SNOOP: AN EVENT SPECIFICATION LANGUAGE FOR ACTIVE DATABASE SYSTEMS

By

DEEPAK MISHRA

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

1991

ACKNOWLEDGEMENTS

It's a great pleasure to express my sincere gratitude to Dr. Sharma Chakravarthy for giving me an opportunity to work on this interesting topic and for providing continuous guidance and support throughout the course of this research work. I am grateful to Dr. Sham Navathe for introducing me to the field of database management and for assisting beyond academic requirements. I thank Dr. Herman Lam for serving on my supervisory committee and for his careful perusal of this thesis. I am deeply indebted to my uncle, Dr. Sachchidanand Mishra, for his continuous support, inspiration and encouragement. I would like to thank Sharon Grant for maintaining a well administered research environment. I also thank many of my friends for making my stay in Gainesville memorable.

This research was supported by the National Science Foundation Grant IRI# 9011216 and the Florida High Technology and Industrial Council Grant UPN# 89090848.

TABLE OF CONTENTS

APPENDICES

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

SNOOP: AN EVENT SPECIFICATION LANGUAGE FOR ACTIVE DATABASE SYSTEMS

By

Deepak Mishra

December 1991

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

Several application areas such as office automation, inventory control, computer integrated manufacturing (CIM), network management and hospital monitoring require timely response to critical situations in addition to automatic monitoring. These applications are poorly served by the state-of-the-art DBMSs that are passive in nature. Active DBMSs present innovative technology to model such applications. These systems maintain predefined events, conditions and actions (or situation-action rules) and whenever a specified event occurs, execute corresponding actions according to the outcome of the condition evaluation.

Making a system active to meet the requirements of a wide range of applications entails an expressive event specification capability and its efficient detection. Extant systems do not support a variety of events and event operators needed to model non-traditional applications.

This thesis defines an event specification language, SNOOP. We give precise semantics of primitive events and several event operators needed to express composite events. SNOOP supports temporal, periodic, aperiodic, explicit and composite events in addition to the traditional database events and provides a method for handling contingency plans used in time-constrained applications.

Furthermore, we propose the design of a composite event detector that captures the notion of modularity and extensibility. We define different contexts, namely Recent, Chronicle and Cumulative, for detecting composite events and describe their implications on event signalling.

CHAPTER 1
INTRODUCTION

A traditional database is a passive repository of data where the system only executes explicit requests from applications. This traditional view of databases as information repositories used for storing and explicitly retrieving required information was adequate for a large class of applications. However, the need for having a database system capable of reacting to specific situations (event-condition pairs) has been recognized in several newer applications. These applications require automatic monitoring of conditions defined over the state of the database and a capability to take actions, possibly subject to some timing constraints, when the state of the underlying database changes. For example, inventory management in a factory requires that the quantity on hand of each item be monitored and whenever the quantity on hand of an item falls below a threshold, then a reorder process may have to be initiated. This scenario involves monitoring for an event – the transaction that caused the quantity to decrease, evaluation of a condition – the quantity of the item going below the threshold and executing one or more actions – reordering the product and notifying the manager. Similarly, in a library database, borrowers are required to return the books before they are due; otherwise a notification needs to be sent to those who failed to return the book. This requires monitoring the due date of the book and issuing a notification and computing the late charges when the book is returned.

A large class of non-traditional applications such as hospital monitoring system, air-traffic control, process control, network management, computer integrated manufacturing, office automation, computer aided design (CAD) and battle management also require timely detection of and response to critical situations. For example, a network control system monitors various types of events, such as network partitions, link overloading, hardware failure and takes corresponding actions such as finding alternate path, to get the communication failure status of modems and to correlate this with past failures to diagnose the problem and take appropriate measures. Similarly, view management in a distributed information system requires monitoring any of changes in the stored relation and automatic rectification of any inconsistency or notification of the user when a cache becomes obsolete. Many time-constrained applications need to take alternate actions if the specified action could not be executed within the prespecified time. For example, in a process control application, the boiler pressure needs to be monitored and if it increases by an undesirable amount, remedial measures need to be taken within a specified time to release the pressure and if that is not possible, a contingency plan (e.g. sounding an alarm to evacuate the boiler room) may have to be executed.

Design databases have non-trivial consistency requirements. Data objects in computer aided design (CAD) are characterized by highly complex structures and a lot of intricate dependencies. This results in numerous consistency constraints that have much higher complexity than traditional business oriented database applications. In addition, CAD applications typically have long transactions. This entails tolerating inconsistency over unpredictably long periods of time. The time and extent of checking consistency and how to react to a consistency violation needs to be determined dynamically and under user control rather than only at the end of the transaction.

These additional requirements of design database are not met by the traditional approach of integrating consistency control within the transaction manager ([DIT86], [KOT88]).

All of the above applications require monitoring of conditions defined on the state of the database and evaluate the conditions when the state of the database changes to invoke specific actions. For these applications, the correctness of result depends not only on the correct interleaving of transactions but also on the timely invocation of conditions and actions. These applications are poorly served by the state-of-the-art database management systems.

In the absence of explicit support for situation monitoring, requirements of time constrained applications can be met using special purpose mechanisms in at least two ways [CHA89a]. The first approach is to write a special application program that periodically queries or polls the database to determine if situations being monitored have occurred. Figure 1.1 depicts the polling approach on a passive DBMS. The second approach is to embed the situation monitoring within each program that updates the database and also include the corresponding action as part of the application code. This approach of augmenting each program to include situation-action mechanism is shown in Figure 1.2. Neither of these two approaches is completely satisfactory. Polling requires fine tuning of the periodicity with which the database should be queried to obtain a timely response. Frequent polling leads to thrashing i.e. overloading the database with queries that return empty answers most of the time, whereas infrequent polling runs the risk of missing the response window. Embedding situation monitoring in an application limits the extent to which condition evaluation can be optimized and severely compromises modularity, since the updating component is logically performing the task of another software component that should respond to the update. Therefore any modification to the situations being monitored or to the corresponding actions will require modifying every application program that updates the database.

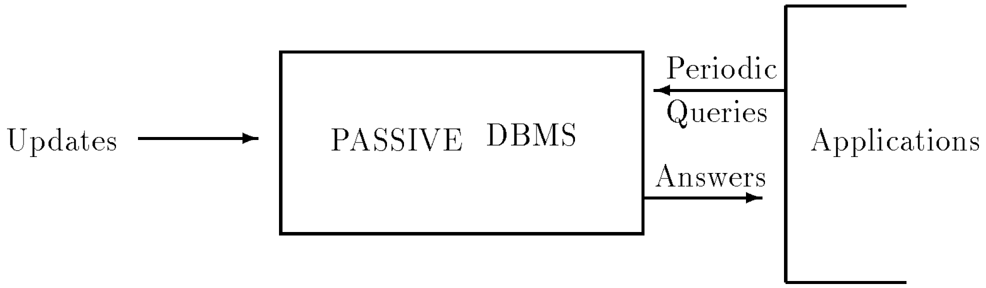


Figure 1.1. The Polling Approach

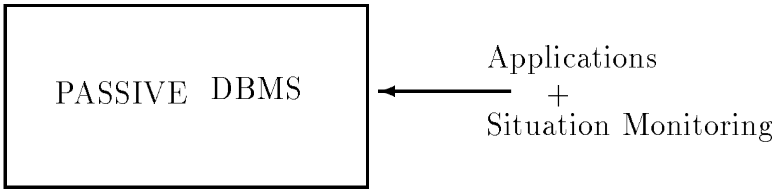


Figure 1.2. Embedding Situation Monitoring in Applications

Making a DBMS active by extending its functionality to incorporate efficient situation monitoring as its integral part (as proposed in HiPAC [CHA89a] is an elegant solution to these problems. We define an active DBMS as a system that provides full functionality of a traditional DBMS and is capable of reacting automatically and asynchronously to events occurring in its environment. The system continuously monitors occurrences of these events and reacts without waiting for explicit user or application-initiated requests. Situations, actions and timing requirements are all specified declaratively to the system. The system then monitors the occurrences of specified events and executes corresponding actions according to the outcome of the condition evaluation. This provides both modularity and timely response.

Before an event can be recognized, it should be specified and its parameters should be defined in the system. A survey of recent work in this field [CHA90b] reveals that these systems support a small number of events and provide few operators to express composite events. In this thesis, we describe an event specification language SNOOP that subsumes the events supported in the existing systems and extends them by providing newer events and event operators.

This thesis is organized as follows: Chapter two briefly describes active constructs provided in the different branches of computer science with emphasis on database systems. We analyze current research and commercial efforts in the field of active databases to identify the events and event operators that are needed but not supported in these systems.

In Chapter three we set the requirements for an event specification language. We discuss applications that are not well served by the extant systems and based on that identify events and event algebra need to be supported.

Chapter four presents our approach to event classification and specification. We discuss event hierarchy, identify primitive events and define operators along with their semantics and give examples to show their need. We also provide an event specification language that can be used to generate complex event expressions and propose a method to implement contingency plans required in a time-constrained environment.

Chapter Five deals with event detection. We define various contexts of detecting complex events and give algorithms for their detection and evaluating their parameters. We also propose the design of a composite event detector.

Finally, Chapter Six concludes the research presented in this thesis and suggests areas for future research.

CHAPTER 2
SURVEY OF RELATED WORK

This chapter briefly discuss situation monitoring in different application areas such as operating systems, programming languages, artificial intelligence and database management systems. We specifically analyze recent research and commercial efforts in the field of active databases to identify the events supported in these systems.

2.1 Interrupts and Signals in Operating Systems

An operating system is an *event-driven* program [PET85]. It waits for various synchronous and asynchronous occurrences, and when such events occur, it responds to their need according to some predefined priority. Operating systems provide two basic facilities namely, interrupts and signals, to model various events.

2.1.1 Interrupts

An interrupt is a signal to the central processor indicating that a special event has occurred and that at the earliest convenient time, the system should temporarily suspend its current activity and respond to the needs of the event. Types of interrupts actually supported in the system and their priorities depend on the computer manufacturer, model of computer and the operating system configuration of the particular installation. Table 2.1 shows a list of typical interrupt classes [TUR86].

Table 2.1. Typical Interrupt Classes

Class *Increasing Priority*	**Description**
System call	Request to the operating system for a standard service
Program error	Invalid instruction, invalid access to protected memory, division by zero, etc.
Input/output	Input/output device needs attention
Timer	Designated time of day is reached or an interval of time has elapsed
Machine Malfunction	Memory parity error, invalid CPU state, etc.
Power failure	Loss of electrical power

2.1.2 Signals

Signals inform processes of the occurrences of asynchronous events. They are the software equivalent of interrupts and signal-handling routines perform the equivalent function of interrupt service routine. Processes may send signals to each other or the kernel may send signals internally. Broadly, signals can be classified as follows [BAC88]:

- Signals having to do with the termination of the process, sent when a process exits or when a process invokes the *signal* system call.

- Signals having to do with process induced exceptions such as when a process accesses an address outside its virtual address space, when it attempts to write memory that is read-only (such as program text), or when it executes a privileged instruction or for various hardware errors.

- Signals having to do with the unrecoverable conditions during a system call, such as running out of system resources during *exec* after the original space has been released.

- Signals caused by an unexpected error condition during a system call, such as making a nonexistent system call, writing a pipe that has no reader processes, or using an illegal "reference" value for the *lseek* (Unix) system call.

- Signals originating from a process in user mode, such as when a process wishes to receive an *alarm* signal after a period of time, or when processes send arbitrary signals to each other with the *kill* (Unix) system call.

- Signals related to terminal interaction such as when a user hangs up a terminal, or when a user presses "break" or "delete" keys on a terminal keyboard.

- Signals for tracing execution of a process.

2.2 Exception Handling in Programming Languages

Program execution may cause some exceptional occurrences as a result of errors, such as overflow, underflow, attempted division by zero or array subscript going out of range. These "exceptional" events may make normal program execution undesirable or even impossible. The term exception refers to an exceptional situation or error. Exceptions have three aspects: declaration, raising and handling. To declare an exception means to give it a name. To raise an exception means to abandon normal program execution. To handle an exception means to take some appropriate actions. When an exception is raised, normal program execution is terminated and the control is transferred to an exception handler which is a specially written part of the program.

Notable examples of languages which provide exception handling are PL/I, MESA, ML, CLU and Ada. Though Algol-68 also included exception handling facilities, these were restricted to files and were not available for general program execution. PL/I and Ada provide exception handling in a fairly general forms to enable them to be used as a standard programming tool rather than exclusively for error handling.

A *condition* in a PL/I program is an occurrence that can cause a program interrupt. It may be the detection of an unexpected error or an occurrence that is expected at an unpredictable time. For example, OVERFLOW is an unexpected error. It occurs when arithmetic expression generates answer that is too large for the specified data format. The FINISH condition is an example of an occurrence that is expected but at an unpredictable time. A number of conditions may be specified in the ON statement [HUG79].

