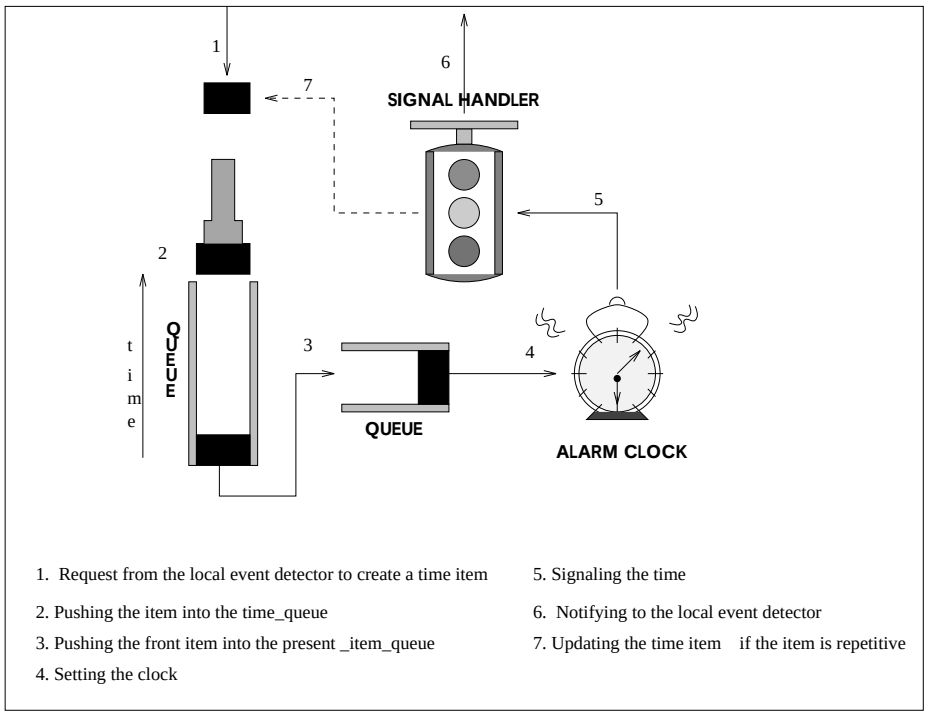


Figure 5.4. Temporal Event Handling

a time expression and is listed in a subscribers' list as it is with a method signature. Since the time expression is used like a method, the time expression should be unique. The *end-of* event modifier is chosen for temporal event by default. Figure 5.5 shows how temporal events are integrated into a local event detector (this figure only shows the reactive class list and primitive events of the local event detector). Once declared, a primitive temporal event checks the type of time expression. If it is absolute, a *time queue item* is created with the time string and pushed into the *time queue*. If it is relative, the *time queue item* is created when its relative event occurs rather than by the primitive event. Relative time expression is used in *P, P*, and PLUS* operators. Since a relative time expression has to know which is its relative event, a time queue item is created with the event number of the relative event as well as the time string while an absolute time expression has *0* as its event number. When the timer signals that a time is finished on the timer, the temporal

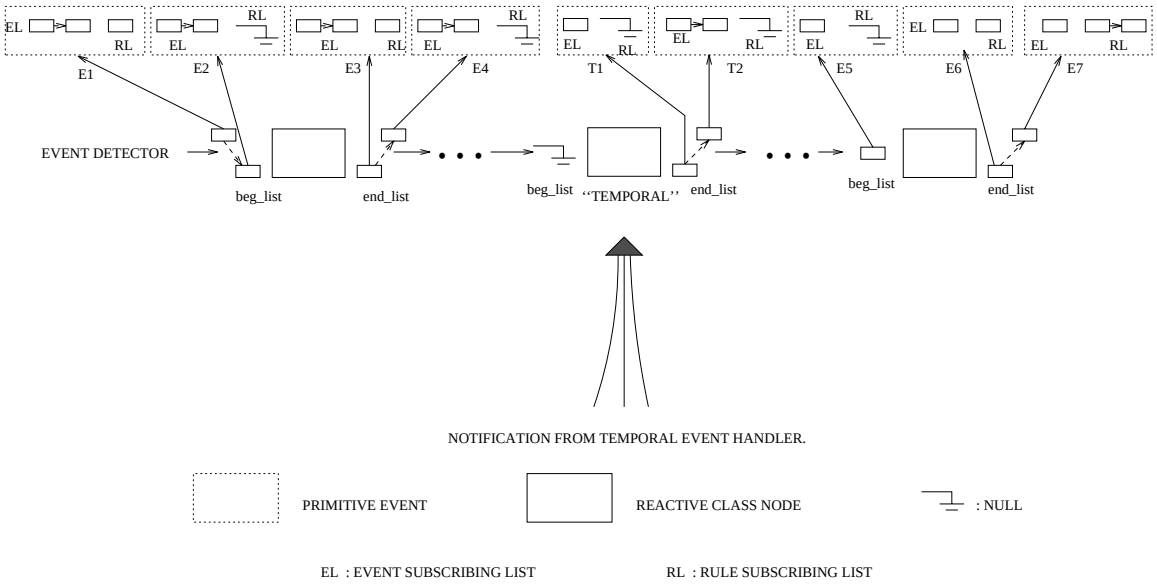


Figure 5.5. Integration of Temporal Event into a Local Event Detector

event handler catches the signal and notifies the occurrence to every time queue item in *pres_items* with the occurrence time. A time queue item creates a parameter list which holds the occurrence time and its event number that were given from its referencing primitive event. The event number is added to the end of the parameter list. The notifications go to the leaf nodes of the event graph. The primitive leaf nodes have these time expressions instead of method signatures. Further notification goes up the event graph through the events that subscribe the temporal event. Temporal event notification from the temporal event handler to the event graph is sent like a method connected to a primitive event except for a few more steps to reach the event graph.

5.3 Newly Implemented Snoop Operators

This section describes the implementation details of PLUS, A, A^*, P, and P^* with their algorithms.

1. PLUS (+): PLUS is used with relative time expressions. A PLUS operator subscribes two events E_1 and E_2, where E_1 can be any types of events and

