SIEVE: AN INTERACTIVE VISUALIZATION AND EXPLANATION TOOL FOR AN ACTIVE OODBMS

By

JUN ZHOU

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

1995

In memory of my mother

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for his excellent guidance, and for giving me an opportunity to work on this challenging topic. I am grateful to Dr. Stanley Su and Dr. Herman Lam for agreeing to serve on my supervisory committee and for their perusal of this thesis.

Thanks are due to Ms. Sharon Grant for maintaining a well administered research environment.

I will also take this opportunity to thank all the graduate students in the Sentinel group and the Database R&D Center for their help and friendship.

Last, but not the least, I thank my parents and brother for their love. Without their encouragement and endurance, this work would not have been possible. And I feel words are inadequate to expressing my thanks to the love of my life, my wife Jinning, for everything.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

LIST OF FIGURES vi

ABSTRACT viii

CHAPTERS 1

1 INTRODUCTION 1

2 RELATED WORK 6

2.1 An Application-Level GUI to Active Databases – EVE 6
2.2 System-Level GUIs to Active Databases 8
2.2.1 DEAR 9
2.2.2 SimBug 10

3 MOTIVATION 12

3.1 Application-level GUIs 12
3.2 System-level GUIs 14

4 OVERVIEW OF OPEN OODB AND SENTINEL 16

4.1 Passive Open OODB 17
4.1.1 Functional Features of Open OODB 17
4.1.2 Operational Features 20
4.2 Active Capability of the Sentinel System 23
4.2.1 Architecture 23
4.2.2 Rule Implementation 26
4.2.3 Functionality of the Sentinel Modules 27

5 DESIGNING GRAPHICAL USER INTERFACES FOR SENTINEL . . . 30

5.1 User Requirements on Data Visualization 30
5.2 MDP - An Application Level GUI 32
5.2.1 Description of the MDP Application 32
5.2.2 Design of the MDP Graphical User Interface 33
5.3 Sentinel Rule Debugger - A System-Level GUI 36
5.3.1 Background, Observations and Objective 36
5.3.2 Design Choices 43

5.3.3 Existing Sentinel Rule Visualization Tool 46
5.3.4 Interactive Rule Debugger - A Revised Design 52

6 IMPLEMENTATION 54

6.1 An Overview of X/Motif 54
6.1.1 The X Window System 54
6.1.2 Libraries for Developing X Applications 55
6.1.3 Application-System Interaction in X/Motif 57
6.2 Implementation of the MDP User Interface 57
6.3 Implementation of SIEVE 59
6.3.1 Layout and Functionality 59
6.3.2 GUI Implementation Notes 61
6.3.3 Implementation Issues on GUI-System Interactions 66
6.3.4 An Example: Stock Demo 72

7 CONCLUSION AND FUTURE WORK 76

7.1 Conclusion 76
7.2 Future Research 77

REFERENCES 79

BIOGRAPHICAL SKETCH 83

LIST OF FIGURES

1.1 Event Hierarchy 2

4.1 Functional Class Lattice of Open OODB 21
4.2 Sentinel Architecture 23
4.3 Class Hierarchy and Functionality of Sentinel 25
4.4 Sentinel Event Specification Syntax 26
4.5 Sentinel Rule Specification Syntax 26
4.6 Local and Global Event Detector Architecture 28

5.1 MDP Layout 35
5.2 Functional Modules of the Non-interactive Sentinel Rule Debugger . . 47
5.3 Layout of the Non-interactive Sentinel Rule Debugger 50

6.1 Underlying X Libraries of Sentinel Graphical User Interfaces 56
6.2 Zooming Feature of the Sentinel Rule Debugger 63
6.3 Before Pruning 65
6.4 After Pruning 65
6.5 Graphical Representation of Subtransaction States for Monochrome Displays 66
6.6 Functional Modules of the Revised Sentinel Rule Debugger 68
6.7 Global Event History 69
6.8 Control Flow at a Break Point 71

6.9 A Snapshot of Trace 75

7.1 Overall Architecture for Global Event Detection and Rule Visualization 78

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

SIEVE: AN INTERACTIVE VISUALIZATION

AND EXPLANATION TOOL FOR AN ACTIVE OODBMS By

JUN ZHOU

August 1995

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

Of late, there has been a surge of interest in active databases. Active database management systems (ADBMSs) have evolved considerably to meet the increasing requirements of non-traditional applications. The active feature can be incorporated into DBMSs by the event-condition-action (ECA) rule abstraction.