Ada contains an extensive set of features for exception handling ([CAV86], [GEH84]). These are on the same lines as the PL/I On-conditions, but are more powerful and somewhat better designed. There are two types of exceptions in Ada: pre-defined and user-defined. Predefined exceptions need not be declared or raised by the user. They are raised automatically when the corresponding situation is detected. User-defined exceptions are raised explicitly by the *raise* statement.

2.3 Procedural Attachment and Demons in A. I.

2.3.1 Procedural Attachment

Procedural attachment is a method of attaching programs to the data structures, such as slots in a frame ([CHAR80],[WIN79]). Slot values are obtained by executing these programs for properly instantiated arguments. There are typically three kinds of procedural attachment– IF-NEEDED, IF-ADDED, and IF-REMOVED that provide a mechanism for triggering arbitrary procedures.

IF-NEEDED procedure is triggered IF we NEED the value of a slot that does not have an explicitly stated value. IF-ADDED procedure is run to perform an arbitrary computation when a new item is added to the slot. IF-REMOVED procedure is invoked when an item is removed from the slot.

2.3.2 Demons

Demons are a convenient construct for situations in which we wish to specify the execution of asynchronous activity that arises from a particular change of state in the modeled environment. A demon has a trigger and a response. As an example, consider a demon DC-OHM in an expert problem solving system ARS [WIN79]. This demon implements ohm's law in a circuit-specific knowledge base.

```
(LAW DC-OHM ASAP ((R RESISTOR) V1 V2 I RES)
        ()
        ((= (VOLTAGE (T1 !?R)) !>V1) (= (VOLTAGE (T2 !?R)) !>V2)
        (= (CURRENT (T1 !?R)) !>I) (= (RESISTANCE !?R) !>RES))
        (EQUATION '(&– V1 V2) '(&* RES I) R))
```

The keyword **LAW** defines the demon DC-OHM. **ASAP** indicates its invocation priority. **V1, V2, I and RES** are the local variables to hold the two terminal voltages, the current, and the resistance value of the resistor. In addition, the type-restricted local variable **R** is used for the resistor about which the deduction will be made. The long list beginning with (= (VOLTAGE ... contains the demon's trigger slots. Their purpose is dual: to provide patterns to direct the invocation or triggering of the demon, and to gather the information needed in applying Ohm's law once the demon is invoked.

The ARS antecedent reasoning mechanism will signal DC-OHM whenever a fact is asserted that matches any of DC-OHM's trigger slots. When the demon is invoked, it will apply all of its trigger patterns to the database, using its argument as the value of **R** during the match to make sure that it finds voltages, current and resistance for a single resistor instead of for four different resistors. Variables appearing in the pattern with the "!>" operator have no effect on the triggering of the demon, but at the matching stage they are assigned whatever value they happen to match, if the pattern matches anything at all.

After the matching phase, the body of the demon is executed. In this case the body is just a call to the function **EQUATION**, which does all the work of extracting any possible new information from the specified equation and the parameter values obtained by the matching phase.

2.4 Situation Monitoring in Databases

In the following subsections we mention initial attempts to handle situation monitoring in databases (CODASYL and System R) and provide a brief survey of recent research and commercial efforts in the field of active database systems. The goal of the survey is to evaluate the approaches taken in these systems and to identify their limitations for modeling various applications. A detailed evaluation of some these systems can be found in [CHA90b].

2.4.1 ON Clause in CODASYL

CODASYL [OLL78] is one of the oldest database system recommendation that was developed to overcome the problems of file systems. The CODASYL recommendation provides database procedures those are defined by the database administrator and are automatically invoked at execution time when some situation arises. This mechanism is implemented by using "ON Clause" which can be defined at the following levels:

- realm

- record type

- item
- set type

As a simple example of the ON clause, consider the one on the item level:

ON [ERROR DURING] *database operation* CALL db-procedure

Where a database operation can be STORE, GET or MODIFY.

For example:

ON STORE CALL LOGPROC

specifies that immediately after the execution of the STORE statement, procedure LOGPROC is executed and then the control is transferred back to the statement after STORE. ERROR DURING option specifies that the action should be executed only when an error occurs during the execution of STORE rather than at the end of the execution.

ON clauses on realm, record types and set types are similar to the above discussion for items.

2.4.2 System R

In system R, ([ESW75],[ESW76]) active constructs are used for integrity checking. It provides a set of integrity facilities in the form of SQL statements such as ASSERT, DROP ASSERTION, DEFINE TRIGGER, DROP TRIGGER and ENFORCE INTEGRITY. The ENFORCE INTEGRITY statement, forces the application to check specified integrity constraints at the end of the transaction without actually causing a COMMIT. Using these constructs, it is possible to specify rules that are triggered whenever the state of the database changes.

2.4.3 Ariel

Ariel [HAN89] is a relational database system which is built upon the foundation provided by the EXODUS database tool kit. It supports rules that are triggered either by database events or temporal events. Database events are associated with the following kinds of database operations (optional clauses are enclosed by square brackets):

- **append [to]** relation-name
- **delete [from]** relation name
- **replace [to]** relation-name [(attribute-list)]
- **retrieve [from]** relation-name [(attribute-list)]

Ariel adopts set-oriented method of rule triggering. This means the rules are triggered only once at the end of the corresponding database operation for a set of *affected* tuples. This is in contrast to the approach where a rule is triggered once for each affected tuple.

Temporal events are also supported in this system. System generates one temporal event every second. Syntax for the *time-specifier* is as follows:

time-specifier →
  time = *time-list*
    | **every [INTEGER]** *time-unit*
      [**starting** *time-value*]
      [**ending** *time-value*]

where a *time-value* has the form YY:MM:DD:HH:MM:SS, a *time-list* is a comma separated list of one or more *time-values* and a *time-unit* can be one of hours, days etc. In the above specification YY:MM:DD and SS are optional. If YY:MM:DD fields are omitted the event becomes a periodic event that occurs each day at HH:MM:SS. Omission of SS fields defaults to 00. It provides *disjunction* of temporal events in the form of a *time-list* which is a list of time-values, separated by commas.

The parameters of database events are represented by a tuple variable whose scope is determined for different database operation (shown in bold with optional clauses surrounded by square brackets) as follows:

- **append [to]** R: The tuple variable R is bound to the set of tuples just appended to R.
- **delete [from]** R: The tuple variable R is bound to the set of tuples just deleted from R.
- **replace [to]** R [(attribute-list)]: The tuple variable R is bound to the set of new tuples just created by modifying existing tuples from R. The tuple variable **previous R** refers to the previous values of the modified tuples.
- **retrieve [from]** R [(attribute-list)]: The tuple variable R is bound to the set of tuples just retrieved from R.

Temporal events seem not to contain any parameters in this model.

2.4.4 Event/Trigger Mechanism (ETM)

Event/Trigger Mechanism (ETM hereafter)([DIT86], [KOT88]) is a system developed at University of Karlsruhe to meet the complex consistency requirements in design databases. In ETM, an event is an indicator (represented by a specific identifier) that can be raised to flag a certain situation to the database. The declaration of an event type is done by giving a event identifier $< E\_id >$ and a list formal parameters $< fop >$ to pass context information from event to action.

$$\texttt{event} < E\_id > (< fop_1 >:< fop\_type >, ..., \\ < fop_n >:< fop\_type >));$$

An event type can be instantiated an arbitrary number of times by 'raising the event'. This is done by calling the operation:

$$\texttt{raise} < E\_id > (acp_1, acp_2....., acp_n);$$

with actual parameters acp_i.

There are two kinds of events supported in ETM, standard (implicit) event and explicit event. Standard events are associated with the *start* and *termination* of database operations and are raised by the system itself. Given a specific database schema, it is possible to generate type-dependent standard events (e.g. insertion of a record of type t). Explicit events are events that are raised explicitly by the user or application program.

2.4.5 High Performance Active Database System (HiPAC)

HiPAC ([CHA89a],[CHA89b], [CHA90a]) was a research effort on active and time constrained data management. In HiPAC, an event is an entity that has an identifier and a list of typed formal parameters. For each event, one operation *signal* is defined that binds the formal parameters specified for the event to actual parameters. It has five kinds of primitive events:

1. Database events
2. Begin of the Transaction
3. End of the Transaction
4. Temporal Events
5. Abstract Events

Database events are related to database operations and are further classified into Insert, Delete and Update. For example, Insert_Position (Ship-ID: String, Location : (Lat, Long)) is a database operation that inserts an entity instance of the Position entity type into the database. When an operation is executed, the parameters are bound to actual entities and values in the database such as Insert_Position (S1234, (40N, 70W)). Since database operation executions are not instantaneous but occupy intervals of time, two events are defined for each operations: the *beginning* of the operation and the *end* of the operation.

As the name implies, begin and end of the transaction are events those are signalled at the beginning and end of a transaction. Parameters of these events include the transaction, user, and session identifiers, and – implicitly – the entire database state.

A temporal event can be absolute points in time, defined by the system clock (e. g., 9:00:00 a.m., April 10, 1988), relative (30 seconds after event A occurred), or periodic (every day at midnight). An absolute event is specified as a time string, relative event is defined as a reference event and a reference interval and periodic event is specified as a reference point (i.e. the point of first occurrence) and a period (an interval). Parameters of a temporal event are Event-id and Description.

Abstract events are not necessarily associated with a database operation or time, and cannot be directly detected by HiPAC. These events and their parameters are defined in the model, but are detected and signalled by users or other programs. For example, Flight_Airborne (Flight_No, Destination, Takeoff_Time, Aircraft, Wind_Speed, Wind_Direction) is an abstract event, which is signalled by a user when a flight takes off.

In addition to these primitive events, three event operators, namely, disjunction, sequence and closure are provided to form composite events.

The *disjunction* of two events, E1 and E2, is a composite event E, denoted (E1 | E2), that is signalled when either E1 or E2 is signalled. E's arguments are the "outerunion" of E1's arguments and E2's arguments. For example, in a battle management application, if both ships and targets can move, then a single rule fired by the composite event (Update_Position (S : Ship) | Update_Position (T : Target)) can check whether the critical distance between ships and targets has been crossed, instead of writing two rules.

The *sequence* of two events, E1 and E2, is a composite event denoted (E1;E2) that is signalled when E2 is signalled, provided E1 had been signalled before. Argument of this composite event is the union of of the arguments of E1 and E2. For example, the event (Update_Inventory(I, A, S); EOT) will be signalled at the end of the transaction in which the inventory is updated.

The *closure* operator is useful for the rules that should be fired only once per transaction, provided a given event was signalled at least once during the transaction, rather than firing every time the event was signalled. The closure of event E is denoted E^*, where E has been signalled an arbitrary number of times in a transaction. E^*'s arguments are accumulated from the E's arguments. This operator is especially useful for integrity checking.

2.4.6 Interbase

Interbase ([INT90a],[INT90b]) is a relational database system developed by Interbase Software Corporation. In Interbase a trigger is a piece of code that executes a specific action when a record in a relation is stored, modified, or erased. Thus Interbase supports only database events. Each database operation has a time indicator, pre or post, associated with it which specifies whether to fire the trigger before or after the operation. There are two predefined context variables, **old** refers to the record that is being modified or deleted and **new** refers to the new record that is being modified or inserted. It is also possible to define and post arbitrary events by using triggers. To model an event the trigger must do the following:

- Identify the event by specifying a unique string.
- Specify the conditions under which the event manager will notify interested programs that the event has occurred.

For example, the following trigger posts an event that indicates a 1% change in the stock price:

```
define trigger stock_event for stocks
        post modify 0:
                if new.price / old.price > 1.01 or
                   new.price / old.price < .99
               post new.company;
end_trigger;
```

This trigger checks to see if the change in stock prices exceeds 1%. When the change does exceed 1%, the trigger posts the event. This consists of passing the name contained in its argument to the Interbase *event manager*. The event manager then checks to see if the name is in the *event table*, which lists the events in which active programs have registered interest.

2.4.7 OSAM*

The Object-oriented Semantic Association Model (OSAM*) ([LAM89], [SU88]) is a system developed at University of Florida. In the rule definition language of OSAM* [SIN90], an event consists of a trigger-operation and a trigger-time. The trigger-operation specifies the operation that causes that event to occur and can be a data manipulation operation such as InsertObject, InsertInstance, DeleteObject, DeleteInstance, Update and Retrieve or can be a user-defined operation. Trigger-time specifies when to trigger the rule and can be one of 'before', 'after' or 'parallel'. More than one operation can be specified in the event part and thus it supports disjunction of events.

An event related to an operation can be signalled 'before' an operation is executed in a transaction, immediately 'after' an operation is executed in a transaction or 'after' all the operations are executed in a transaction. By default, all the events are signalled after all the operations are executed in the transaction. Parameters of these events are all the 'affected' data items.