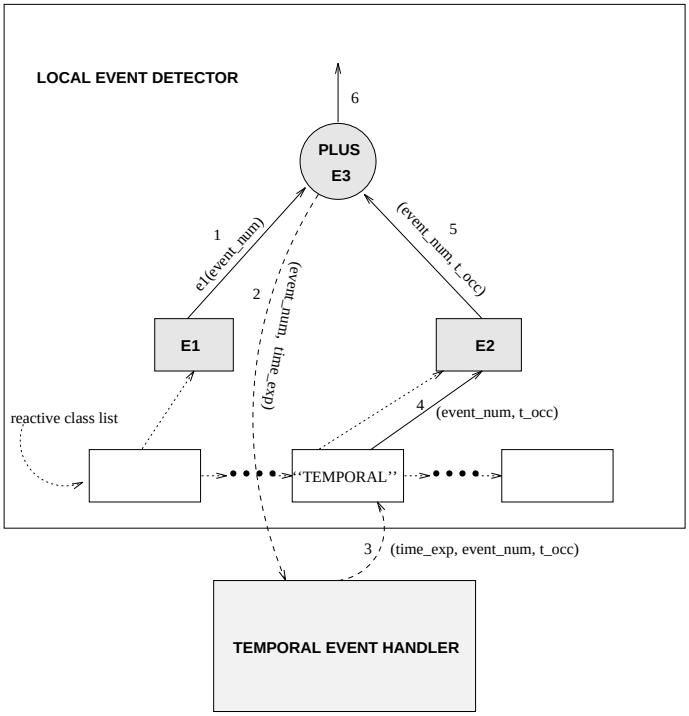
E_2 should be a relative temporal event. Originally E_2 is specified as a relative time expression from a user. The Snoop preprocessor declares it as a relative temporal event. A relative temporal event occurs after the amount of time expressed by the time string since an instance of E_1 occurs. Unlike absolute temporal events which produce *time queue items* when they are declared as primitive events by the Snoop preprocessor, relative temporal events produce *time queue items* when an E_1 instance occurs. Each *time queue item* carries an identification number which is the identification number of its referencing E_1, and passes this number to E_2. When the *PLUS* operator gets the occurrence notification from E_2, it checks the identification number of the instance of E_2 to verify that the matching E_1 instance is still valid. The identification number is carried in the last parameter of the parameter list brought with the notification. Any E_2 occurrence whose identification number cannot be found in the list of E_1 is ignored. The following Figure 5.6 shows how the PLUS operator communicates with the temporal event handler to handle its relative temporal event.

The algorithm of detecting relative temporal events follows where ei represents an instance of E_i.

PLUS(E_1, E_2):

```
PROCEDURE plus_recent( Ei, parameter_list) /* Recent */
    if left event e1 is signalled
        get the terminator ID of e1
        create a time queue item with the ID and the time string of E2
        replace e1 in E1's list.

    if right event e2 is signalled
        get the ID of e2
        if E1's list is not empty and the ID of e1 and that of e2 are the same
            pass <e1, e2> to the parent with a new t_occ
```



1. E1 occurs. The instance number of E1 is passed to E3.
2. Ask temporal event handler to produce a time queue item with the instance number of E1 and the time expression of E2
3. Notify that the time has passed. Event number of E1, time expression, and the occurrence time are passed.
4. Notify to E2.
5. E2 occurs. The instance number of E1 and the occurrence time are passed in the parameter of E2.
6. Check the instance numbers from E1 and E2. If they are matched, notify the occurrence of E3 to the subscibers.

Figure 5.6. PLUS Operator

The local event detector allows multiple parameter context without any interference to other context detections. To make this no-interference gracefully, a PLUS operator has to check which other contexts are set. The priorities are given in the order of *recent, chronicle, continuous*, and *cumulative*. By this priority list, only the *recent* context does not need to check the other contexts set. *Chronicle* context checks whether *recent* context is set; only when *recent* is not set does the *chronicle* context produce a *time queue item*. *Continuous* context checks whether *recent* or *chronicle* context is set. If either one of them is set, the *continuous* context cannot produce a *time queue item*. In the *cumulative* case, it should check the other three contexts to produce a *time queue item*. Without context checking, multiple *time queue items* are produced for the same instance of E_1 and send unnecessary notifications from the *time queue handler* to the *event graph*, thus increasing traffic bandwidth unnecessarily. Note that the validating identification number of a temporal event and checking parameter contexts should be done for P and P* as PLUS.

PROCEDURE plus_chronicle(E_i, parameter_list) /* Chronicle */

```
if left event e1 is signalled
    get the terminator ID of e1
    if Recent context is not set
        create a time queue item with the ID and the time string of E2
    append e1 in E1's list.

if right event e2 is signalled
    get the ID of e2
    if E1's list is not empty and the head's terminator ID
    and that of e2 are same
        pass <E1's head, e2> to the parent with a new t_occ
```

If a Snoop operator is set to *continuous* context, a terminator may detect one or more occurrences of the same event. In a PLUS operator, however, the terminator, which is a relative temporal event, is only matched with the initiator

which gave birth to the *time queue item* of the terminator. Since the order of relative temporal events is the same as their referencing E_1 instances, detections of the *continuous* context are the same as detection of *chronicle* context.

PROCEDURE plus_continuous(E_i, parameter_list) /* Continuous * /

if left event e1 is signalled
get the terminator ID of e1
if *Recent* or *Chronicle* context is not set
create a *time queue item* with the ID and the time string of E_2
append e1 in E_1's list.

if right event e2 is signalled
get the ID of e2
if E_1's list is not empty and the terminator ID of the head of the list and that of e2 are same
pass <E_1's head, e2> to the parent with a new t_occ
delete the head of E_1's list

Note that in implementation, instead of pairing the head of E_1's list and e2, the e1 whose terminator's ID is the same as that of the e2 is searched in the E_1's list in case of losing any alarm clock signal from the signal handler. Unmatched e1s are deleted.

PROCEDURE plus_cumulative(E_i, parameter_list) /* Cumulative * /

if left event e1 is signalled
get the terminator ID of e1
if *Recent*,*Chronicle*, or *Continuous* context is not set
create a *time queue item* with the ID and the time string of E_2
append e1 in E_1's list.

if right event e2 is signalled
get the ID of e2
if E_1's list is not empty
pass <all e1's, e2> to the parent with a new t_occ
flush E_1's buffer

2. A: We can express an aperiodic event with the A operator. The aperiodic event occurs whenever the second event E_2 occurs during the interval defined

by the first and the third events. 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).

A(E_1, E_2, E_3) :

PROCEDURE a_recent(E_i, parameter_list) /* Recent */

```
if left event e1 is signalled
    replace e1 in E1's list.

if middle event e2 is signalled
    if E1's list is not empty
        pass <e1, e2> to the parent with a new t_occ

if right event e3 is signalled
    flush E1's buffer
```

PROCEDURE a_chronicle(E_i, parameter_list) /* Chronicle */

```
if left event e1 is signalled
    append e1 in E1's list.

if middle event e2 is signalled
    if E1's list is not empty
        pass <E1's head, e2> to the parent with a new t_occ
        delete the head of E1's list

if right event e3 is signalled
    flush E1's buffer
```

From the above chronicle pseudocode, we can notice that if the previous E_2 instance makes E_1's list empty, a new occurrence of E_2 will be ignored even though it occurs during the interval. The same situation happens in the *continuous* context.