Using ECA rules in active database systems for real-life applications involves implementing, debugging, and maintaining large numbers of rules. For the effective deployment of active database systems, there is a clear need for providing flexible user interfaces to cater to different application requirements. Furthermore, experience in developing large production rule systems has amply demonstrated the need for understanding the behavior of rules especially when their execution is non-deterministic. In this circumstance, a graphical debugging and explanation facility to assist understanding of the interactions – among rules, among events, between rules and events, and between rules and database objects becomes necessary.

Research and development of graphical user interfaces (GUIs) in the realm of database management systems have lagged behind advances in other database management and visualization technologies. This is especially true in the fast-growing

area of active DBMSs. The active semantics demands appearance, design and implementation that are different from those of traditional passive DBMSs. Research of active database visualization has been a novel area.

In the context of active databases, the functionality of data visualization will switch its emphasis from traditional data navigation/browsing (data-oriented tasks) to reflecting the changes resulting from event triggering/rule firing (action-oriented tasks).

In addition to the new requirement imposed by active databases and general rules for GUI design, different user perspectives create a diversity of design choices. There is a dichotomy between application-level and system-level GUIs for active DBMSs.

The debugging of rules involves both static and run-time analysis. The importance of runtime analysis becomes more noteworthy in applications using a large number of rules.

In this thesis we concentrate on the design and implementation of both application-level and system-level GUIs for an active object-oriented DBMS – *Sentinel*, with an emphasis on the development of *SIEVE* – **S**entinel **I**nteractive **E**CA Rule **V**isualization **E**nvironment. To elaborate, this tool enables the user to visualize the event detection and rule execution details at run-time in on-line mode as well as at post-run-time on a replay basis. The architecture and layout have been re-tuned to accomplish this functionality. Moreover, the user gains a new dimension of interactivity through the debugger's two-way communication channel i.e. rather than passively receiving information delivered out of the user interface, the user is able to input changes to the rule system, modify and monitor the execution in a more active fashion. Run-time analysis is also addressed in a primitive approach of trapping execution cycles.

CHAPTER 1
INTRODUCTION

Over the last few years database management systems (DBMSs) have evolved considerably to meet the diverging requirements of the application domains. These requirements include monitoring of situations and responding to them automatically. Conventional DBMSs have been *passive* which implies that the database changes its state only on user specified operations, queries or update operations. New generation databases requires *active* capability for meeting application requirements. This new paradigm implies that a DBMS (as opposed to user/application program) would continuously monitor user-defined situations and initiate appropriate actions in response to certain database updates, occurrences of particular states or transitions of states without user or application intervention.

The feature of active capability can be incorporated into DBMSs by Event-Condition-Action *(ECA) rule* abstraction [16]. ECA Rules, in the context of an active DBMS, consists of three components: an *event*, a *condition* and an *action*. An event is an indication of an happening, mostly a state change produced by a database operation such as insert, delete and update. There also exist external/explicit and temporal events which are externally detected and signalled to the DBMS by the system or user. The condition can either be a simple or a complex query on the existing database states and the set of data objects, transitions between states of objects and even trends and historical data. Actions specify the operations to be performed when an event is detected and its associated condition evaluates to true. Once rules are specified declaratively, it is the responsibility of the DBMS to monitor the situation and trigger the rules when the event happens and the condition is satisfied.

Events are further classified into

- i) *primitive events* – events that are predefined in the system (using primitive event expressions and event modifiers), and
- ii) *composite events* – events that are formed recursively by applying a set of operators (described in the event language *Snoop* [10]) to primitive and composite events.