2.4.8 Postgres

Postgres [STO87] is the successor to INGRES relational database system. In the second Postgres rule system [STO90] (PRS2), only database events (retrieve, replace, delete and append) are provided as primitive events. Semantics of these events is that at the time an individual tuple is accessed, updated, inserted or deleted, there is a CURRENT tuple (for retrieves, replaces, and deletes) and a NEW tuple (for replaces and appends). If the specified event is true for the CURRENT tuple then the condition is evaluated. Postgres does not differentiate between "begin" and "end" of the operation.

Besides these database events, Postgres provides a restricted "disjunction" of events. Keywords new (i.e. replace or append) or *old* (i.e. delete or replace) can appear in place of retrieve, replace, delete or append.

An event can be specified as follows:

ON event TO object

where object is either

a relation name

or

relation.column, ..., relation.column.

2.4.9 Starburst

Starburst [WID90] is a relational database system developed by IBM Almaden Research Center. It has the notion of operation block, which is a stream of data manipulation operations such as insert, delete and update, that is executed indivisibly. Execution of an operation block causes a state change of the database called *transition.* A transition predicate is a list of *basic transition predicates*, which specify particular operations on particular tables. For example, basic transition predicate **inserted into t** (where **t** is the table name) holds with respect to any transition effect that identifies one or more tuples in table t. The syntax for transition predicate is:

trans-pred ::= basic-trans-pred
| basic-trans-pred **or** trans-pred

basic-tran-pred ::= **inserted into** table
| **deleted from** table
| **updated** table.column
| **updated** table

Starburst supports *set oriented* production rules. After a given transition, those rules whose transition predicate holds with respect to the effect of the transition are triggered.

Thus starburst provides database events that occur as an overall effect of a transition, rather than individual effects of the component operations. For example, if a tuple is updated by several operations and is then deleted, only deletion is considered, since this is the net effect of the transition. It also provides *disjunction* as an event operator.

In starburst, parameters of an event are one or more logical tables corresponding to the associated transition predicate.

- If **inserted into t** is the basic transition predicate, then logical table **inserted t** holds the tuples inserted into **t** by the transition that triggered the rule.
- If **deleted from t** is the basic transition predicate, then logical table **deleted t** holds the tuples deleted from **t** by the transition that triggered the rule.
- If **updated t.c** is the basic transition predicate, then logical table **old updated t.c** refers to the tuples of table **t** in the previous state of the database in which column **c** was updated by the transition that triggered the rule; logical table **new updated t.c** refers to the current values of the same tuples.
- If **updated t** is the basic transition predicate, then logical table **old updated t** refers to the tuples of table **t** in the previous state of the database that were updated by the transition that triggered the rule; logical table **new updated t** refers to the current values of the same tuples.

2.4.10 Sybase

Sybase is a relational database system that is developed by the Sybase Inc. In Sybase [SYB87] a trigger is a special kind of stored procedure that goes into effect when a database operation is carried out. One or more of three operations, insert, update and delete, can be specified on a particular table. At the *end* of the execution of these operations, corresponding events are signalled.

Parameters of **delete** event is logical table *deleted* that contains the rows removed from the specified table. Parameters of **insert** event is a logical table *inserted* that holds the rows added to the specified table. Parameters of **update** event are the *updated* rows: rows before the changes are added to the logical table *deleted* and rows after the changes are added to the logical table *inserted*.

In the above sections we have briefly described some approaches to situation monitoring in different branches of computer science. Though active constructs supported in fields other than database systems satisfy the requirements of specific application areas, they are not suitable for modeling database applications.

Analysis of current active database systems reveals that these systems support events with varying degrees of capabilities. Most of the systems provide only database events and *disjunction* as the only operator in a restricted fashion. In HiPAC, although the need for temporal events and contingency plans was established, no formal approach was given. Ariel supports temporal events, however its specification is not very expressive.

In the next chapter, we discuss the need for various events and present our approach to event specification.

CHAPTER 3
PROBLEM STATEMENT AND OUR APPROACH

3.1 Requirements

Event-condition-action or ECA rules impart capabilities to a database system to react automatically and asynchronously to state changes and/or to the occurrences of other events. When an event occurs (is signalled), the condition is evaluated and if the condition is satisfied, the action is executed. This requires monitoring various events of interest that may cause a rule to fire. Before an event can be recognized, it should be specified and its semantics should be defined in the system. Commonly used data definition and manipulation languages (DDL and DML) do not support specification of events required in active database system. Some systems allow assertions, invariants, and integrity constraints to be specified which can be viewed as special provisions for supporting some requirements in the absence of a general purpose event specification language. In addition to the traditional notion of events which correspond to database operations, such as access, insert, delete and update, various other events need to be supported.

Many database and nondatabase applications have requirements intimately related to time. These applications need to execute different actions at prespecified time. For instance, an automatic door locking system in an office may be required to lock all the doors of the building exactly at 5 p.m. and to open them again next day at 7 a.m. except holidays. Similarly, in a garage or in an elevator it may be required to close the door at a prespecified time after the event "door_open". As a traditional database example consider the case of a bank database. In this database it may be necessary to prevent any transaction after 5 p.m. until 8 a.m. next day. As another example, a patient's database in a hospital, may be required to prompt the operator to enter patient's recent bloodpressure and temperature one hour after the drugs have been given. Besides these applications many other applications need to deal with time. This makes it necessary to define time events formally and to provide a concrete method for their specification and detection.

In several applications, databases are a part of the system and a number of events are detected by application programs outside the purview of the database system. For example, in network management, sampling of multiplexors and modems are done by the system and their results are maintained by the database system. For example, in a process control system, any changes in the parameters of the process such as temperature or pressure are detected by the application programs and then are signalled to the database system for processing. System should provide facilities for specifying and managing such user or application generated signals.

Many other time-constrained applications require to execute an alternate action if the action can not be realized within a prespecified time. For example, in a process control system it may be necessary to open the boiler valve automatically within five seconds after the pressure exceeds maximum limit and if this could not be done evacuation of boiler room and other appropriate action should be performed as contingency plans. System should provide a method for handling such contingency plans required in time-constrained applications.

3.2 Motivation for this Work

Primarily, rules comprising of event, condition, and action specifications (ECA rules) were proposed as a uniform paradigm for accomplishing active databases. Of the three components, event specification is perhaps the least understood although database events and few operators were defined in HiPAC and the need for temporal and other types of events were also recognized. Conditions and actions are better understood as they correspond to queries and transactions respectively.

Expressiveness of the event specification language and the efficiency of event detection determines the power of an active system. In other words, support for condition-action needs to be complemented adequately with an expressive event specification language. A myopic view of supporting a small number of events (e.g. database events such as insert, delete and modify) will severely limit the ability to model complex applications where temporal and other events need to be combined with database events. This thesis attempts to define an event specification language that not only supports primitive events of several types but also operators for constructing complex events needed for the applications discussed earlier.

3.3 Our Approach

Although it is possible to provide a predefined set of events in any system, this approach will make the system restrictive and difficult to extend. Instead, if we take the approach of identifying primitive events and providing a language for constructing composite events, we would overcome both the limitations. In fact, this approach is similar to one taken in programming languages by providing a small set of primitive data types and constructs for building complex types as desired by the user. Composite events are formed by using a set of primitive events and event operators. For example, in a process control system, the operator may have to be notified whenever one or more parameters of the process change their values. This event can be specified by using the 'Or' operator. To meet the requirements of several advanced and nontraditional applications, we have provided various operators such as Or, All, Sequence, Periodic, and Aperiodic.

In the next chapter, we describe an event hierarchy, define various operators and give examples to show their need.

CHAPTER 4
EVENT SPECIFICATION LANGUAGE

In this chapter we present an event specification language *SNOOP*. We start with a formal definition of event and describe various events and event operators provided in our language. We discuss parameter computation for composite events and propose a method for implementing contingency plans.

4.1 Definition of Event

To conceive the notion of events, it is helpful to imagine a continuous time line. The line is divided into a number of segments where each segment equals the granularity of the time scale of a given abstraction. Thus from the system's point of view time is discrete rather than continuous. A time scale is the marking scheme on the time line. For example, a calendar can be considered as a time scale with years, months, days, hours etc. as time units and second as its least count. Some other marking scheme may have the least count, equal to the fraction of a second.

An event is something that happens at a point in time (e.g. flight 456 takes off, insert tuple t1 into R). As none of the things are instantaneous, an event is simply an occurrence that is fast compared to the granularity of the time scale chosen. Occurrences of an event are expressed using the granularity of the time scale and hence, are mapped to distinct points on the linear time line. For example, any event that occurs at some point t_o which lies between two consecutive seconds t_{s_n} and $t_{s_{n+1}}$ will be extrapolated to $t_{s_{n+1}}$. Most often an event such as 'depressing a button' has multiple occurrences corresponding to the points (on the time line) at which the button was depressed. Hence occurrences of events for a given event type can be characterized as a relation which maps event type to its occurrences. Alternately, it can also be expressed as a boolean function on event instance and the time of occurrence.

It is important to distinguish conditions from events. A condition is a boolean function of object values, such as 'the temperature is above 50°F'. A condition is valid over an interval of time. For example, 'the temperature remains above 50°F from 1 p.m. to 6 p.m. on Aug 1, 1991'. 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 a person sits in the car (event), if the temperature is above 80°F (condition), then turn on air-conditioner (action which leads to another state).

To indicate common structure and behavior, events can be organized into an hierarchy of event classes. Each event is an instance of its class and has a unique occurrence. The term event is typically used to refer both an event instance and an event class. However these usage does not pose any problem and the intent is usually clear from the context. We will use 'E' to represent an event class and 'e' to represent an event instance to make the distinction where necessary. Each event class has formal parameter members associated with it, that are instantiated and passed as actual parameters when the event occurs. Event-type and time of occurrence are implicit parameters of all events. For example, in a relational database system, 'INSERT' is an event class and each `insert` operation to relation R is an event instance of this class, which may have parameters such as the relation name and inserted tuples in addition to implicit parameters.

An event has three aspects: *occurrence*, which is the process of instantiating its formal parameters; *detection*, which is the process of collecting and recording its actual parameters and *signalling*, which is the process of sending an interrupt to the condition evaluator indicating that the event has occurred and providing the actual parameters.

The language developed in this thesis is application and model independent. However, for concreteness we use the relational model to show the computation of parameters for composite events.

4.2 Event Classification

Figure 4.1 shows the event hierarchy we have developed to depict the event classification. Events can be broadly classified into: i) primitive events - events which act as the basic building blocks and for each of which an detector need to be associated and embedded in the system and ii) composite events - events those are formed using a set of operators, primitive events and composite events constructed so far.

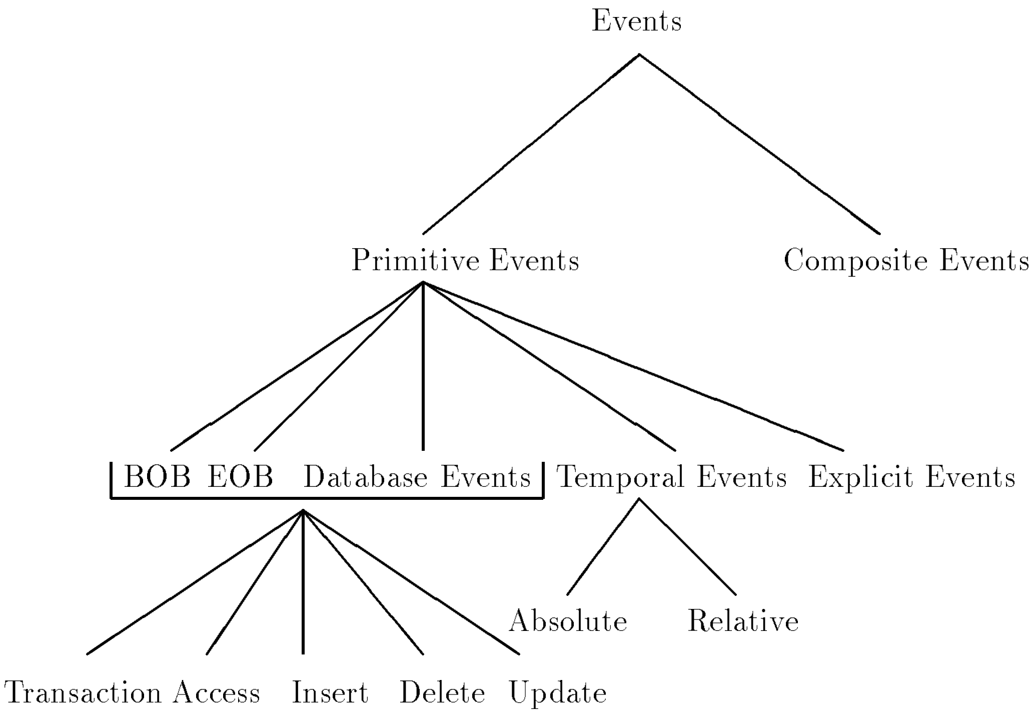


Figure 4.1. Event Classification

4.2.1 Primitive Events

Primitive events are further classified into database events, begin of block (BOB) and end of block (EOB), temporal events and explicit events. BOB and EOB are events that occur at the beginning and at the end of the execution of a block of statements. Database events are related to database operations such as transaction, access, insert, update and delete. Temporal events are related to time and are of two types: absolute events are absolute point in time, for example, at 5p.m. and relative events, that are defined with respect to an explicit reference point, for example, 5 seconds after event E where E is either a primitive event or a composite event. Explicit events are the events which are signalled along with their parameters by the user or other application programs and are managed by the system. Each event (primitive or otherwise) has a well-defined set of parameter attributes that are instantiated for each occurrence of that event.