PROCEDURE a_continuous(E_i, parameter_list) /* Continuous * /

```
if left event e1 is signalled
    append e1 to E1's list
```

```
if middle event e2 is signalled
    if E1's list is not empty
        for every instance e1 in E1's list
            pass <E1's head, e2> to the parent with a new t_occ
            delete E1's head

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

PROCEDURE a_cumulative(E_i, parameter_list) /* Cumulative */

```
if left event e1 is signalled
    append e1 to E1's list

if middle event e2 is signalled
    if E1's list is not empty
        pass <all e1s, e2> to the parent with a new t_occ
        flush E1's buffer

if right event e3 is signalled
    flush E1's buffer
```

In all the contexts in A, the first occurrence of E_3 after the interval has began terminates the detection of A. A new instance of E_1 is necessary to reinitiate the detection.

3. A^*: A^* is a cumulative variant of A. A^* is detected only once when E_3 occurs instead of being detected every time E_2 occurs. When E_2 occurs, the occurrence time of the instance and the values of the parameters which E_2 carries are accumulated in E_2's list. These accumulations of E_2 instances require separate storage from E_1's list storage.

   In recent context, an occurrence of E_1 flushes E_2's list since the new occurrence of E_1 starts another recent interval.

   $A^*(E_1, E_2, E_3)$:

   PROCEDURE astar_recent(E_i, parameter_list) /* Recent */

```
if left event e1 is signalled
    replace e1 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 E1's list is not empty
        if E2's list is not empty
            pass <e1, all e2s, e3> to the parent with a new t_occ
            flush E1 and E2's buffers
        else
            pass <e1, e3> to the parent with a new t_occ
            flush E1's buffer
```

Unlike all contexts of A or recent context of A^*, in the chronicle context an occurrence of E_3 does not delete all previous E_2 occurrences. The ith occurrence of E_1 and the ith occurrence of E_3 are paired with all E_2 occurrences between them.

PROCEDURE astar_chronicle(E_i, parameter_list) /* Chronicle */

```
if left event e1 is signalled
    append e1 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 E1's list is not empty
        if E2's list is not empty
            pass <E1's head, all e2's whose t_occ are greater than
            that of E1's head, e3> to the parent with a new t_occ
            delete the head of E1's list
            delete all e2s in E2's list whose t_occ is less than that
            of E1's new head
            if E1's list is empty
                flush E2's buffer
        else
            pass <E1's head,e3> to the parent with a new t_occ
            delete head of E1's buffer
```

In the continuous context, the first occurrence of E_3 flushes the lists of E_1 and E_2 like recent context.

```
PROCEDURE astar_continuous(E_i, parameter_list) /* Continuous */

    if left event e1 is signalled
        append e1 to E_1's list

    if middle event e2 is signalled
        if E_1's list is not empty
            append e2 to E_2's list

    if right event e3 is signalled
        if E_1's list is not empty
            if E_2's list is not empty
                for every instance e1 in E_1's list
                    pass <e1, all e2 whose t_occ is greater than that of
                    e1,e3> to the parent with a new t_occ
                flush E_1's and E_2's buffers
            else
                for each e1 in E_1's list
                    pass < e1, e3> to the parent with a new t_occ
                flush E_1's buffer
```

In cumulative context, we use only one storage for both E_1 and E_2 parameters since, when E_3 occurs the first time, all of the stored parameters will be sent in their occurrence order.

```
PROCEDURE astar_cumulative(E_i, parameter_list) /* Cumulative */

    if left event e1 is signalled
        append e1 to E_1 and E_2's common list

    if middle event e2 is signalled
        if E_1's list is not empty
            append e2 in E_1 and E_2's common list

    if right event e3 is signalled
        if E_1's list is not empty
            pass <all e1 and all e2 in the common list, e3> to the parent
            with a new t_occ
        flush E_1 and E_2's common buffer
```

4. P: P repeats itself within a constant and finite amount of time expressed by E_2. P occurs in every E_2 amount of time (that is, when E_2 occurs). It is important to note that the time expression is a constant and preferably does not contain wild card specifications in all fields because this will result in continuous occurrences of P. Instances of E_2 play the role of terminators. Except for the first occurrence of P, the previous occurrence of E_2 is the initiator of the current occurrence of P. An occurrence of E_1 can be an initiator only once for the first occurrence of E_2, which has the time queue item requested by the E_1 occurrence. Thus, E_2 can be an initiator and a terminator. Checking parameter context to avoid unnecessary multiples of *time queue item* is done like a PLUS operator.

   In recent context, a P event can be repeated until E_3 occurs by requesting creating a *time queue item* with the instance number of the recent occurrence of E_1 and the relative time expression of E_2 to the temporal event handler.

   P(E_1, E_2) :

   PROCEDURE p_recent(E_i, parameter_list) /* Recent */

   ```
   if left event e1 is signalled
       get the terminator ID of e1
       create a time queue item with the ID and the time string of E2
       replace e1 in E1's list.

   if middle event e2 is signalled
       get the ID of e2
       if E1's list is not empty and the ID of e1 and that of e2 are same
           create a time queue item with the ID of terminator ID of e1
           pass <e1, e2> to the parent with a new t_occ

   if right event e3 is signalled
       flush E1's buffer
   ```

   In chronicle context, initiators and the matching terminator should be removed after they are used. The first occurrence of E_2 is used as a terminator, and

it cannot be used as the next initiator. Thus, the P repeats only once in a half-open interval.

PROCEDURE p_chronicle(E_i, parameter_list) /* Chronicle */

```
if left event e1 is signalled
    get the terminator ID of e1
    if Recent context is not set
        create a time queue item with the ID and the time string of E2
    append e1 in E1's list.

if middle event e2 is signalled
    get the ID of e2
    if E1's list is not empty and the terminator ID of the head of the
    list and that of e2 are same
        pass <E1's head, e2> to the parent with a new t_occ

if right event e3 is signalled
    flush E1's buffer
```

Since the initiators of continuous and cumulative contexts should be removed after they are matched with terminators, these two contexts are also repeated only once. And the instances of E_2 have to be matched only with the instances of E_1 which caused the occurrence of E_2. For these two reasons, the *continuous* and *cumulative* contexts of a P event are detected like the *chronicle* context. That is, an instance of E_1 is matched with an instance of E_2 to detect a P event only once in time order.

PROCEDURE p_continuous(E_i, parameter_list) /* Continuous */

```
if left event e1 is signalled
    get the terminator ID of e1
    if Recent or Chronicle context is not set
        create a time queue item with the ID and the time string of E2
    append e1 in E1's list.