The event hierarchy is illustrated in Figure 1.

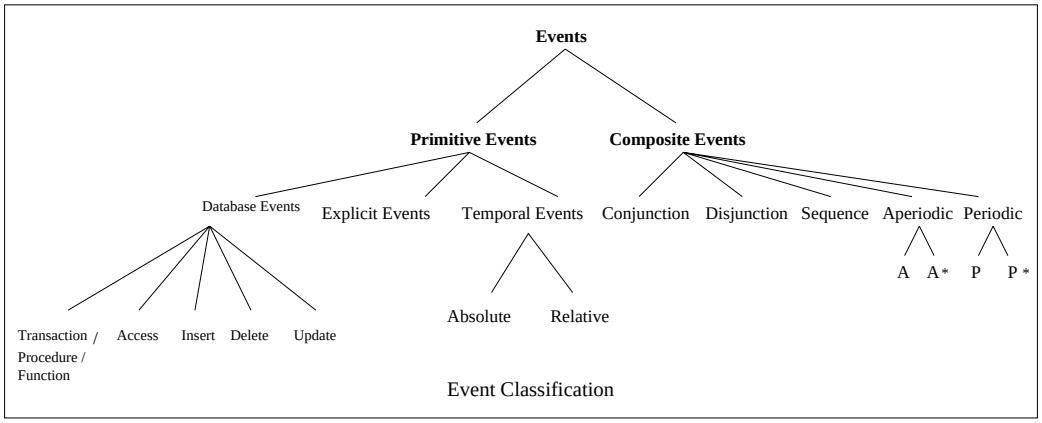

Figure 1.1. Event Hierarchy

The mechanism for primitive event detection is assumed to be available. The detection of composite events may require the detection of one or more constituent events as well as one or more occurrences of a constituent event type. The implementation of event detection has been discussed by Krishnaprasad [19].

For the effective deployment of active database systems, there is a clear need for providing flexible user interfaces to cater to different application requirements. Furthermore, a debugging and explanation facility to understand the interaction – among events, among rules, between rules and events, and between rules and database objects becomes necessary.

The visualization technology, along with hardware technology, have progressed significantly from command line interpreters to *What You See Is What You Get* (WYSIWYG) graphical user interfaces (GUIs). With the rapid evolution of windowing environments, users of DBMSs are expecting and demanding more from their user interfaces.

Unfortunately, research and development of GUIs in the realm of database management systems have lagged behind advances in other database management and visualization technologies. This is especially true in the fast-growing area of active DBMSs. The active semantics demands appearance, design and implementation that are different from those of traditional passive DBMSs. Research of active database visualization has been a novel area in which inadequate work has been done.

User interfaces to advanced DBMSs should make available to the users the entire gamut of functionalities provided by the underlying DBMSs. The user interface should take advantage of the additional semantics to provide a "semantics-driven" GUI which is simple and user friendly. Usability is an important aspect of a GUI design as the principle function of a GUI is to facilitate the human-computer dialogue. Portability is another common requirement in the context of current programming environments.

In data visualization, objects that appear before the user can be thought of as "widgets" while various behaviors are assigned to screen visuals. These behaviors might be elicited when pointed at with a picking device (typically, a mouse) via a *callback routine*. Such actions can trigger additional database interactions and visualizations, thus affording a mechanism with which to do database browsing.

The emergence of active databases has added a new dimension to the interaction between applications and DBMSs. Traditionally all these interactions have been

driven by the application. Visualization tools following this driving force are data-oriented. In such applications the user normally submits requests to the DBMS and receives responses from the system in *synchronous* or conversational fashion. In contrast, in an active database application there is no way for the user to completely predict what event(s) might occur and what action(s) the system might take as a consequence; responses are received from the DBMS in an *asynchronous* way. In many circumstances the user may not receive any reply if no rule fires. This kind of human-computer interaction distinguishes itself with traditional ones in passive database environments. We call it *action-oriented.*

In the context of active databases, the functionality of data visualization will switch its emphasis from traditional data navigation/browsing to reflecting the changes resulting from event triggering/rule firing.