4.2.1.1 Begin (BOB) and end (EOB) of block

BOB and EOB are the events that correspond to the beginning and end of a block of statements. Block is a generalization of functions, procedures, and transactions. It can be a common database operation such as access, insert, delete and modify or a transaction or any arbitrary procedure. Parameters of these events are event-type, time-stamp, block-id and the parameters of the block itself. Typically, parameters of BOB include the input parameters and parameters of EOB include the output parameters of that block. Block-id is the parameter that identifies the type of operation such as insert, delete or the name of the procedure. In case of a transaction its value identifies the transaction uniquely.

Most of the operations including database operations such as insert, delete etc. are not instantaneous as they take a finite amount of time for their execution. Therefore they can not be treated as a single event or the event would correspond to the completion of the operation. Using BOB and EOB, we can define two events, begin of the operation, and end of the operation for each operation or for any arbitrary sequence of operations (e.g. transaction) that has well-defined bounds. This corresponds to *pre* and *post* of other systems such as OSAM*. The term "begin" connotes a point before the first operation after entering the block and "end" corresponds to the point after the last operation but before leaving the block or sequence of operations. This concept of *begin* and *end* holds in mapping an event to a point as an event must map to a single point on the time line. Whenever an attempt is made to execute an operation, corresponding *begin* event occurs. On successful execution of the operation, corresponding *end* event occurs, however in case of an abort the operation is terminated without the occurrence of an *end* event. We further stipulate that whenever an operation is specified as an event without *begin* or *end* prefix, *end* is assumed.

4.2.1.2 Database events

Access. Operation occurs when a data item is accessed. Parameters of *begin access* event include the relation name and the predicate. The parameters of *end access* event include the relation name and the accessed tuples.

Insert. Operation occurs when a data item is inserted into the database. The parameters of *begin insert* and *end insert* are the relation name and the data items being inserted.

Delete. Operation occurs when a data item is deleted from the database. Parameters of the *begin delete* are the relation name and the deletion predicate. Parameters of *end delete* event include the relation name and the data items being deleted.

Update. Operation occurs when a data item is updated in the database. Conceptually it is possible to simulate an update operation by an insert following a delete. However, "update" is a primitive event because during update, the data item is not deleted from the database but is overwritten. Thus the update operation implies that the object-id (whether value_based or not) remains unchanged. However, [1] it can be replaced by an *atomic* sequence of deletion followed by an insertion. For example, a block of statements, [Begin of Transaction, delete, insert, End of transaction] simulates an update. Arguments of the *begin update* and *end update* event include the relation name and the updated data items.

4.2.1.3 Temporal events

A temporal event is an instance of the temporal event class. To define a temporal event, we need to define the corresponding point on the time line. Because it's an infinite line, any point on it can only be defined with respect to a predefined reference point. Calendars have been used for time specification and have an implicit reference point. It is possible to have calendars with different reference points and granularity (i.e. timing units such as years, months etc.). A time string comprises of timing units and is used for specifying a definite point on the line. For example, (h:m:s)mm/dd/yy is a time string. Again, any form of time string can be used that conforms to the calendar rules and is *predefined* in the system. For example, (h:m:s)dd/mm/yy is the time string used in Europe.

[1] In case of value-based object-ids which are accessible to the user and hence can be reused in the newly formed tuple.

Another approach to time specification is to provide a reference point as well as the offset in terms of timing units. The reference point should be chosen as any well defined point on the time line. For example, it can be the moment when a certain event takes place or it may be taken as the moment when the calendar starts. *Offset* is the amount of time in terms of timing units and is also represented by a time string. Thus a time string may either specify an absolute value of time as a point on the line or it may represent an amount of time as offset. To resolve this ambiguity we adopt the notation in which we enclose the time string representing a point on the line within angular brackets <> and the time string representing interval or offset within square brackets []. The method of providing reference point and offset is more general.

Depending upon the method of specification, temporal events have been classified into absolute event and relative event.

Absolute event. Absolute temporal event is defined with respect to the implicit reference point of the calendar system. It is the absolute value of time and is represented as: $< timestring >$. It is possible to choose any calendar and any format for the time string. Thus in the western calendar an absolute event may be denoted as the time string $< (h : m : s)mm/dd/yy >$. For example, start of the 21st century may be specified as $< (00 : 00 : 00)01/01/2001 >$. Here it should be noted that this time string isn't the same as offset. Offset would be $[(00 : 00 : 00)00/00/2000]$, because in a $< timestring >$ "yy", "mm" and "dd" fields denote the current year, month and day in progression and "(h:m:s)" shows the amount of time that has elapsed on that particular day. Absolute temporal events are useful in many situation. For example, in a university database all students may be required to pay the fees by a certain date and in case of the delay, a late fees may have to be charged in their account. Such a database would contain a rule which will be triggered by the absolute event and will scan all the student records checking for dues. As the action part, it will charge a certain late fees in the account of those students who missed the deadline. In the specification of an absolute temporal event, a field in the time string may contain a wild card, which is denoted by '-' and represents *any* valid value of that field. This is especially useful in the specification of a periodic event discussed latter. In addition, a wild card can be used as a method for increasing the granularity. An empty field is not allowed in the time string representing an absolute event in order to avoid ambiguities. Parameters of an absolute temporal event are event-type and time of occurrence.

The notion of current time is specified by the variable 'now'. Based on this we define **definite events** as events those are bound to occur between the interval t such that $now \leq t < t'$ where t' is a point on the time line. A definite event indicates that its occurrence is guaranteed within a finite amount of time. For example, 'end of the 21^{st} century is a definite event whereas 'the end of the *cold war* is not a definite event.

Absolute events that have the value less than 'now' are assumed to have occurred and are not detected and processed. Only meaningful absolute events whose values are greater than 'now' at the specification or activation time (i.e. definite events) are detected and processed.

Relative event. Relative event too corresponds to a unique point on the time line but in this case reference point and offset both are explicitly specified as part of the event. Reference point may be any event including an absolute temporal event. Offset may be defined in any time scale known to the system which may be different than the calendar being used for absolute events. The syntax for relative event is *event* + [*timestring*]. In the representation of an offset, an empty field in the time string, results in a zero value of that field. As an example of relative event, consider a patient database in a hospital which prompts the operator for the patient's blood pressure one hour after the medications are given. Similarly, an airline reservation system may have the rule which requires the passengers to purchase the ticket within 24 hours after the booking has been made or the booking may be canceled.

A relative event has been defined to be a primitive event as it's the most natural representation of a point on the temporal line. Parameters of a relative event E = E1 + *time* consists of event-type and the time of occurrence. Parameters of E1 which acts as an explicit reference point *does not* constitute the parameters of E, and E is treated strictly as a temporal event. [2]

4.2.1.4 Explicit events

Explicit events are events whose parameters are explicitly specified and supplied by the users or application programs. For consistency, we insist that parameters of explicit event must include event-type and the time of occurrence. Explicit events are assumed to be detected outside of the system but are signalled to the system along with their parameters. Thus these events are accepted and processed but are not detected by the system. For example, tornado_watch(location,expected_time, wind_speed) is an explicit event which will be signalled by a satellite center to all the systems.

Similarly, when executing the action part of a rule, it may be necessary to signal an event that conveys the effect of action to other rules. For convenience we call them rule_events which are essentially explicit events because they are raised while

[2] There are situations in which parameters of E are relevant and need to be included in the parameter relations of E. For such cases, the 'sequence' operator (defined later) need to be used: E = E1; (E1 + *time*).

executing the action part of a rule and their parameters are explicitly specified by the user. Rule_events are especially useful in implementing contingency plans which are discussed later.

4.2.2 Composite Events

Though primitive events discussed so far, alone are sufficient for several classes of applications, it is not possible to specify complex events needed to model many other applications. For example, requirement that event e1 be followed by event e2, can not be specified by using only the primitive events. We define a composite event as an event expression formed by using a set of primitive events, event operators, and composite events constructed so far. Figure 4.2 shows the event operators provided in SNOOP. Below, we describe each of these operators and give precise semantics for computing the parameters of composite events. In this thesis, we have restricted ourselves to a relational system and hence relational operators are used for computing the parameters of composite events.

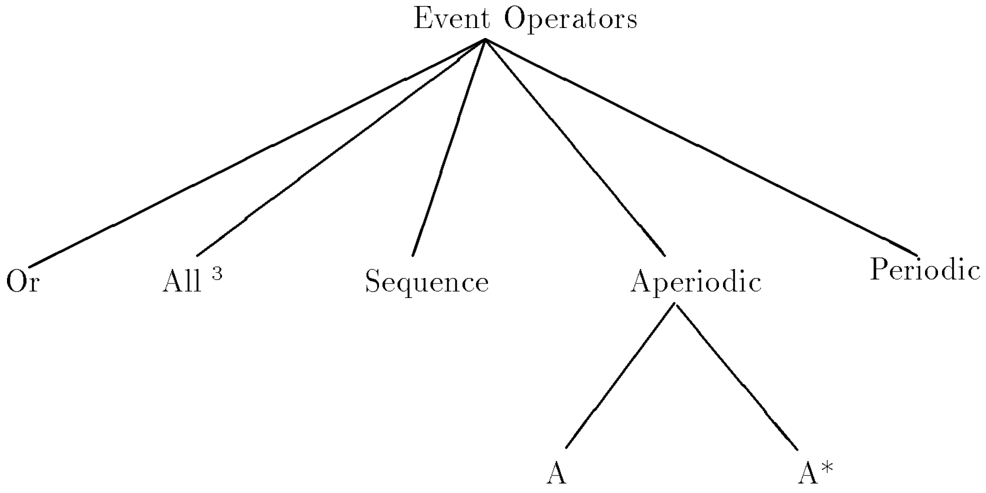


Figure 4.2. Event Operators

[3] It is possible to simulate this operator by combining 'Or' and 'Sequence'.

4.2.2.1 Disjunction

Disjunction of two events, denoted by $E1 \vee E2$ is signalled when E1 or E2 is signalled. Arguments of the composite event thus formed, are the outerunion of E1's arguments and E2's arguments. This operator is useful when the occurrence of one or more (i.e.inclusive-Or) events out of a set of events may cause a rule to fire. For example in a process control system, if pressure, temperature or humidity exceeds the limit, the operator may have to be notified. In this case 'Or' constructor provides a succinct solution to the problem.

In this thesis, we have assumed that no two events can occur simultaneously. However this assumption no longer holds in a multiprocessor environment. Moreover, explicit events can also occur simultaneously. In case of concurrent occurrences, it is necessary to gather the parameters of all the events and then signal the composite event only once. How long the system should wait for the availability of the parameters of these event is not clear and further research is required in this area.

4.2.2.2 Sequence

Sequence of two events E1 and E2 is a composite event, which is denoted by *E1;E2*, and is signalled when E2 is signalled provided E1 has already been signalled. This means that the time of occurrence of E1 is guaranteed to be less than the time of occurrence of E2. Parameters of the composite events are the outerunion of the parameters of E1 and E2. This constructor is necessary when a predefined order has to be imposed over the occurrences of events. For example, in a network management system, if corrective actions need to be taken on the occurrence of the composite event "link_break_down; diagnosis_completed".

It's is possible that after the occurrence of E1, E2 does not occur at all. This will lead the system into a state of infinite waiting, expecting E2 to occur. To avoid this situation, it is desirable that every ';' expression be followed by a definite event such as EOT (end of transaction) or any absolute temporal event.

4.2.2.3 Conjunction

A composite event E, denoted by All(E1,E2), is signalled when both the events are signalled disregarding the order of occurrence. Parameters of E are defined as the outerunion of the parameters of both the events. The order of occurrences of component events can easily be derived from this parameter relation by sorting the time of occurrences of these events. Conjunction of events is useful in the situations where firing of a rule depends upon the occurrence of a set of events. It is possible to simulate an 'All' condition with the combination of ';' and 'Or'. For example; (E1,E2) can be written as $((E1; E2) \vee (E2; E1))$. However this will result in long event expressions or it will be necessary to write many rules and save the arguments of each event in the database so that they can be used after the occurrences of all the events. As an example, consider the cooling system of an atomic reactor that has three pumps for redundancy that circulates the coolant. When all three pumps fail, the reactor need to be shut down immediately. In some situation, it may desired to use this event in disjunction with a definite event to avoid infinite waiting. For example; (E1,E2)∨EOT.