if middle event e2 is signalled
    get the ID of e2
    if E1's list is not empty and the terminator ID of the head of the
    list and that of e2 are same
        pass <E1's head, e2> to the parent with a new t_occ
```

if right event e3 is signalled
flush E_1's buffer

PROCEDURE p_cumulative(E_i, parameter_list) /* Cumulative */

if left event e1 is signalled
get the terminator ID of e1
if *Recent*,*Chronicle*, or *Cumulative* context is not set
create a *time queue item* with the ID and the time string of E_2
append e1 in E_1's list.

if middle event e2 is signalled
get the ID of e2
if E_1's list is not empty and the terminator ID of the head of the list and that of e2 are same
pass <E_1's head, e2> to the parent with a new t_occ

if right event e3 is signalled
flush E_1's buffer

Figure 5.7 illustrates the detection of P event, P(E_1, E_2, E_3) in various contexts.

5. P^*: P^* is a cumulative variant of P. P^* occurs only once when E_3 occurs and accumulates the time of occurrences of the periodic event whenever E_2 occurs. Unlike P, an instance of E_2 cannot be either an initiator or a terminator. Only an E_1 occurrence can be an initiator and an E_3 occurrence can be a terminator. In the implementation, however, an occurrence of E_2 requests the temporal event handler to create the next time queue item without defecting this operator's semantics.

P^*(E_1, E_2, E_3):

PROCEDURE pstar_recent(E_i, parameter_list) /* Recent */

if left event e1 is signalled
get the terminator ID of e1
create a *time queue item* with the ID and the time string of E_2
replace e1 in E_1's list.

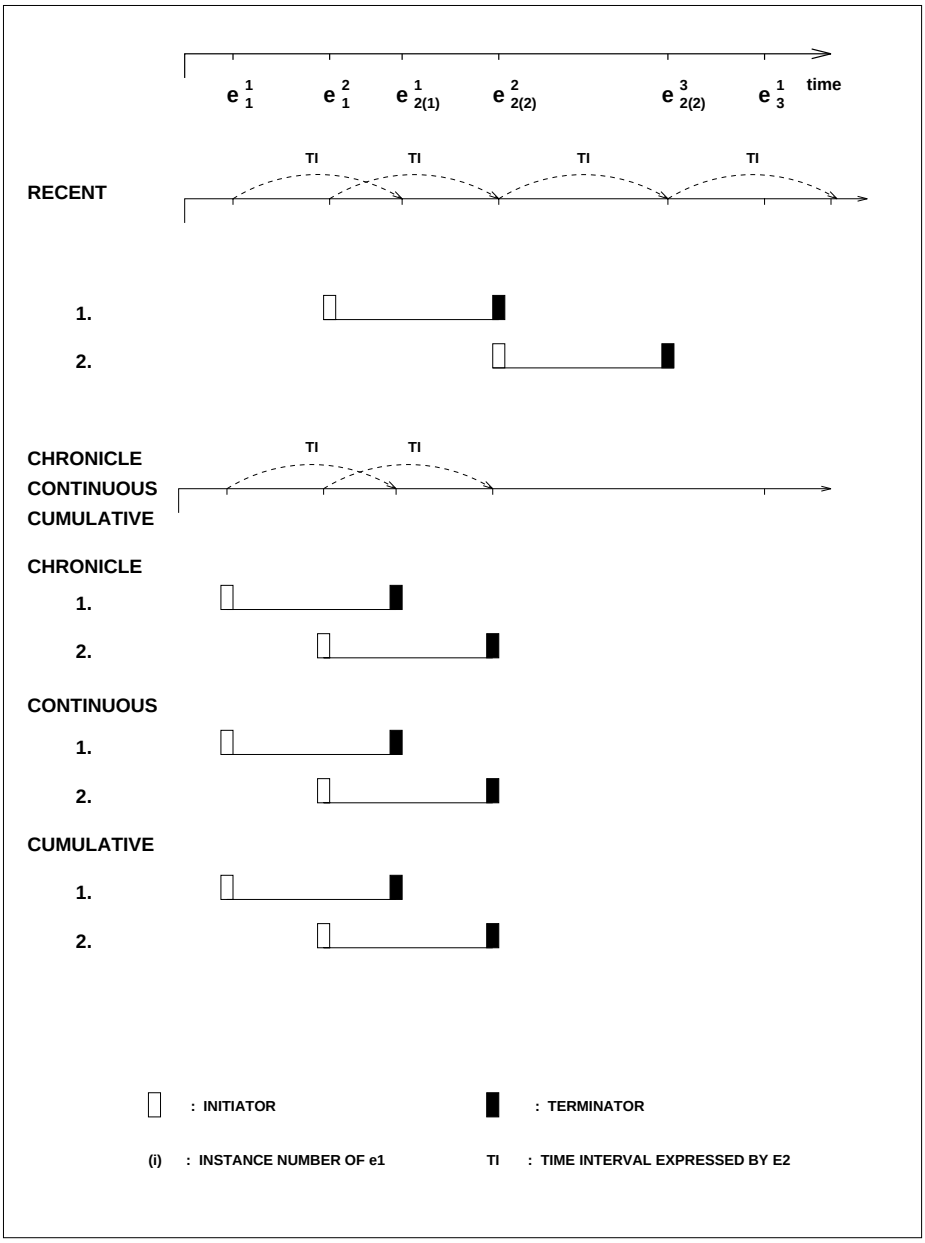


Figure 5.7. Illustration of P(E1, E2, E3) event detection in various contexts, where E2 is a time expression.

```
if middle event e2 is signalled
    get the ID of e2
    if E1's list is not empty and the ID of e1 and that of e2 are same
        create a time queue item with the ID and the time string of E2
        append e2 to E2's list

if right event e3 is signalled
    if E1's list is not empty
        if E2's list is not empty
            pass <e1, all e2s in E2's list, e3> to the parent with a new
            t_occ
        else
            pass <e1, e3> to the parent with a new t_occ
        flush E1 and E2's buffers
```

Except in recent context, an occurrence of E_1 stays valid until its matching E_3 occurs. Until the matching E_3 occurs, a time queue item is created with the instance number of the E_1 occurrence every time interval expressed by E_2. That is, it is like having a separate E_2' list for each occurrence of E_1.

```
PROCEDURE pstar_chronicle(Ei, parameter_list) /* Chronicle */

    if left event e1 is signalled
        get the terminator ID of e1
        if Recent context is not set
            create a time queue item with the ID and the time string of E2
        append e1 in E1's list.

    if middle event e2 is signalled
        get the ID of e2
        if E1' list is not empty
            if the ID is found in E1's list and if Recent context is not set
                create a time queue item with the ID and the time string
                of E2
            append e2 to E2's list

    if right event e3 is signalled
        if E1's list is not empty
            if E2's list is not empty
                pass <E1's head, all e2s in E2's list whose IDs are the same
                with the event ID of the E1's head, e3> to the parent
                with a new t_occ
                delete the head of E1's list
                delete all e2s in E2's list whose t_occ is less than that
                of E1's new head
                if E1's list is empty
                    flush E2's buffer
```

```
        else
            pass <E1's head,e3> to the parent with a new t_occ
            delete head of E1's buffer
```

PROCEDURE pstar_continuous(E_i, parameter_list) /* Continuous */

```
    if left event e1 is signalled
        get the terminator ID of e1
        if Recent or Chronicle contexts are not set
            create a time queue item with the ID and the time string of E2
        append e1 in E1's list.

    if middle event e2 is signalled
        if E1's list is not empty
        get the terminator ID of e2
        if Recent or Chronicle context is not set
            create a time queue item with the ID and the time string of E2
        append e2 to E2's list

    if right event e3 is signalled
        if E1's list is not empty
            if E2's list is not empty
                for each e1 in E1's list
                    pass <e1, all e2's whose t_occ are greater than that
                    of e1s, e3>
                    to the parent with a new t_occ
                flush E1's and E2's buffers.
            else
                for each e1 in E1's list
                    pass <e1, e3> to the parent with a new t_occ
                flush E1's buffer
```

PROCEDURE pstar_cumulative(E_i, parameter_list) /* Cumulative */

```
    if left event e1 is signalled
        get the terminator ID of e1
        if Recent,Chronicle, or Continuous context is not set
            create a time queue item with the ID and the time string of E2
        append e1 to E1's list

    if middle event e2 is signalled
        if E1's list is not empty
            get the terminator ID of e2
            if Recent,Chronicle, or Continuous contexts are not set
                create a time queue item with the ID and the time string
                of E2
            append e2 to E2's list
```