In addition to the new requirement imposed by active databases and general rules for GUI design, different user perspectives may also create a diversity of design choices.

Non-database experts/end-users are more interested in application related data than system related data. In other words, they only expect to carry out requests in the application and receive results out of the system, without understanding the details of the execution process. The user interface is meant to highlight information related to the application without it being clustered with irrelevant data reflecting the states of the underlying system.

Application developers, on the other hand, need to understand the details of the system's functioning modules, trace the execution, discover existing or potential errors and correct the errors if necessary. Their interests can go well beyond a specific application's running behavior. More information of the system kernel should be presented expressively via the user interface to ensure a good understanding. For this

community of users, an emerging need in the context of an active database system is to provide an environment for debugging and visualization of rule execution. The debugging of rules involves both *static analysis* and *runtime analysis*. Static analysis checks the termination, confluence, and observable determinism characteristics [28, 1]. Runtime analysis deals with features such as the execution trace of rules, interaction between events and rules, and rules and the database. The importance of runtime analysis becomes even more clear in applications using a large number of rules.

In this thesis we concentrate on the design and implementation of both application-level and system-level GUIs for an active object-oriented DBMS named *Sentinel*.

The remainder of this thesis is structured as follows. Chapter 2 briefly describes related work in the area of graphical user interfaces to active databases. Chapter 3 presents the motivation for our design and implementation. Chapter 4 provides a description of the Sentinel system. In chapter 5 we discuss design issues and present an application-level GUI and a system-level rule debugger. Chapter 6 details the implementation and functionalities of the user interfaces with illustrative examples. Chapter 7 concludes this thesis and shows future directions.

CHAPTER 2
RELATED WORK

In this chapter we provide a summary of the graphical user interfaces for active DBMSs found in current literature. Both-application level and system-level examples are discussed. Later in this thesis, we provide a comparison of our work for Sentinel with other work discussed here.

Although active DBMS research is a fast growing area, the development of graphical user interfaces for such new-concept systems has not been given enough emphasis. A literature survey showed that most of related work on GUIs to databases was done in support of illustrating data browsing and schema editing [24, 20, 18, 6, 23, 25]. They were not only data-oriented, but also for passive databases.

Presented here are an application level-GUI and two system-level GUIs designed for active object-oriented databases.

2.1 An Application-Level GUI to Active Databases – EVE

A variety of non-traditional database applications will benefit from the active capability. These applications include process control, work-flow control, computer-aided manufacture (CIM), battle management, network management, etc. Graphical user interfaces are needed in many of these applications to help users query and visualize data, and the active capability of underlying DBMSs may play an important role in the design and functionality of the interfaces. When the following visualization tool is discussed we highlight its relationship with the system's rule mechanism and the impact of the latter on the design.

EVE [13] is a graphical browser used to visualize objects in an active OODB system, ADAM [14].

EVE aims to support *dynamic displays* based on active rules. Graphical database interfaces allow some portion of the data stored in the database to be displayed for browsing or manipulation. Changes to database objects which are depicted on screen can lead to inconsistencies between the data stored and the information displayed. Dynamic display is defined as propagating changes to the state of the database to interface where the affected data is being displayed.

An active DBMS is able to support dynamic displays if the interface is informed automatically of changes to the state of objects which it is currently displaying. The approach taken is to define a rule which fires only when changes are made to objects that are actually on screen. For example, the following rule corresponds to the objects `11#student` and `23#course`:

```
rule 1

active_object: [11#student, 23#course]
active_method: modify_method
when: after
condition: true
action: warn the interface of the change
```

active_object, *active_method* and *when* stand for an event which occurs before or after (specified by *when*) the invocation of *active_method* of object *active_object*.

This approach reduces the overhead of rule triggering because only those instances which are actually being displayed, rather than every potentially diaplayable instance will be involved.