4.2.2.4 Aperiodic event

A. An aperiodic event E is represented by A(E1,E2,E1') where E1, E2 and E1' are event expressions. E is signalled each time E2 occurs during the interval defined by the occurrences of E1 and E1' and its parameters are passed as the parameters of E. This operator is useful when the occurrence of an event has to be monitored within a predefined time interval. For example, an application that requires any change in the temperature of an object to be signalled from the beginning of an experiment to the end of that experiment can easily be modeled by this operator.

A*. There are situations when a given event is signalled more than once during a given interval (e.g. transaction) and rather than firing the rule every time the event was signalled, we want the rule to be fired only once. To meet this requirement, we provide an operator A^*(E1,E2,E1') that accumulates the effect of E2 and signals E only once at a point that corresponds to the occurrence of E1'. Between the occurrences of E1 and E1', it is necessary that the parameters of the event E2 should be accumulated so that they can be passed to the condition evaluator. Therefore the parameters of the composite event E = A^*(E1,E2,E1') are the union of the parameters of all the occurrences of E2. If E2 does not occur between E1 and E1', parameters of E that correspond to E2 are assigned NULL. This constructor is especially useful in integrity checking, because we want to check the constraints only at the end of the transaction.

4.2.2.5 Periodic event

We define periodic event as an event p that repeats itself within a constant and finite amount of time. Though, p may be instances of any event class [4], only temporal event class makes sense from the event detection perspective. From this definition, it is obvious that a periodic event may be represented by a triplet, comprised of an event E, the time period t after which an instance p of temporal event class takes place and a terminating event E' that marks the end of the periodic event. Events E and E' may be any well defined events including absolute temporal event and interval t can be specified as a time string. Periodic event has many applications. For example, In

[4] This is already accomplished in aperiodic event.

a bank database, it may be required to print the summary of all the transactions of each customer at the end of the month.

Periodic event p is specified as P(E1,t,E1') where E1 and E1' are event expressions and t is the time period in some time scale . It is important to note that t is a *constant* and can not be replaced by a wild card because this will result in continuous occurrences of p. Terminating event E' marks the end of the time interval, that begins at the occurrence of E1, and over which occurrences of instances p have to be monitored. For example, beginning of each month of 1992 is a periodic event and can be specified as- P((00:00:00)01/01/1992, (00:00:00)01//, (-:-:-)12/-/1992). In this particular case, use of wild card may result in a succinct representation of the same event - ((00:00:00)-/01/1992). However, not all periodic events can be specified using wild card and in that case the user will have to resort to the former method of specification. For example,in an inventory database it is required to place a new order for some material after each 17 days, in the year 1993, until the project is over. Also assume that at the end of the year if the project is not over new tenders would have to be invited. This event is a periodic event with two terminating events and can be specified as: $P((00:00:00)01/01/1993,(00:00:00)00/17/00,(23:59:59)12/31/1993 \vee End\_of\_Project)$ where End_of_Project is an *explicit* event. Parameters of a periodic event is same as the arguments of any other temporal event.

In both the events, aperiodic and periodic, a time interval is marked by the occurrences of two different events E1 and E1' which can either be a primitive event or a composite event. To form a meaningful expression, time of occurrence of E1 must always be less than the time of occurrence of E1'. We also provide two special symbol $-\infty$ and $+\infty$ which can be used to mark begin and end of an interval that begins at time $-\infty$ and ends at time $+\infty$ on the time line respectively.

4.3 Timing Constraints and Contingency Plans

There are applications that require to take certain actions if the expected event does not take place within a specified time. For example, in a public telephone booth, after the connection is established, an event "deposit coin" is monitored and if the required amount is not deposited within a prespecified time, the action "terminate connection" may have to be executed. Similarly, in a distributed database system, after the data has been transferred to the remote cite, sender must wait for the acknowledgement and should retransmit the same data in case the receiver does not acknowledge the receipt of the data within a certain time. If 'S' is the event that corresponds to the transmission of data and 'Ack' is the event that represents the receipt of the acknowledgement from the remote site then the event and condition parts for the rule that retransmit the data if the acknowledgement does not arrives within 10 seconds after the transmission can be expressed by using A* operator as follows.

Event: A*(S, Ack, S+[(10::)//])

Cond : Count('Ack' in P_rule-id) = 0

The event of this rule occurs 10 seconds after the occurrence of event 'Ack'. We assume that parameters of the event of a rule are recorded in a relation P_rule-id. "Count" is a function that returns the number of tuples corresponding to the occurrences of event 'Ack' in the parameter relation of A* event. Above rule executes its action if the acknowledgement is not received within the specified interval.

Many other time-constrained applications require to monitor specified situations and execute corresponding actions, subjected to some timing constraints. Contingency plans are alternate actions which should be executed if the specified action can not be realized within the specified time limit. Contingency plans can easily be implemented using the 'A*' operator. The rule compiler converts a rule that has contingency plans into two rules. The first rule contains the same event, condition and action as that of the main rule and signals a "Rule_Event" explicitly at the beginning and at the end of the "action". In this case, rule_events are essentially BOB and EOB where block is the "action" part of the rule. These two rule_events form the event part of the second rule. For example, a system monitoring the boiler status may have the following rule [5]-

```
rule7 : ON pressure_change
           IF pressure > dangerous
                 open valve
                    WITHIN 00:00:25
              ELSE
                    sound alarm
```

Rule compiler will converts this rule into following two rules :

```
rule7a : ON pressure_change
           IF pressure > dangerous
                    signal(Rule_Event1)
                    open valve
                    signal(Rule_Event2)
rule7b : On pressure_change; A*(Rule_Event1, Rule_Event2,
                             Rule_Event1 + [(25::)//])
           IF Count (Rule_Event2 in P_rule7b) = 0
                    Terminate(action7a)
                    Sound Alarm
```

[5] In this example, we have assumed 'Immediate Coupling Mode', further research is necessary to explore the implications of other coupling modes on this method of implementing contingency plans.

Rule7 is triggered by an event 'pressure change'. It evaluates the condition to check whether the pressure exceeds a predefined limit. If the condition is evaluated to be true, it tries to open the valve within 25 seconds and if it can not be done an alarm is sounded. As can be seen, combined effect of rule 7a and 7b is equivalent to the effect of rule7. The function 'count' used in the condition part of rule7b, is a special function that gives the number of tuples in the parameter relation corresponding to the occurrence of the specified event.

4.4 Grammar for SNOOP

To provide an unambiguous evaluation method for event expressions, we have given a default precedence for event operators, which can be changed by the user if desired by changing the priorities associated with them. Priorities of the operators are used by the detection algorithm for building the *event tree* discussed later. Following is the grammar for the event specification language.

E1 ::= E1 $\vee$ E2 | E2

E2 ::= E2 ; E3 | E3

E3 ::= All(E4) | E5

E4 ::= E4, E5 | E5

E5 ::= | A(E1,E1,E1)

$A^*(E1, E1, E1)$

| P(E1,[time string],E1)

$|< timestring >$

$| (E1) + [timestring]$

| *External Event*

| BOB

| EOB

| L:(E1) /* Where L is a label */

| (E1)

Event expressions generated by above BNF are left associative. An event expression can either be a primitive event or a composite event. Label association with an event expression identifies that primitive or composite event. Rule-id is considered as the label for the top-level event expression. Labels are useful in the parameter computation of composite events. They are the identifications for the subexpressions in a complex event expression. They act as the *event type* for that composite event and for each such event a 'time of occurrence' attribute is inserted in the parameter relation. Without label association it would be impossible to distinguish the occurrence of that composite event and to access the time of its occurrence in the parameter relation.

The above BNF can be used to generate complex event expressions. The computation of the parameters of corresponding composite events depends upon the *context* in which the events are detected. In the next chapter, we discuss various contexts for detecting composite events and give the design of a composite event detector.

CHAPTER 5
EVENT DETECTION

This chapter discusses issues related to event detection. In section 5.1 we define detection of events and discuss the need for different contexts for detecting primitive and composite events. In section 5.2, we give examples to demonstrate the expressiveness of SNOOP in different application domains. Finally, in section 5.3, we propose the architecture of a composite event detector and give algorithms for detecting composite events.

5.1 Detection

Detection of an event is defined as the process of collecting and recording the parameters of the event including the time of occurrence. This process may take finite amount of time and as a consequence, an event may not be detected immediately at the point of occurrence but rather a point after the occurrence. Assuming the discretization of time line, if T_o is the point of occurrence and T_d is the point of detection of an event, then $T_e = T_d - T_o$ is called the *latency* of event detection. Value of this delay, in a sense, quantifies the *timeliness* of the detection of an event. In order to avoid the risk of missing the response window, it is desirable to keep the value of T_e as small as possible. In time-constrained applications (that have contingency plans discussed in chapter4), T_e plays a critical role.

Event detector is a component of the system that records the occurrences of events by collecting their parameters and makes this information available to condition evaluator.

Process of event detection has two aspects: primitive event detection and composite event detection. Following subsections discuss each of them individually.

5.1.1 Primitive Event Detection

In order to detect each primitive event (except, explicit events that are detected by users or application programs) a separate detection mechanism need to be embedded within the system. For example, to detect temporal events a separate event detector need to be composed that uses the system clock to detect the specified temporal events. Similarly, to detect database events, detection mechanism need to be embedded within the part of the system which is responsible for reading and writing the data on the disk.

Database events that are related to database operations are commonly used in various applications. Typically, a single database operation affects many tuples. For example, a single insert operation may insert five tuples. Moreover, these operations are always executed in a transaction that usually comprise of many database operations. Thus there are three possibilities for signalling database events: **tuple-oriented**, **operation-oriented** and **set-oriented**.

- In the **'tuple-oriented'**approach, the event related to a database operation is signalled at the tuple level. For example, if an insert operation inserts five tuples then the event *insert* will be signalled five times corresponding to each tuple and parameters will be passed separately. For example, consider an interactive data entry application which allows the operator to enter student's records. This application may have a rule that informs the operator regarding any obvious error in the inserted tuple immediately at the end of its insertion. This rule need to be triggered at the tuple level. In addition, this approach may be useful for enhancing concurrency.

- In the **'operation-oriented'** approach, signalling of an event is based upon the corresponding operation. Thus, the event 'insert' in the above example will be signalled only once with accumulated parameters (i.e. with all the inserted tuples). We take this approach to model database events.

- The **'set-oriented'** approach [WID90] has the notion of a block of statements in which a block may contain many operations and at the end of the execution of a block, events corresponding to each operations are signalled with accumulated parameters. In this approach operations that cancel out each other (e.g. insert followed by a delete) are not counted at all and same operations (e.g. more than one inserts to same relation) are merged into one operation. Also deletion of a tuple followed by the insertion of a new tuple is not considered as an update to the original tuple. This approach is especially useful in integrity checking as we want to check integrity constraints at the end of the transaction. We support this approach by providing the A* operator.

5.1.2 Composite Event Detection

Composite events are expressed in SNOOP by event expressions that are generated by using the BNF given in chapter 4. A composite event may be comprised of several primitive events. Because the occurrences of these events may not be simultaneous, the event detector need to record the occurrences of each event by saving its parameters so that they can be used to evaluate the event expression representing the composite event. We adopt a notation E(E1,E2,...En) to represent an event expression E, where $E_{i=1..n}$ are its component primitive events. Many occurrences of components primitive events propound different possibilities for detecting composite events that involve operators 'All' and 'Sequence'. To illustrate this, consider following two events:

A = All(E1, E2); E3

B = $E1 \vee E2 \vee E3$ and

Figure 5.1 shows the occurrences of different instances of event E1, E2 and E3. Event A is signalled at the point where atleast one instance of all the three events has occurred with E3 as the last occurrence. Event B is signalled each time when an instance of any of the three events E1, E2 or E3 occurs. Parameters of event A include parameters of all the three events E1, E2 and E3 and depend upon their

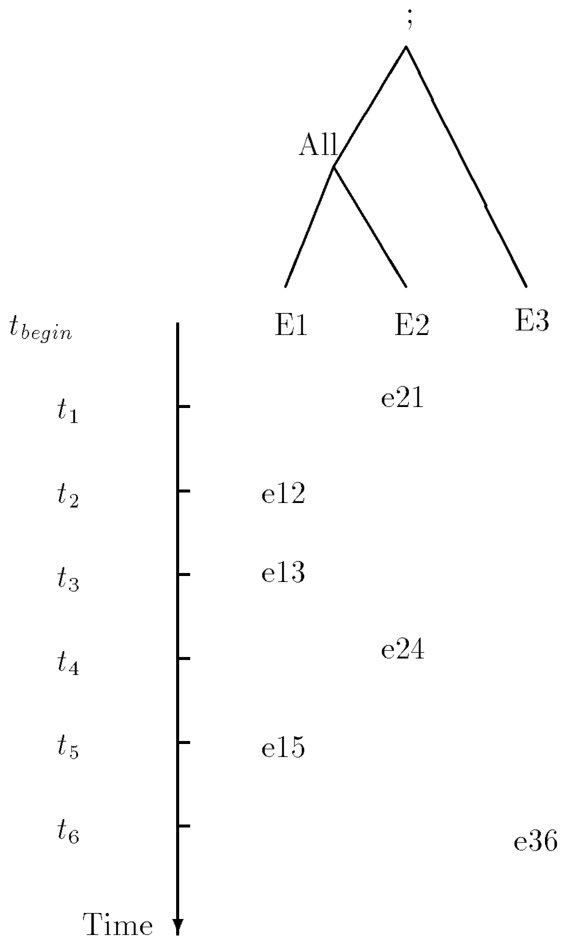


Figure 5.1.

instances being taken into account. However, this problem does not exist in the computation of parameters of event B because it involves only one occurrence. In the following paragraphs, we explore different possibilities for combining the instances of component events and based on that define various contexts for composite event detection.