```
if right event e3 is signalled
    if E1's list is not empty
        for each e1 in E1's list
            link <e1 and all e2s whose IDs are the same as that
            of e1s >
        pass the <e1 and e2 list, e3> with a new t_occ
        flush E1's and E2's buffers
```

Figure 5.8 illustrates the detection of P^* event, $P^*(E_1,\ E_2,\ E_3)$ in various contexts.

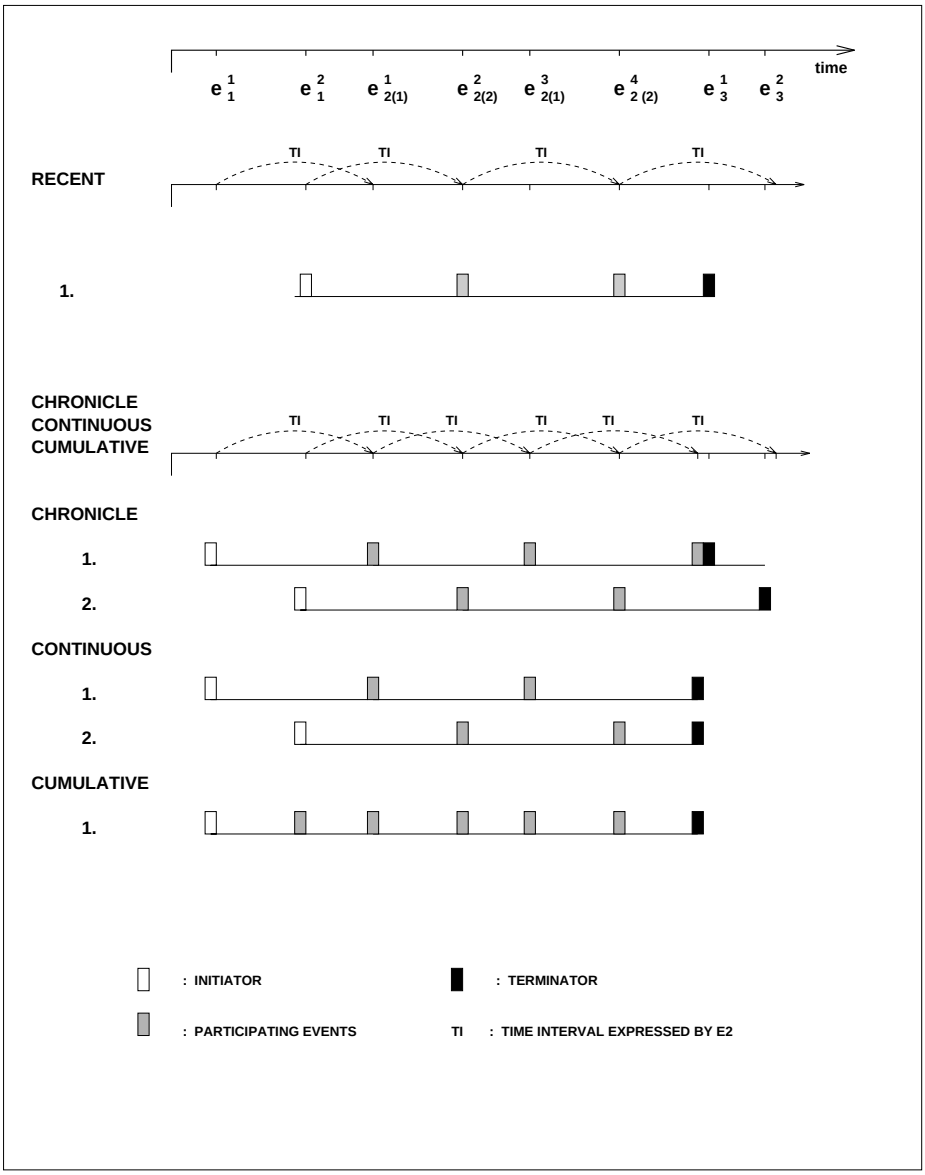


Figure 5.8. Illustration of P^*(E1, E2, E3) event detection in various contexts, where E2 is a time expression

CHAPTER 6
SNOOP PREPROCESSOR

In this chapter, we presents what a user should know to use the *Snoop* preprocessor and what the preprocessor produces, and discuss how the preprocessor works with Open OODB preprocessor. This chapter provides the syntactical aspects of the Chapter 3. The Snoop preprocessor preprocesses and postpreprocess all applicational source programs given by the user. The preprocessing declares appropriate events and rules with user-defined event and rule specifications expressed in Snoop and inserts them in the application program. And the postpreprocessing wraps all of the methods in the user-defined reactive classes with notifications.

6.1 Snoop BNF

The grammar for Snoop is shown below:

E ::= *begin-of* E1 | *end-of* E1 | E1
E1 ::= E1 AND[1] E2 | E1 OR E2 | E2
E2 ::= E2 SEQ E3 | E3
E3 ::= Any(Value, E4) | E5 | Any(Value, E5)
E4 ::= E4, E5 | E5
E5 ::= A(E1,E1,E1)
| $A^*(E1, E1, E1)$
| P(E1,[time string],E1)
| P(E1,[time string]:parameter,E1)
| $P^*(E1, [timestring] : parameter, E1)$
| $[absolutetimestring]$
| $(E1) + [relativetimestring]$
| *Explicit Events*
| *Database Events*
| L:(E1) /* Where L is a label */
| (E1)
Value ::= *integer* | ∞

Event expressions generated by the BNF[16] are left associative. Labels are used for expressing interest in a sub-expression and their presence determines the parameter attributes computed for that expression. A label acts as an identification for the subexpression in a complex event expression. A label is also interpreted as the *event type* for the sub-expression with which it is associated; for each such event a 'time of occurrence' attribute is generated as a parameter. Without label association it

[1] operators, OR, AND, SEQ will be given as symbols, which are |, ^ , and >> by a user

would not be possible to distinguish the occurrence of an event corresponding to a sub-expression and to access, if need be, the time of its occurrence from the parameter object.

6.2 Event and Rule Specification

The syntax of the Snoop event/rule specification is:

event_spec ::= **event** *event_modifier* method_signature
| **event** event_name = *event_exp*

event_modifier ::= event_name
| **begin** (event_name)
| **end** (event_name)
| **begin** (event_name) && **end** (event_name)
| **end** (event_name) && **begin** (event_name)

rule_spec ::= **rule** rule_name (event_name,
condition_function, action_function
[,[*parameter_context*],[*coupling_mode*]
,[*priority*],[*rule_trigger_mode*]])
parameter_context ::= RECENT | CHRONICLE | CONTINUOUS
| CUMULATIVE
coupling_mode::= IMMEDIATE| DEFERRED| DETACHED
priority::= positive integer
rule_trigger_mode::= NOW | PREVIOUS

Items enclosed by square brackets ([ ]) are optional. Both begin-method (by indicating **begin***(event_name)*) and end-method events (by indicating **end***(event_name)*) are supported. By default, the end of a method is taken to be the event. A composite event is expressed with *event_exp*, and a primitive event is specified with *event_modifier* and *method_signature*, which is a prototype of a method. In Sentinel both *class-level* and *instance -level* events are supported. A class-level event can be specified either inside of the class definition or in the application program by prefixing a class name before method signature as *"class_name::method_signature"*. An instance-level event can be specified only in the application program. The name of the instance and the name of the class should be given by the user. The instance name is given by postfixing after the name of the event as *"event_name:instance_name"*, and the class name is given as the class-level event specified outside of the class definition is. Only *integer, float, long integer, character,* and *character string* are supported as the types of the parameters of a method. Currently, only functions are used for specifying condition/action. In the current host environment (i.e., C++), methods cannot be used for condition/action since their invocation is tied to an object which is not known at compile time. But these condition and action functions can access stored objects as well as objects in the main memory. The parameter option must

come first; other can take any place. The parameter context of the event which a rule subscribes should be placed right after action_function if it exists. If several rules need to be defined on the same event in different parameter contexts, then the rule definition has to be duplicated for each context. *Coupling modes* refer to the execution points. Currently, *immediate* and *deferred* coupling modes are supported. We use *priority* classes for specifying rule priority. An arbitrary number of priority classes can be defined and totally ordered. We allow rule specification at class definition time and as part of an application. We also support rule activation and deactivation at run time. Moreover, named events can be reused later. This implies that a number of rules may be defined on the same event expression. We provide an option *rule_trigger_mode* for specifying the time from which event occurrences are to be considered for the rule. Two options, NOW (start detecting all constituent events starting from this time instant) and PREVIOUS (all constituent events are acceptable) are supported as rule triggering modes, with NOW being the default. To declare events and rules, the class should be a user-defined *reactive* class. Below, examples of events and a rule specified inside of the class definition are shown:

```
class STOCK: public REACTIVE  {
    public:
    event begin(e1) && end(e2) void set_price(float price);
    event end(e3) int sell_stock(int qty);
    int get_price();
    event e4 = e1 ^ e3; /* AND composite event */
    rule R1(e4, cond1, action1, CUMULATIVE, DEFERRED, 10, NOW); /* class-level rule */
    };
```

The following examples show the events and rules specified outside of the class definition, where is in an application program :

```
.....
STOCK *IBM, *DEC, *INTEL;
.....
event begin(e5:IBM) && end(e6:DEC) void STOCK::set_price(float price);
          /* Instance(IBM and DEC)-level event */
event end(e7) int STOCK::get_price(); /* Class-level event */
event e8 = STOCK_e4 >> e7; /* SEQ composite event */
rule R1(STOCK_e3, cond2, action2, DEFERRED, 20, NOW); /* class-level rule */
rule R1(e5, cond3, action3, CUMULATIVE, IMMEDIATE, 15); /* instance-level rule */
```

To access the events specified inside of the class definition from the outside, we have to prefixing the class name to the event name with an underline as *"class-name_eventname"*. In the above examples, STOCK_e3 and STOCK_e4 show the prefixing.

6.3 Preprocessing: Event and Rule Declarations

The preprocessing declares appropriate events and rules with user-defined event and rule specifications expressed in Snoop and inserts them in the application program. Here, we show how these events and rules preprocessing takes place and how the non-Snoop codes are processed with examples.

6.3.1 Primitive event

- Class-level event specified in a reactive class definition :
  class STOCK: public *REACTIVE*
  {
  ...
  event begin(e1) int buy_stock(int number);
  ...
  }
  The event specification in the above example is transformed as,
  PRIMITIVE *STOCK_e1 = new PRIMITIVE("STOCK_e1", "STOCK",
  "begin", "int buy_stock(int number)");

- Class-level event specified not in a reactive class definition :
  event begin(e2) && end(e3) void STOCK::set_price(float price);
  The above example is transformed into two primitive events as,
  PRIMITIVE *e2 = new PRIMITIVE("e2", "STOCK", "begin",
  "void set_price(float price)");
  PRIMITIVE *e3 = new PRIMITIVE("e3", "STOCK", "end",
  "void set_price(float price)");

- Instance-level event :
  An instance-level event can be specified only outside of a reactive class definition.
  ...
  STOCK *IBM, *DEC;
  ...
  event end(e4:IBM) int STOCK::sell_stock(int number);
  ...
  The instance name should be specified as well as the class name.
  It is transformed as,
  PRIMITIVE *e4 = new PRIMITIVE("e4", IBM, "end",
  "int sell_stock(int number)");

6.3.2 Composite event

A composite event is specified with a name and an event expression. The event expression specifies its constituent primitive or composite events using the Snoop operators. When an event expression is processed, calls for creating the event graph for that event expression which itself composes an event tree are added to the application

code. The examples are followed.

event e8 = A*(!(e1, e2, e3), e2, A(e4, e5, (e6^ e7)));

event e9 = (e3 | e1) ^ e2 ;

The two examples are transformed each as,
A_star *e8 = new A_star(new NOT(e1,e2,e3), e2, new A(e4, e5, new AND(e6,e7)));
AND *STOCK_e9 = new AND(new OR(STOCK_e3,STOCK_e1), STOCK_e2);
Note that in the second example, the event names are prefixed with the class name. This means that the example is specified inside of the class definition.

6.3.3 Temporal event

Both absolute and temporal events are mostly used as constituent events of composite events. They have time expressions instead of method signatures and event modifiers. The time expression does not have anything to do with a reactive class. Thus "*TEMPORAL*" replaces the class name, and a null string replaces the event modifier.

- Relative temporal event:
  When a relative time expression is used in one of P, P*, and PLUS operators, it is declared as a relative temporal event. But the temporal event does not have any name, thus, a name should be given.

  event e10 = P(e1, [1 hr], e2);

  If the above event is specified in the reactive class named STOCK, it is transformed as,
  PRIMITIVE *STOCK_rel1 = new PRIMITIVE("STOCK_rel1", "TEMPORAL",
  "", "1 hr");
  P *STOCK_e10 = new P(STOCK_e1, STOCK_rel1, STOCK_e2);
  The name "rel1" is given by the preprocessor. The number (here, it is 1) starts from 1 and increases by 1 in every reactive class. If the above example is specified in an application program, then it is converted as,
  PRIMITIVE *rel1 = new PRIMITIVE("rel1", "TEMPORAL", "",
  "1 hr");
  P *e10 = new P(e1, rel1, e2);
  The number in the name "rel1" starts from 1 at the beginning of the application program and increases by 1.

- Absolute temporal event :

  event e11 = [14:25:00/04/23/96];

  If the above absolute temporal event is specified in a reactive class definition, where the class name is "STOCK", it is converted as,
  PRIMITIVE *STOCK_e11 = new PRIMITIVE("STOCK_e5", "TEMPORAL",
  "", "14:25:00/04/23/96");
  If it is specified outside of the class definition, it is transformed as,
  PRIMITIVE *e11 = new PRIMITIVE("e5", "TEMPORAL", "",
  "14:25:00/04/23/96");

event e12 = P([00:00:00/01/01/96], [7 days], [00:00:00/12/31/96]);

In the above example, two absolute temporal events are specified without their names. The Snoop preprocessor give them names. If an absolute temporal event is specified outside of a class definition, an integer number is given with "abs" string as "abs1" for the event. If it is specified in a reactive class definition, the class name is prefixed as "STOCK_abs1". The integer starts from 1 and increases by 1 in every reactive class definition. The above example is transformed as,