In EVE the interface is seen as an event generator which warns the active mechanism but the reaction to this warning is handled within the database. Such reactions can be expressed by another rule which is triggered by external events (e.g. clicking

on the *next* button) and whose action updates the *active_object* item in rule 1. The second rule is described as follows:

```
rule 2

active_object: [adam_browser]
active_method: user interface event
when: before
condition: displayed objects changed
action: update the list of displayed objects
```

Hence the communication with the database interface is described by rule 2 rather than the code in the *callbacks* within the interface. Here EVE is integrated within ADAM as an object, thus the detection of events generated by the interface is simplified.

In summary, EVE's dynamic displaying feature works as follows: a list of currently displayed database objects is maintained by the system. This list is updated whenever an external user interface event such as clicking of the "next" button occurs (rule 2 fires). If any object that is currently being displayed is modified then its new content is flushed to screen (rule 1 fires).

2.2 System-Level GUIs to Active Databases

The complexity of a rule's action and the automatic execution nature of rules without user intervention, makes it necessary to provide a debugger/explanation toolkit for active DBMSs. System level visualization tools for active databases are developed for the purpose of demonstrating the event-condition-action active mechanism and assisting application analyzers in understanding and debugging the rule system. This is achieved by making explicit the context in which rules are fired, i.e., which event cause the rule(s) to fire. Described here are two rule debuggers of different design approaches.

2.2.1 DEAR

DEAR [12] is implemented for the EXACT rule system on top of an object-oriented DBMS – ADAM [14].

DEAR keeps track of both rules and events. The purpose of DEAR is to provide mechanisms to

- make explicit the context in which the active rule is fired,
- focus the search during the debugging process,
- automatically detect inconsistencies and potentially conflicting interactions among rules.

The model of traditional debuggers cannot be totally migrated into the context of active rules. Conventional debuggers for programming languages are *context-independent* while the conflict set of rules (i.e., rules eligible for firing) depends on the events raised hence it is *context-dependent*.

DEAR shows the intertwined cycle of rules and events by presenting a tree whose root is artificially created and direct descendents are the first events to be raised. Nodes can represent either events or rules, where event nodes alternate with rule nodes. An arc from an event node to a rule node means that the event awakens the rule, and an arc from a rule node to an event node means that the event was produced by the rule. The debugger also makes explicit the conflict set of simultaneously triggered rules in the displayed tree. A link between event nodes shows that these events arose simultaneously. This group of events can define a conflict set of rules since a rule is eligible for firing if its event is awakened.

To provide a more focused tracking, DEAR has a "pruning" feature to reduce the size of the tree shown by the debugger. Debugging can be restricted to certain rules and/or events by the *spy_rule* and *spy_event* facilities. The tree will show only

those nodes corresponding to any of the spy points specified by the user. DEAR also allows the designer to trace when some situation changes rather than by following certain events and/or rules. The user can indicate which database attributes should be watched, so that the tree will be generated once an update is detected on any of these attributes. This is supported by the *data_spy* window.

For detection of inconsistencies and conflict interaction, DEAR can point out potential cycles by highlighting the branch where an event is awakened twice.

2.2.2 SimBug

The approach of SimBug [5] is to pursue the debugging of active databases with *simulated event schedules*. A set of event schedules will be generated for each activity description to simulate different real world scenarios. Relevant atomic and complex events including their parameters are mapped on a time line using a simulator firing these *fictitious* events. During this process of firing fictitious events the user can compare the observed behavior with the expected one. A small explanation component is included to support understanding of the active behavior. Moreover, the user can decide at different points how the execution should go on.

SimBug provides asynchronous simulation using event-oriented clock technique. The complex event detector is bound to receive the information about the occurrence of each fictitious event by the simulator.

The execution of the debugging process can be described as follows: once a program has been compiled successfully and a set of event schedules has been created, one schedule will be chosen to run. The clock values of the system will be set to the time of the first fictitious event marked on the timeline of the event scheduler. Subsequently raised atomic-, composite events and fired rules will be presented on the graphical user interface through icons and their relationships (event triggering rule) can be shown through lines and arrows.