The event detector begins monitoring a composite event E, represented by the event expression E(E1,E2,...En), at a point t_{begin} on the time line, which corresponds to the time when the event E is last *activated*. Event detector maintains parameter relations for all component events in the composite event. At time t_{begin}, which is specific to each composite event, parameter relation of all component events E1, E2, ... En are cleared. Whenever an event E_i occurs, its parameters are recorded in its parameter relation. In case of multiple occurrences of the same event, its parameters are accumulated in the same parameter relation. When the last component event needed to evaluate the event expression occurs, the expression is evaluated by using the event algebra and if true, the composite event is signalled along with its parameters. Also when an event is *deactivated*, all unused occurrences of its component events are discarded. We define three contexts for combining the instances of component events in a composite event.

- In **recent context**, most recent occurrence of each E_i is taken into account for computing the parameters of E. When E is evaluated to be true, the composite event is signalled and all the entries in the parameter relations of component events are deleted. For example, in *recent* context, parameters of event A will be computed by using event instances e13, e24 and e36.

- In **chronicle context**, instances of component events are taken into account in a chronological order for computing the parameters of the composite event E.

When E is signalled, its parameters are computed by using the oldest instance of each component event and then parameters of these instances are deleted from the corresponding relations. For example, parameters of event A in this context will be computed by using event instances e12, e21 and e36.

- In **cumulative context**, parameters of E include the parameters of multiple occurrences of each E_i. Whenever E is signalled, all the entries in the parameter relations associated with each E_i are deleted. For example, parameters of event A will include all the instances of each event.

A context can be specified only to the top-level event expression. Allowing different context specification to the sub-expressions of an event expression will entail further complexities and future work is required in this area.

These contexts affect the semantics of operators that involve *multiple instances* of component events such as 'All' and 'Sequence'. A context does not have any significance to other operators.

5.2 Examples

In this section, we give examples of various events and discuss parameter computation. Context is assumed to be 'recent' unless specified otherwise. We assume that the name of the parameter relation of the event of a rule is the rule-id of that rule and T_id denotes the transaction_id attribute in the parameter relation.

Example 1.

- Suppose a rule is triggered when a tuple is inserted in the relation R. This event (operation-oriented approach) can be specified as:

  ON Begin Insert to R

  This will have the parameter relation schema as:

(BOB, Time, T_id, Insert, STUDENT, inserted tuple)

Where 'Insert' is the block-id.

- Above rule will be fired each time an insert operation takes place. If it is required to fire the rule only once at the end of the transaction (but before commit i.e. set-oriented approach) with accumulated parameters then the event can be expressed as: A*(BOT, I, EOT)

  Where BOT = Begin of the transaction
  I = Insert to relation R and
  EOT = End of the transaction

  The event will be signalled only once at the end of the transaction but before commit. Parameters of this composite event will be the union of the parameters of all the occurrences of 'Insert' event. In addition, it will also have its own event_type and the time of occurrence attributes.

Example 2. Consider a hospital database that has a rule which prompts the operator to enter the patient's bloodpressure 30 minutes after the medications are given. Event for this rule can be specified as:

ON (Given_Medicin to patient_id) + [(:30:)//]

Parameters of this event will be (Temporal, Time), where 'Temporal' is the event_type.

Example 3. In this example, we consider an application that requires to take certain action according to the variations in the IBM stock price.

- If the application requires to sample the IBM stock every 30 minutes and plot the price trend every day. In addition, whenever the price crosses a predefined number , it needs to inform the manager. Event for this rule can be written as:

P(< (08::)-/-/->, [(:30:)//)],< (17::)-/-/->)

The event will be signalled after each 30 minutes. In the action part, new price will be plotted against the time and if the point goes beyond a certain line the manager will be informed perhaps for selling or buying the stocks. Parameters of this event will be (Temporal, Time), where 'Temporal' is the event_type.

- Suppose the same application is also required to plot the price trend of Xerox stocks. Further assume that any change in this stock price is explicitly signalled by an agent (explicit event) and need to be plotted at that time. Also as in the above example, whenever the price crosses a predefined number it needs to inform the manager. Such event can be expressed as:

  A(< (08::)-/-/->, price_change, < (17::)-/-/- >)

  Parameters of this event are the parameters of the event 'price_change'.

- Suppose it is required to plot the price trend of the Xerox stock and analyze it at the end of every day. In this situation, the rule need not to be triggered each time the price changes, rather we want the event to be signalled only once at the end of the day with accumulated parameters. This can expressed as follows:

  A*(<(08::)-/-/- >, price_change, < (17::)-/-/- >)

  This event will be signalled only once with accumulated parameters. Parameters of this event will be the union of the parameters of all the occurrences of the middle event.

<u>Example 4.</u> Consider a process control system which has a rule that alarms the operator whenever temperature, pressure or humidity crosses a threshold. Event part of this rule can specified as:

(Pressure_change)$\vee$ (Temperature_change) $\vee$ (humidity_change)

This composite event will be signalled whenever any of these event occurs. Parameters of this event will be the outerunion of the parameters of all three events.

Example 5. Consider the problem of distributed database managers (dm1, dm2,.... dm_n). Whenever a database manager makes an update to the database, it informs other managers about the update and then goes into an inactive state until all the managers acknowledge the receipt of this information. If s_{ij} is the event that represents the transmission of information from i^{th} manager to j^{th} manager and a_{ij} is the event that represents the receipt of the acknowledgement by i^{th} manager from j^{th} manager then the event part of a rule that brings an inactive manager dm1 into an active state can be expressed as follows:

All$((s_{12}; a_{12}), (s_{13}; a_{13}), ...(s_{1n}, a_{1n}))$

This composite event will take place when manager dm1 receives the last acknowledgement. Parameters of this event will be the outerunion of all the component primitive events.

Example 6. Suppose in a reactor, the coolant is circulated by three pumps A, B, and C. The system need to be shut down if 2 out of the three pumps fail. Event for such rule can be written as:

$All(fail_A, fail_B) \vee All(fail_A, fail_C) \vee All(fail_B, fail_C)$

Example 7. To show the need of 'cumulative context', consider a software development environment. Suppose the software contains three modules and each module is worked on by a separate development team. A set of bugs to be fixed in the next release is assigned to each development team. After fixing its set of bugs, corresponding team signals the event ($fixed\_module_n$, Time, bug_number, description). Even after fixing these bugs, teams continue to signal removal of any other possible bugs until all three teams accomplish the debugging of their corresponding set. At that time, they are informed to stop signalling further bug fixes and the release is made. This event can be expressed as:

$$\text{All}((fixed\_module_1, ...), (fixed\_module_2, ...),$$
$$(fixed\_module_3, ...))$$

Figure 5.2 shows a possible parameter relation of this event. It should be noted that this event can not be expressed by using A* operator.

Event_Type	Time	Event_Type	Time	Bug_Number	Description
Rule_id	-	$fixed\_module_2$	-	1	-
				2	-
				3	-
		$fixed\_module_1$	-	1	-
				2	-
		$fixed\_module_2$	-	4	-
		$fixed\_module_3$	-	1	-
				2	-
				3	-
				4	-

Figure 5.2. Parameter Relation

5.3 A Composite Event Detector

Architecture of the event detector is shown in the figure 5.4. Event detector accepts rule definitions as input and creates appropriate structures which is used in the actual detection process. When a primitive event occurs, event detector records its occurrence and evaluates event expressions those contain this event as a component. When an expression is evaluated to be true, corresponding event is signalled and its parameters are passed to the condition evaluator.

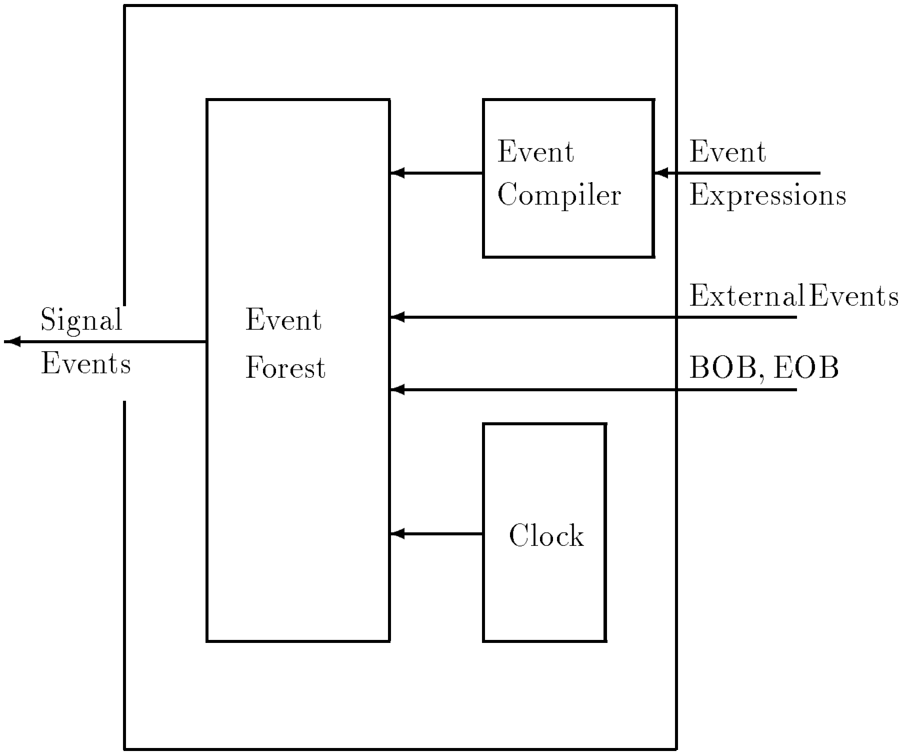


Figure 5.3. Composite Event Detector

5.3.1 Event Compiler

Event Compiler takes rule definitions as input and converts the event expressions contained in them into an appropriate tree structure which is used in the actual detection process. If the rule contains an 'else' part, it constructs corresponding rule_events and creates the other rule. It also accepts definitions of explicit events to form the corresponding parameter relation. Algorithm for building the graph is as follows:

```
Algorithm Compile_Event
begin
      Read rule_definition;
      if the rule has a contingency plan
        Define trees (i.e. nodes) corresponding to rule_event1 and rule_event2 in the forest;
        Modify 'action' part of the original rule;
        Create Rule_b;
      endif;
      For all event expressions
      begin
            build_tree;
            Mark the top node with rule-id;
            merge it in the event_forest;
      end;
end.
```

5.3.2 Event Graph

Event graph comprises of non-terminal nodes (N-nodes), terminal nodes (T-nodes) and links. Each node represent either a primitive event or a composite event. N-nodes represent composite events and may have several incoming and several outgoing links. T-nodes represent primitive events and have one incoming and several outgoing links. Links are the means through which stimulation is sent to the nodes. When a primitive event occurs, it stimulates the terminal node that represents this event. This in turn ripples stimulation to all the nodes attached to it via outgoing links. When a node is stimulated, it executes the corresponding procedure which evaluates the corresponding operator and if true, sends stimulation to all other nodes connected to it by pushing the parameters of the events. If the node is marked with the rule-id, it also signals the corresponding event to the condition evaluator. Following is the algorithm for detecting composite events.

```
Algorithm Detection
begin
    On the occurrence of a primitive event
    begin
        store its parameter in the corresponding terminal node ‘t’
        in the forest;
        call proc_stimulate(t);
    end;
end.
Procedure proc_stimulate(n)
begin
    For all rule-ids attached to the node ‘n’
```