```
PRIMITIVE *abs1 = new PRIMITIVE("abs1", "TEMPORAL",
                                "", "00:00:00/01/01/96");
PRIMITIVE *rel1 = new PRIMITIVE("rel1", "TEMPORAL",
                                "", "7 days");
PRIMITIVE *abs2 = new PRIMITIVE("abs2", "TEMPORAL",
                                "", "00:00:00/12/31/96");
P *e12 = new P(abs1, rel1, abs2);
or
PRIMITIVE *STOCK_abs1 = new PRIMITIVE("STOCK_abs1", "TEMPORAL",
                                "", "00:00:00/01/01/96");
PRIMITIVE *STOCK_rel1 = new PRIMITIVE("STOCK_rel1", "TEMPORAL",
                                "", "7 days");
PRIMITIVE *STOCK_abs2 = new PRIMITIVE("STOCK_abs2", "TEMPORAL",
                                "", "00:00:00/12/31/96");
P *STOCK_e12 = new P(STOCK_abs1, STOCK_rel1, STOCK_abs2);
```

6.3.4 Rule

There should be an event, a condition, and an action to specify a rule. As you can see in the rule specification, a rule can have at most four options for detecting the event which it subscribes or for triggering the rule in a certain mode. Except the first option which is *parameter_context*, they can be specified in any order.

rule r1[e1, check_price, set_price, RECENT, IMMEDIATE, NOW, 10];

The above rule specification is converted as the following if it is a class-level rule whose class is STOCK:

```
RULE *r1 = new RULE("r1", STOCK_e1, check_price, set_price, RECENT);
r1->set_mode(IMMEDIATE);
r1->set_parameter(NOW);
r1->set_priority(10);
```

6.3.5 Non-Snoop codes

The non_Snoop C++ codes are passed through the Snoop preprocessing without any modifications. The preprocessor also inserts Sentinel-related codes in the application program to make it easy for the user to use the Sentinel local event detector without worrying about details. The example application program and its after-Snoop-preprocessed C++ codes can be found in section 6.7.

6.4 Side-Output of Preprocessing

The preprocessor produces several files for other Sentinel server applications such as the rule execution/visualization tool[20] and rule editor.[2]. For the visualization, events and rules are reported with their descriptions and their class names For rule editor, all of the method signatures of reactive classes are enumerated in a file.

6.5 Postprocessing and Integrating into Open OODB preprocessor

Event methods that can generate primitive events should be wrapped with notifications. The Open OODB preprocessor also wraps class methods for its *sentry* mechanism. The Open OODB preprocessor renames an original method by postfixing it with a string *"_OOdbFn"*, creates a wrapper method which has the original method name, and inserts calls into the wrapper method. The function named *xwrapper_func_code* generates OODB code for the wrapped methods. We modified it to insert notifications if the method is one of a reactive class. The rule editor allows the user to create rules in run time. For the event methods which will be created and subscribed by these rules, the notifications are inserted to all of the methods of a reactive class with a condition. The condition checks to see if there are any rules subscribing the method at that time. If there are, the notifications go to the local event detector before and after the invocation of the original user method. Before any notification of the method, the parameters of the method are collected, linked in a list and sent with the notifications to the event detector. An example of a wrapper method after the Open OODB and Snoop postprocessing can be found in section 6.7.

6.6 Running Snoop Preprocessor

The Open OODB toolkit drivers calls the C++ preprocessor, Open OODB preprocessor and then C++ compiler. By modifying the driver to call the Snoop preprocessor first, using the Sentinel local event detector becomes easy for the user. The Snoop preprocessing can be disabled by the **s** option; in that case, the user only calls the Open OODB preprocessor. The option is given like those of the Open OODB preprocessor or C++ compiler.

6.7 Example of Snoop Preprocessing

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();
```

[2]It is under development.

```
        event e4 = e1 ^ e2; /* AND operator */
        /* class-level rules */
        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 begin(instance_set_price:IBM) void STOCK::set_price(float price);
        /* SEQUENCE operator */
        event seq_event = STOCK_e4 >> instance_set_price;
        /* Creating class-level primitive event */
        event begin(sell_stock) void STOCK::sell_stock(int qty);
        /* Creating class-level P event */
        event p_event = P([00:00:00/01/01/96], [7 days], [00:00:00/12/31/96]);
        /* Creating a rule which contains both class-level
        and instance-level events */
        rule R2[seq_event, cond2, action2,20, PREVIOUS];
        /* Creating a class-level rule */
        rule R3[p_event, cond3, action3, RECENT];
        .....
        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();
}
```

Snoop preprocessed program

```
class STOCK : public REACTIVE
{
        private:
        .....
        public:
        .....
        int sell_stock(int qty);
```

```
    void set_price(float price);
    int get_price();
};
/* Main program */
STOCK IBM, DEC, Microsoft;
LOCAL_EVENT_DETECTOR *Event_detector;
void init_func();
main()
{
    ..................
    /* Creating the local event detector */
    Event_detector = new LOCAL_EVENT_DETECTOR();
    init_func();
    /* Creating primitive events */
    PRIMITIVE *STOCK_e1 = new PRIMITIVE("STOCK_e1", "STOCK"
                                "end", "int sell_stock(int qty)");
    PRIMITIVE *STOCK_e2 = new PRIMITIVE("STOCK_e2" "STOCK",
                                "begin", "void set_price(float price)");
    PRIMITIVE *STOCK_e3 = new PRIMITIVE("STOCK_e3", "STOCK",
                                "end", "void set_price(float price)");
    /*Composite event AND */
    AND *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 *instance_set_price = new PRIMITIVE("instance_set_price",
                                        IBM, "begin", "void set_price(float price)");
    /* Composite event SEQUENCE */
    SEQ *seq_event = new SEQ(STOCK_e4, instance_set_price);
    /* Composite event P */
    PRIMITIVE *abs1 = new PRIMITIVE("abs1", "TEMPORAL",
                                "", "00:00:00/01/01/96");
    PRIMITIVE *rel1 = new PRIMITIVE("rel1", "TEMPORAL",
                                "", "7 days");
    PRIMITIVE *abs2 = new PRIMITIVE("abs2", "TEMPORAL",
                                "", "00:00:00/12/31/96");
    P *p_event = new PSTAR(abs1, rel1, abs2);
    /* Creating Rule R2 */
    RULE *R2 = new RULE("R2", seq_event, cond2, action2);
    R2->set_priority(20);
    R2->set_trigger_mode(PREVIOUS);
    /* Creating Rule R2 */
```

```
    RULE *R3 = new RULE("R3", p_event, cond3, action3, RECENT);
    Notify(NULL, "OODB", "beginT", "begin", system_list);
    OpenOODB->beginTransaction();
    Notify(NULL, "OODB", "beginT", "end", system_list);
        IBM.set_price(115.00);
        DEC.set_price(100.00);
        Microsoft.sell_stock(200);
        DEC.get_price();
        IBM.set_price(75.95);
    Notify(NULL, "OODB", "commitT", "begin", system_list);
    OpenOODB->commit();
    Notify(NULL, "OODB", "commitT", "end", system_list);
}
```