The debugging process will be interrupted at points where, for instance, parameter values of fictitious events are missing, but needed for the evaluation of a condition. Conflicting rules will be presented through flashing icons. During the simulation the user can select one of the conflicting rules and decide how the simulation should go on.

Net-shaped presentation of icons relate to cycles of rule execution. These cycles may have implication on the guaranteed termination of the rule system. The user shall decide if the cycles are a bug.

CHAPTER 3
MOTIVATION

3.1 Application-level GUIs

Current research has shown that both the end-users of visualizations and databases can strongly benefit from the *integration* of these systems [4]. To support this integration, there are three types of architectures that can be identified from the human user perspective:

- *close-coupled*: all components reside on one machine, sometimes all written in the same language, and even existing in the same process, i.e., the database manages its own visualization. Close-coupling is best suited for single-user, frequent-transaction scenarios.

- *client-server*: two different processes connected over a network. The database acts as a server and visualization acts as the client. This architecture allows the distribution of services and multiple clients.

- *distributed asynchronous process communication*: database and interface separated by a manager layer, where separated objects of the database are responsible for informing the display layer about changes in the database.

Also, appropriate *software components* such as libraries, toolkits and various hardware environments need to be researched.

The generalization of *application-level* database user interfaces is not well defined. A rough categorization of this large and diverse group of applications includes *schema-level* and *data-level* browsers, according to the types of objects they manipulate.

Normally there are two kinds of tasks that a browser can conduct: *display* and *query*. Both operations entail the mapping from database objects to visual objects. We refer to this mapping as **visualization** – a set of conventions for obtaining pictures from data.

However, the emergence of active database applications has produced a new kind of higher-level database user interfaces. We call it *action-oriented* GUIs, as opposed to the schema- and data- browsers, which can be called *data-oriented* GUIs. Unlike the latter, action-oriented visualization tools are interested in some user/application-defined situation or abstraction of data rather than data itself. The situation usually changes without user knowledge and these changes are the result of the active database system. Clearly this situation is tightly related to the rules of the system. When some rule fires the action may cause the situation to change. The task of the user interface is to demonstrate the situation changes and convey them to users graphically. As mentioned before, integration of visualization module into the database favors performance, reusability and extensibility. It is advantageous to couple action-oriented GUIs with the rule system, especially with the action module of the ECA mechanism.

Action-oriented graphical user interfaces is a new concept which has not been fully addressed in the literature so far.

EVE [13] adopts active capability to achieve dynamic displays. As we have seen, updates of display is done by the action part of rule 1, i.e., architecturally speaking EVE is tightly coupled with the rule system. However, EVE is a data-oriented browser. The functionality of EVE is not totally action-driven.

Our goal is to examine the idiosyncrasies of action-oriented graphical user interfaces, focusing on the architecture, design and implementation.

3.2 System-level GUIs

As discussed earlier, it is necessary to have a debugging tool to monitor events, rules, database objects and their inter-relationships at run-time. In order to test the correctness of hypothetical scenarios it is often desirable to allow user intervention (e.g., enable/disable an event or a rule, compose new event/rule).

Apparently this kind of interactive feature is lacking in both of DEAR [12] and SimBug [5] discussed in 2.2.1 and 2.2.2 respectively. In DEAR, events and rules are monitored and displayed at run-time without user interruption. The only user-initiated request that may alter the visualization is the "spy" command prior to execution. In SimBug, all events in execution are fictitiously created by a simulator, so the user cannot specify any desired sequence to trace and the visualization process is uninterruptable. Furthermore, there is no actual run-time debugging of any real application in SimBug because the trace is a simulated one.

While examining the extant rule debugging tools we also find the following limitations in addition to the above one:

- The intricacy of composite event detection is not adequately explained.
- Relationships among events are not shown.
- Rule-database interactions are not addressed.