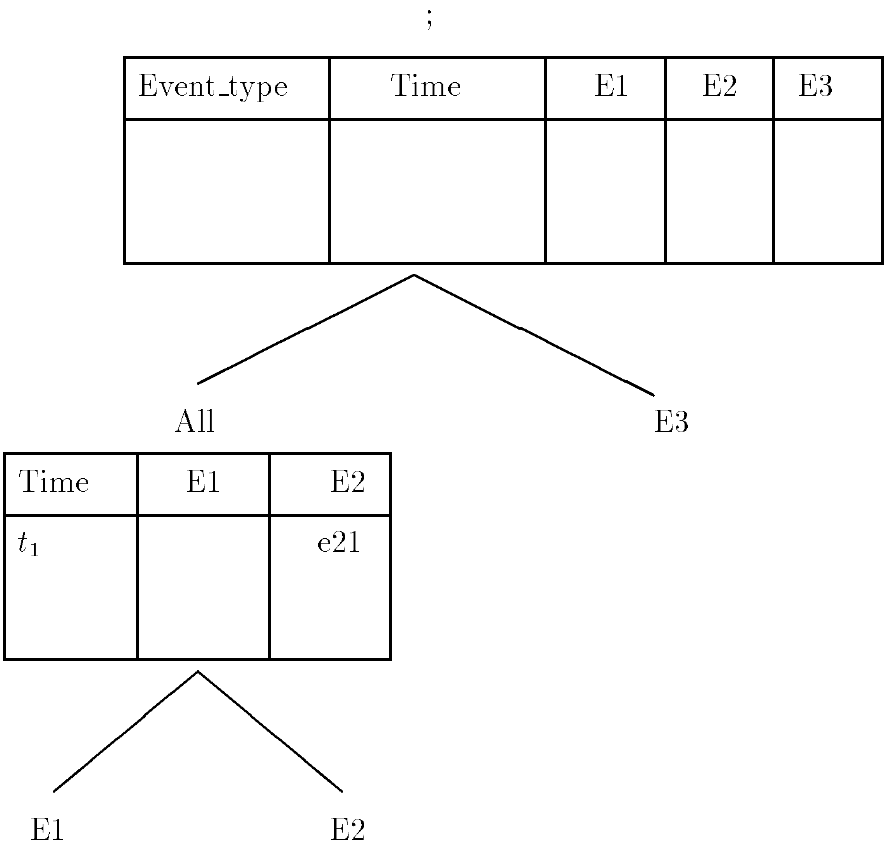
```
            signal event;
        For all outgoing links i
        begin
            push parameters in the $node_i$ connected by link i
            call proc_N_node($node_i$);
        end;
        Delete pushed entries in the parameter relation of $node_i$;
end.

Procedure proc_N_node
begin
  case $node_i$ of
        'Or' : store the parameters in the relation;
               proc_stimulate(This_node);
        'All': store the parameters in the relation;
               If all the component events have occurred
                     proc_stimulate(This_node);
        'A' : If *left event*
                mark it as occurred;
              else if *middle event* and *left event* is marked as occurred
                if (context = Chronicle)
                  call proc_stimulate(This_node);
                else if (context = recent)
                  save current parameters as the most recent occurrence;
                else if (context = cumulative)
                  accumulate parameters
               endif
             else if right event
               if (context = Recent or Cumulative)
                 call proc_stimulate(This_node);
               mark left and right events as not occurred.
             endif
        'A*' : If left event
               mark it as occurred;
             else if middle event and left event is marked as occurred
               accumulate parameters;
             else if right event
               call proc_stimulate(This_node)
               mark left and right events as not occurred.
             endif
        'P' : If left event
               mark it as occurred;
             else if middle event and left event is marked as occurred
               call proc_stimulate(This_node);
             else if right event
               mark left and right events as not occurred.
             endif
  end case;
end.
```

As an example, consider the following composite event:

E = All(E1, E2); E3

Detection of this event in *recent context* is shown in the following figures. Event e_{ij} indicates the occurrence of event type E_i at time t_j.

In the next chapter, we conclude this thesis by comparing our approach with the systems studied in chapter 2 and discuss areas for future work.



Event_type	Time	E1	E2	E3

Time	E1	E2
t_1		e21

Figure 5.4. Occurrence of event e31

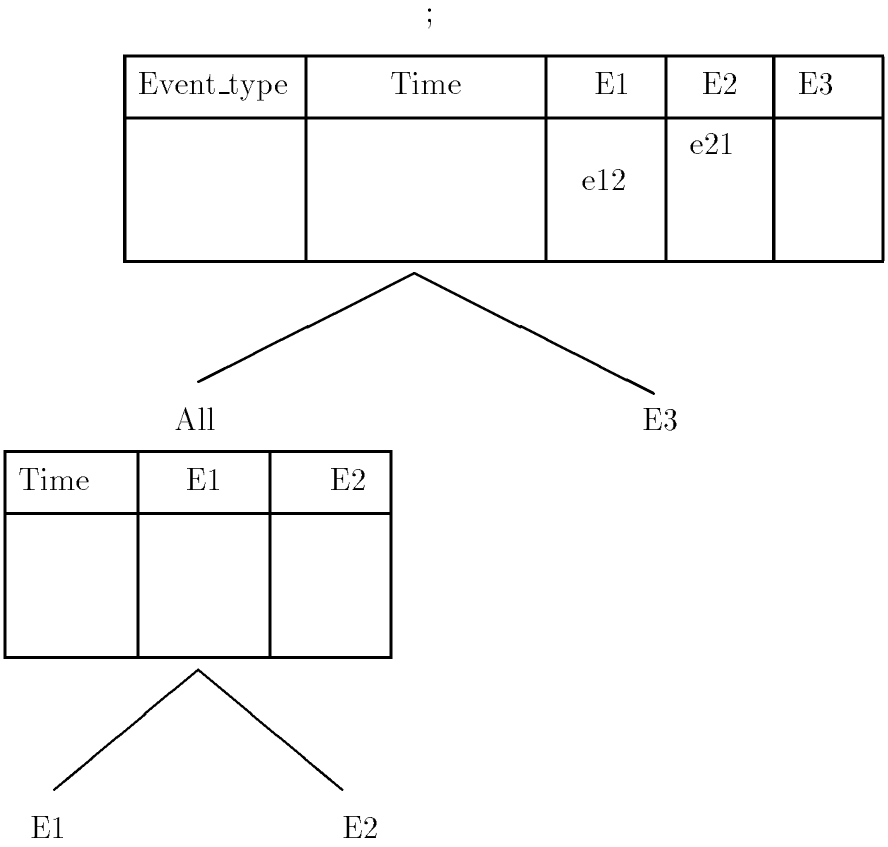


Figure 5.5. Occurrence of event e12

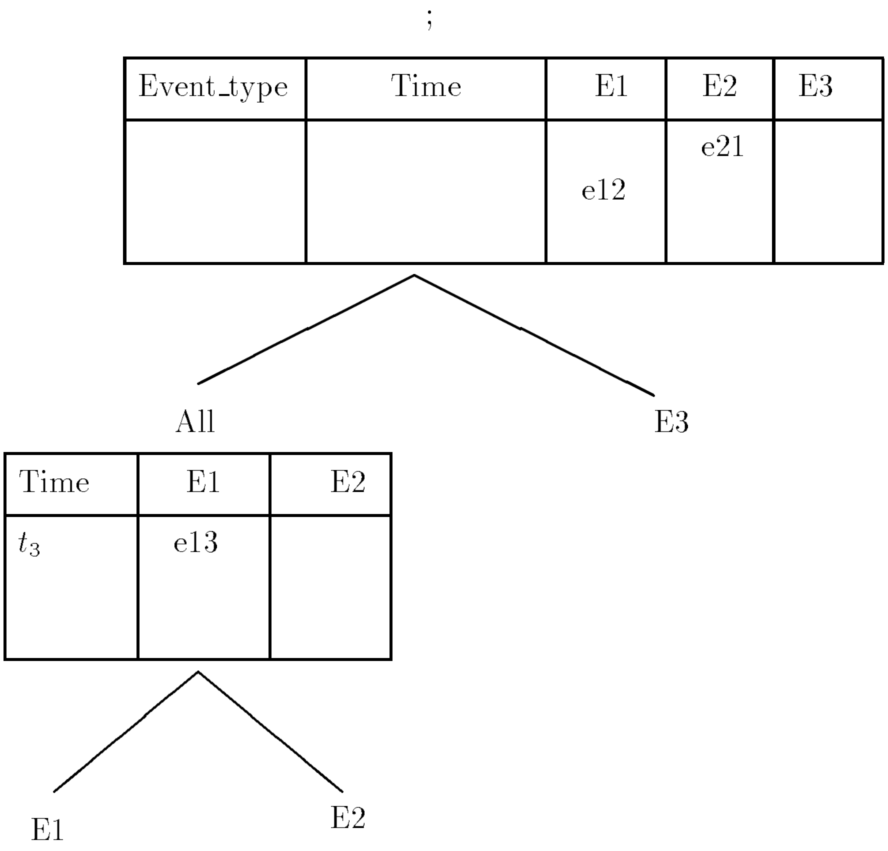


Figure 5.6. Occurrence of event e13

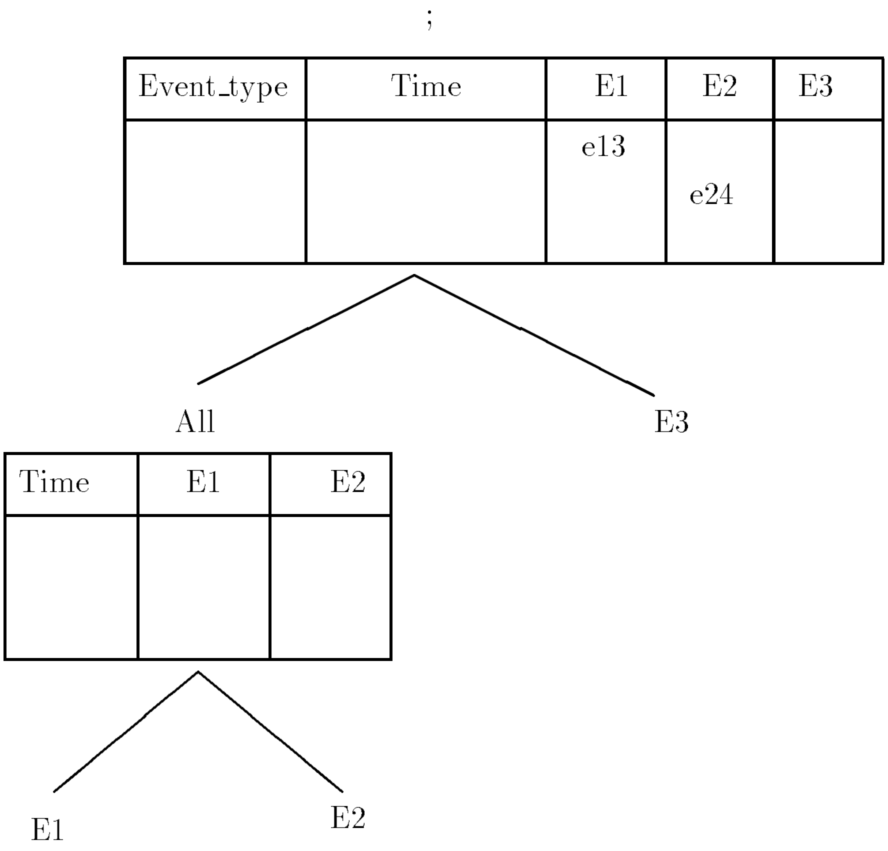
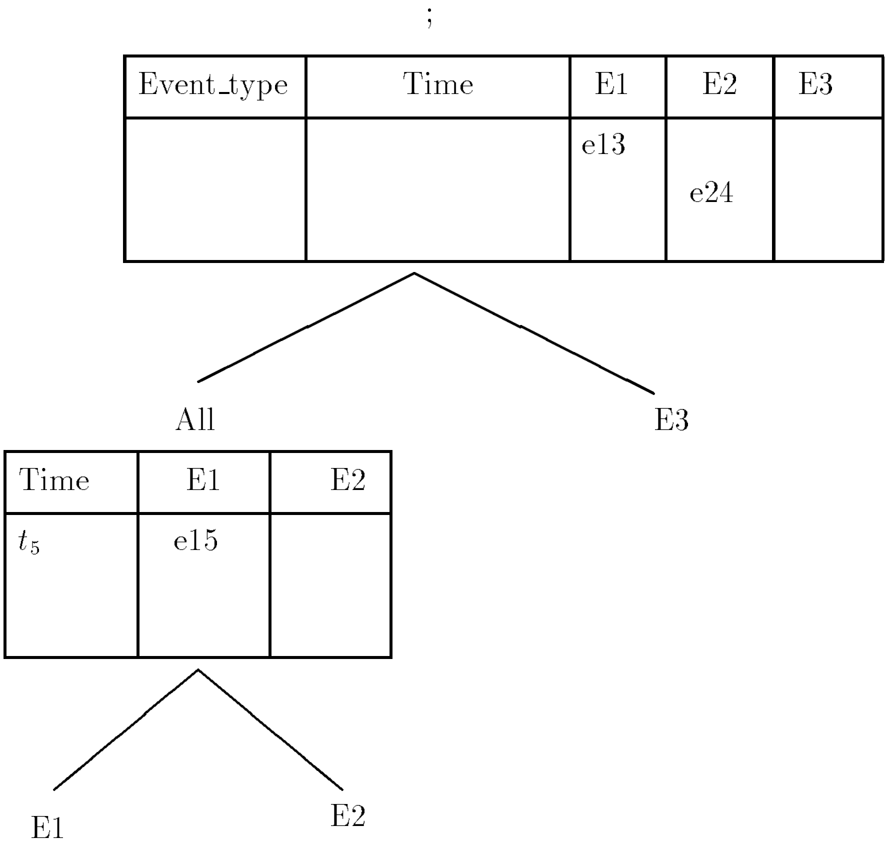