Open OODB preprocessed program

```
class STOCK : public virtual _Wrapper, public REACTIVE
{
    private:
    .....
    public:
    .....
    int sell_stock_OOdbFn(int qty);
    int sell_stock(int __1ooAgr0);
    void set_price_OOdbFn(float price);
    void set_price(float __1ooAgr0);
    int get_price_OOdbFn();
    int get_price();
};
int STOCK::sell_stock_OOdbFn(int qty)
{
    /* original sell_stock method */
}
int STOCK::sell_stock(int __1ooArg0)
{
    .... Open OODB code ...
    /* Parameters are collected in a linked list */
    PARA_LIST *sell_stock_list = new PARA_LIST();
    sell_stock_list->insert("qty", INT, __1ooArg0);
    if is_begin_of_this_subscribed_
       /* Notify begin of method */
       Notify(this, "STOCK", "int sell_stock(int qty)",
              "begin",sell_stock_list);
    /* The original sell stock method is invoked here */
```

```
    int ret_value = sell_stock_OOddbFn(__1ooArg0);
    /* Only if this event is subscribed */
    if is_end_of_this_subscribed
       /* Notify end of method */
       Notify(this, "STOCK", "int sell_stock(int qty)",
                                "end",sell_stock_list);
    return(ret_value);
}
void STOCK::set_price_OOdbFn(float price)
{
    /* original set_price method */
}
void STOCK::set_price(float __1ooArg0)
{
    .... Open OODB code ...

    /* Parameters are collected in a linked list */
    PARA_LIST *set_price_list = new PARA_LIST();
    set_price_list->insert("price", FLOAT,__1ooArg0);
    if is_begin_of_this_subscribed
       /* Notify begin of method */
       Notify(this, "STOCK", "void set_price(float price)",
                                "begin",set_price_list);
    /* The original set price method is invoked here */
    set_price_OOdbFn(__1ooArg0);
    if is_end_of_this_subscribed
       /* Notify end of method */
       Notify(this, "STOCK", "void set_price(float price)",
                                "end", set_price_list);
}
int STOCK::get_price_OOdbFn(char *n1)
{
    /* original get_price method */
}
int STOCK::get_price(char __1ooArg0)
{
    .... Open OODB code ...
    /* Parameters are collected in a linked list */
    PARA_LIST *get_price_list = new PARA_LIST();
    get_price_list->insert("n1", char, __1ooArg0);
    if is_begin_of_this_subscribed
       /* Notify begin of method */
       Notify(this, "STOCK", "void get_price(char *n1)",
```

```
                                "begin",get_price_list);
/* The original set price method is invoked here */
int ret_value = get_price_OOdbFn(_1ooArg0);
if is_end_of_this_subscribed
   /* Notify end of method */
   Notify(this, "STOCK", "void get_price(char *n1)",
                                "end", get_price_list);
return(ret_value); ]] }
```

This example illustrates the use of class-level and instance-level events and rules and also shows the wrapping of the methods with a collection of parameters, which are done by the Open OODB preprocessor. 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). A periodic event *p_event* is defined with absolute time interval. R3 subscribes p_event. Notice that after preprocessing the user-defined methods 'sell_stock', 'set_price', and 'get_price' are renamed as 'sell_stock_OOdbFn', 'set_price_OOdbFn' and 'get_price_OOdbFn', and wrapper methods 'sell_stock','set_price' and 'get_price' are introduced. Currently 'get_price' is not subscribed by any of event of rule, but the reason why it is also wrapped is that we allow the user to create the rules subscribing the event later through the rule editor. As seen from the example, appropriate code is introduced in the wrapper methods to notify the events. Also the processing of the application - level rule and event specification procues 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. Rule R3 is fired every seven days between "00:00:00/01/01/96" and "00:00:00/12/31/96".

CHAPTER 7
CONCLUSIONS

This thesis extends earlier work on local event detection in Sentinel. The local event detector was extended to support temporal events according to the semantics. We integrated the local event detector with the stand-alone temporal event handler first to support other Snoop operators. For the integration, the implementation of the local event detector and the temporal event handler were modified. In the local event detector, the implementation of primitive events was modified to generate temporal items which are handled by the temporal event handler since a temporal event is a primitive event. The Snoop operators, A and A*, are reimplemented with new parameter computation algorithms. P and P* are implemented to have time capabilities with their new event detection algorithms. PLUS is implemented for supporting relative temporal events. In addition, the stand-alone temporal event handler has been changed to integrate with the local event detector to meet the new complicate situations which were not found when it was implemented originally.

The earlier work assumed that Snoop event/rule specifications were transformed to internal event/rule declarations, and the specified event methods were wrapped with notifications to the local event detector. We implemented the Snoop preprocessor so that the event/rule specifications are transformed to the internal forms.

We removed the Snoop postpreprocessor and modified the Open OODB preprocessor to wrap the methods of a reactive class. Every method of a reactive class is wrapped by notifications with condition statements which check the method is bound to an event at run time. This extended wrapping allows extra rule editing at run time. The preprocessor produces a few side files for other Sentinel applications such as global rule editor and rule debugger.

This thesis also explained the local event detector and the temporal event handler. Constructing of an event graph, detecting of both primitive and composite events, and handling of temporal events were described.

REFERENCES

[1] 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.

[2] R. Badani. Nested Transactions for Concurrent Execution of Rules: Design and Implementation. Master's thesis, University of Florida, Gainesville, 1993.

[3] S. Chakravarthy. Hipac: A research project in Active,Time-constrained Database Management. Technical report, Xerox Advanced Information Technology, 1989.

[4] S. Chakravarthy. Divide and conquer: A Basic for Augmenting a Conventional Query Optimizer with Multiple Query Processing Capabilities. In *Proc. of the 7th Int'l Conf. on Data Base Technology (EDBT)*, Venice,Italy, March 1990.

[5] 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, Gainesville, December 1993. (Submitted for publication.).

[6] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings, International Conference on Very Large Data Bases*, pages 606–617, August 1994.

[7] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(10):1–26, October 1994.

[8] S. Chakravarthy and D. Mishra. Towards an Expressive Event Specification Language for Active Databases. In *Proc. of the 5th International Hong Kong Computer Society Database Workshop on Next Generation Database Systems*, Kowloon Shangri-La, Hong Kong, February 1994. (Invited Paper).

[9] S. Chakravarthy, Z. Tamizuddin, and J. Zhou. SIEVE: An Interactive Visualization and Explanation Tool for active Databases. In *Proc. of the 2nd International Workshop on Rules in Database Systems (RIDS'95)*, pages 179–191, October 1995.

[10] U. Dayal, A. Buchmann, and D. McCarthy. Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database Management System. In *Proceedings 2nd International Workshop on Object-Oriented Database Systems*, Bad Muenster am Stein, Ebernburg, West Germany, Sept. 1988.

[11] 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 (Catalonia), Spain, Sept. 1991.

[12] 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 (Catalonia), Spain, Sep. 1991.

[13] 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.

[14] 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.

[15] V. Krishnaprasad. Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation. Master's thesis, University of Florida, Gainesville, 1994.

[16] D. Mishra. SNOOP: An Event Specification Language for Active Databases. Master's thesis, University of Florida, Gainesville, August 1991.

[17] J. Moss. Nested Transactions: An Approach To Reliable Distributed Computing. MIT Laboratory for Computer Science, MIT/LCS/TR-260, 1981.

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

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

[20] J. Zhou. Sieve: An Interactive Visualization and Explanation Tool for an Active OODBMS. Master's thesis, University of Florida, Gainesville, Aug 1995.

[21] G. A. Ziemann. Detailed design description: Time queue handler. Senior Project Report, University of Florida, Gainesville, May 1995.

BIOGRAPHICAL SKETCH

Hyesun Lee was born on December 25, 1969, at Taejeon, Korea. She received the Bachelor of Science degree in physics from the Ewha University, Seoul, Korea, in February 1992. In the Spring of '94, she started her graduate studies with a major in computer and information science and engineering at the University of Florida. She will receive her Master of Science degree in computer and information science and engineering from the University of Florida, Gainesville, in August 1996. Her research interests include active and object-oriented databases.