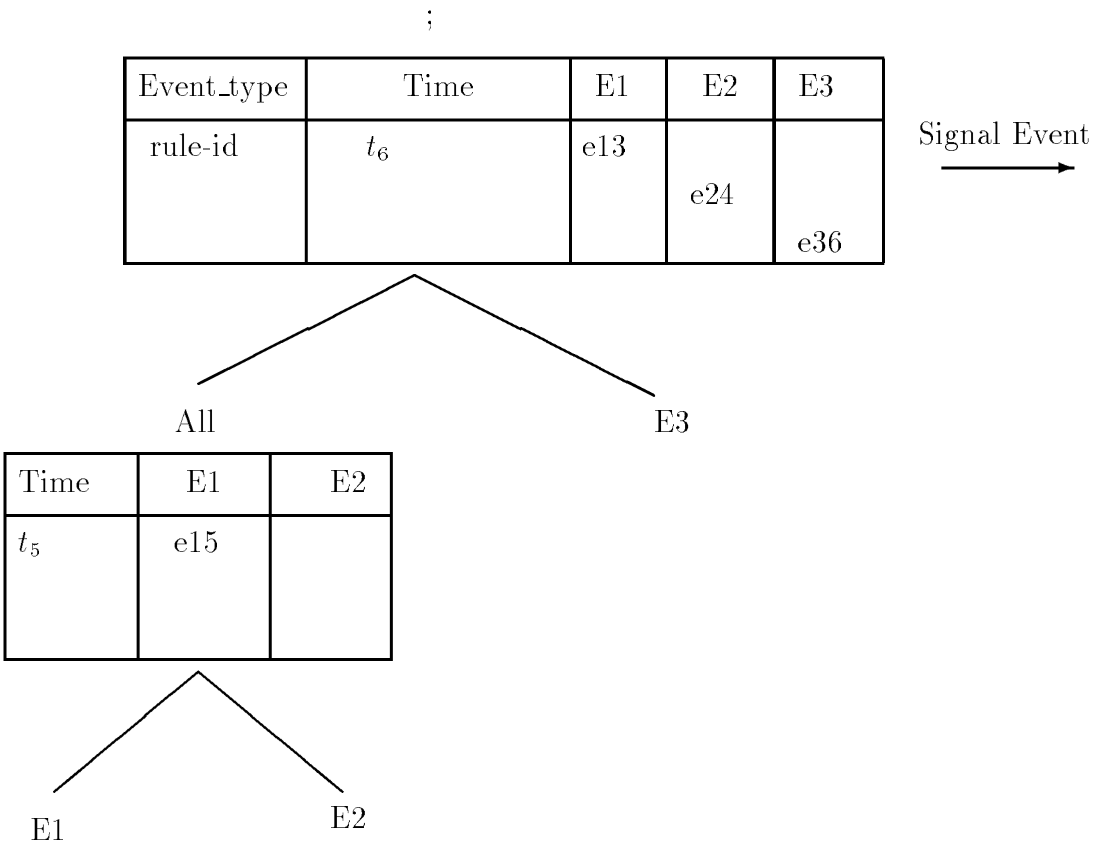
Figure 5.7. Occurrence of event e34



Event_type	Time	E1	E2	E3
		e13	e24	

Time	E1	E2
t_5	e15	

Figure 5.8. Occurrence of event e15



Event_type	Time	E1	E2	E3
rule-id	t_6	e13		
			e24	
				e36

Time	E1	E2
t_5	e15	

Figure 5.9. Occurrence of event e26

CHAPTER 6
CONCLUSION AND FUTURE RESEARCH

6.1 Conclusion

In this thesis we have presented an event specification language that is rich enough to express variety of events needed to model non-traditional applications. We have defined several events including database, temporal, explicit, periodic and aperiodic events and have proposed a technique for implementing contingency plans required in a time-constrained environment. We have identified various contexts for detecting composite events and have presented the design of a composite event detector. The contribution of the thesis are:

- Classification of events into an event hierarchy.
- Generalization of BOT and EOT of HiPAC into BOB and EOB that can be applied to any grouping of statements.
- Temporal events and their semantics.
- Several event operators for constructing composite events and their semantics.
- Parameter computation within the relational model as a relation.
- A mechanism for dealing with contingency plans without stepping out of the framework proposed in this thesis.
- Identification of various contexts for computing parameters of composite events.
- Algorithms for detecting composite events.

Our language subsumes the events supported in the extant active database systems including Ariel, ETM, HiPAC, Interbase, OSAM*, Postgres, Starburst and Sybase. Table 6.1 compares SNOOP with the events supported in the systems studied in chapter 2.

6.2 Future Research

Defining an event specification language and devising methods for detecting complex events is an abstruse task. This research represents an initial attempt in this direction. Following paragraphs mention areas those remain unaddressed in our work but are critical to the task at hand.

- Efficient detection of primitive events is crucial to the performance of ECA rule mechanism. Design and specification of primitive event detectors that need to be embedded within the database system is an important task. We feel that considerable amount of work is required to incorporate these event detectors within different database models including relational and object-oriented.

- Our approach to event specification and parameter computation is restricted to a relational model. We have defined parameters of each event as relation and have used relational concepts for computing the parameters of composite events. Further work is necessary to tailor the theory for object-oriented environment. Especially, parameters of database events need to be redefined.

- While discussing *inclusive-or* operator we have ignored the possibility of simultaneous occurrence of events. However, explicit events may take place with other primitive events at the same time. Moreover, simultaneous occurrences are natural in a multiprocessor environment. In this situation two courses of action exist: either to fire the rule as soon as the parameters of a primitive event becomes available to the composite event detector or to wait until the parameters of all the events, those occurred simultaneously, become available and then fire the rule with the combined parameters of all events.

- We feel that specification of temporal events need to be made more expressive. Some applications such as an automatic door locking system in an office which is required to open the doors everyday at 8a.m. and then close them at 5p.m. except *week-ends* are not well served by SNOOP. Similarly, operator 'All' should be made more expressive. For example, consider a cooling system that comprises of six pumps and if four out of six pumps fail the event has to be signalled as a catastrophic condition to shut off the entire system. This composite event can not be succinctly specified in our language.

- Currently, we do not allow *context* association to subexpressions of an event expression. However, this may be useful for modeling complex applications. Allowing context association to subexpression will require modifications in the detection algorithm.

- While discussing contingency plans, we have assumed the 'immediate' coupling mode. Implications of other modes, such as deferred and detached, are yet to be investigated.

- A rich SQL-like rule language that can hide the coarse rule structure, and can provide a user-friendly interface will certainly be beneficial. This higher level language must address *parameter passing* to the condition and action and should provide higher level constructs to ease the task of end users. For example, in the specification of periodic event, it would be useful to provide a modified operator P^* that would be signalled at the end of the specified interval and would contain an optional action as its fourth parameter. This would be useful in many situation. For instance, consider a financial database where it is required to sample the IBM stock price each hour during an interval 8a.m. to 5p.m. and then plot the graph at 5p.m. Such applications would be elegantly served by this operator.

REFERENCES

[Bac88] Maurice J. Bach. *The Design of the Unix Operating System.* Prentice-Hall International, Inc., Englewood Cliffs, N.J., 1988.

[C$^+$89] Sharma Chakravarthy et al. HiPAC: A Research Project in Active, Time Constrained Database Management. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, July 1989.

[CG86] Phillip Caverly and Philip Goldstein. *Introduction to Ada: A Top-Down Approach for Programmers.* Brooks/Cole Publishing Company, Monterey, California, 1986.

[Cha89] Sharma Chakravarthy. Rule management and Evaluation: An Active DBMS Perspective. In *Special issue of ACM Sigmod Record on rule processing in databases*, September 1989.

[Cha90] Sharma Chakravarthy. Overview of HiPAC: A Research Project on Active, Time-Constrained Database Management. Technical Report UF-CIS TR-90-18, University of Florida, Gainesville, Florida, May 1990.

[CNGM90] Sharma Chakravarthy, S. B. Navathe, S. Garg, and A. Mishra, D. and-Sharma. An Evaluation of Active DBMS Developments. Technical Report UF-CIS TR-90-23, University of Florida, Gainesville, Florida, 1990.

[CRM80] Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1980.

[DKM86] Klaus R. Dittrich, Angelika M. Kotz, and Jutta A. Mulle. An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases. In *ACM SIGMOD*, September 1986.

[EC75] K. P. Eswaran and D. D. Chamberlain. Functional Specifications of a Subsystem for Data Base Integrity. In *Proceedings of 1st International Conference on Very Large Data Bases*, Sept. 1975.

[Esw76] K. P. Eswaran. Specifications, Implementations, and Interactions of a Trigger Subsystem in an Integrated Database System. Technical Report RJ1820, IBM, San Jose CA, 1976.

[Geh84] Narain Gehani. *Ada, An Adavanced Introduction including Reference Manual For The Ada Programming Language.* Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1984.

[Han89] Eric N. Hanson. An Initial Report on The Design of Ariel: A DBMS With an Integrated Production Rule System. In *Special issue of ACM Sigmod Record on rule processing in databases*, September 1989.

[Hug79] Joan K. Hughes. *PL/I Structured Programming.* John Wiley & Sons, 1979.

[Int90a] Interbase Software Corporation, 209 Burlington Road, Bedford, MA 01730. *Data Definition Guide*, February 1990.

[Int90b] Interbase Software Corporation, 209 Burlington Road, Bedford, MA 01730. *DDL Reference*, February 1990.

[KDM88] A. Kotz, K. Dittrich, and J. Mulle. Supporting Semantic Rules by a Generalized Event/Trigger Mechanism. In *Proceedings International Conference on Extending Database Technology*, Venice, March 1988.

[LSA89] H. Lam, S.Y.W. Su, and A. M. Alashqur. Integrating the Concepts and Techniques of Semantic Modeling and the Object-Oriented Paradigms. In *Proceedings of the 13th International Computer Software and Applications Conference*, Orlando, Florida, September 1989.

[Oll78] William T. Olle. *The Codasyl Approach to Data Base Management.* John Wiley & Sons, 1978.

[PS85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley Publishing Company, second edition, 1985.

[S$^+$90] M. Stonebraker et al. On Rules, Procedures, Caching and Views in Database Systems. In *ACM SIGMOD*, May 1990.

[Sin90] Madhulika Singh. Transaction Oriented Rule Processing in An Object-Oriented Knowledge Base Management System. Master's thesis, University of Florida, 1990.

[SKL88] S. Y. W. Su, V. Krishnamurthy, and H. Lam. An object-oriented semantic association model. In *Artificial Intelligence: Manufacturing Theory and Practice.* The Institute of Industrial Engineers, 1988.

[SR87] M. Stonebraker and Lawrence Rowe. The Postgres Papers. Technical Report UCB/ERL M86/85, Dept. of Electrical Engineering and Computer Science, Univ. of California, University of California, Berkeley, CA94720, June 1987.

[Syb87] Sybase, Inc., Sybase, Inc. Berkeley, CA 94710. *Transact-SQL User's Guide*, 1987.

[Tur86] Raymond W. Turner. *Operating Systems, Design and Implementations.* Macmillan Publishing Company, 1986.

[WB79] Patrick Henry Winston and Richard Henry Brown. *Artificial Intelligence: An MIT Perspective.* The MIT Press, Cambridge, Massachusetts, 1979.

[WF90] Jennifer Widom and Sheldon J. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *ACM Sigmod*, May 1990.

BIOGRAPHICAL SKETCH

Deepak Mishra was born on April 2, 1966, in Raipur, (M.P.), India. He received a Bachelor of Engineering degree (with distinction) in Computer Engineering from G.S. Institute of Technology and Science, Indore, India in June 1988. After his graduation, he worked as a lecturer in the Department of Computer Engineering of the institute.

He joined the Department of Computer and Information Sciences at the University of Florida in August 1989 to pursue a masters' degree and since then has worked as a research assistant in the database systems research and development center of the department. His research interests include artificial intelligence, object-oriented engineering design databases, and active database systems.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Sharma Chakravarthy, Chairman
Associate Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Shamkant B. Navathe, Cochairman
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Herman Lam
Associate Professor of Electrical Engineering

This thesis was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Master of Science.

December 1991

Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School