Below, we enumerate some design goals of our approach:

1. Enhance interactivity by allowing user-input dynamic changes to the rule system at "break points".
2. Make both run-time and post-analysis trace available.
3. Demonstrate event detection process by displaying parameter contexts and global event history.

4. Clearly show event-event, event-rule, rule-database relationships.

5. Explore alternative loosely-coupled architecture using communication network facilities.

The design and implementation of application- and system-level visualization tools for the Sentinel active OODBMS is detailed in Chapter 5 and Chapter 6.

CHAPTER 4
OVERVIEW OF OPEN OODB AND SENTINEL

It is clear that the capabilities of the conventional record-oriented data models are limited in capturing complex structural relationships and behavioral properties in advanced application domains. These limitations have led to the transition to the object-oriented models which offer a variety of modeling constructs, which simplify the task of modeling complex data. An object-oriented database management system (OODBMS) has the following salient features which distinguish it from the conventional systems.

1. Every object has a unique, system wide object identifier (OID). objects can be created to permanently exist, or *persist*.

2. Flexible abstract data types are supported by *encapsulation* of data and operations in the object type definitions. Complex objects are defined in terms of hierarchies.

3. *Inheritance* of structural and behavioral properties is supported among object classes in the hierarchies.

Benefits of the object-oriented paradigm is many-folded. One of them is extensibility and reusability of existing software. Using the feature of abstract classes and the inheritance mechanism, we can easily extend an existing system. In this thesis a prototype active OODBMS called **Sentinel** (which is based on a passive OODBMS named **Open OODB**) was used as the basis for design and implementation of our work.

4.1 Passive Open OODB

The Open OODB project [27, 17], initiated by Texas Instruments, was an effort to build a high performance, multi-user object oriented database management system (OODBMS) in which the the database functionality can be tailored for the diverse needs of applications.

The system provides an incrementally improvable framework that can also serve as a common testbed for research by database, framework, environment and system developers who intend to experiment with different system architectures or components.

The Open OODB system architecture is divided into

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

ii) an extensible collection of *policy manager modules* which provide functionality to the system.

Since Open OODB is an Object-oriented front end, it uses **Exodus** storage manager as its underlying storage manager through an interface.

4.1.1 Functional Features of Open OODB

- **Seamless Interfaces:** Open OODB *seamlessly* adds functionality such as: persistence, concurrent transactions, and schema evolution to developers' existing programming environments. Open OODB does not require changes to either type (class) definitions or the way in which objects are manipulated. Rather, applications "declare" normal programming language objects to possess certain additional properties; such objects then transparently "behave properly" according to the declared extensions when manipulated in the normal fashion. For example, if an object is declared as persistent, the DBMS is responsible for

moving it between the computational and long term memory as needed to ensure both its residency during computation and its preservation during program termination. This allows programmers to:

i) stay within familiar programming paradigms,

ii) stay within familiar programming languages, and

iii) support legacy code and data.

OODB extends existing languages (C++ and Common Lisp) rather than trying to invent a new "database language".

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

controls and performs the actual extension if it is determined that an event should be extended. The sentry manager is used for specifying events to be extended, and is responsible for deploying sentries to detect extended events.

- **Extensibility:** When an object is declared to Open OODB to have "extended" behavior, there are certain "invariants" associated with the extension that must be enforced. When an operation involving an extended object occurs, the sentry is called which as detailed above interrupts the operation and transfers control to a *policy manager* module responsible for ensuring that operations against extended objects "behave properly". Each semantic extension is implemented by a different policy manager. Thus, there is a policy manager for persistence, another for index maintenance, etc. Policy managers can be added independently, and are inherited from a common root class to make them type compatible for invocation purposes. This strategy allows new extensions to be added, the semantics of a given extension to be changed, and implementation of a given policy to be changed or selected dynamically. It allows for hiding the semantic extension from applications to obtain seamlessness. Basic services used by policy managers are provided by a collections of "support modules".

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

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

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

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

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

4.1.2 Operational Features

The class lattice that manifests the functionality of the system is shown in Figure 4.1.

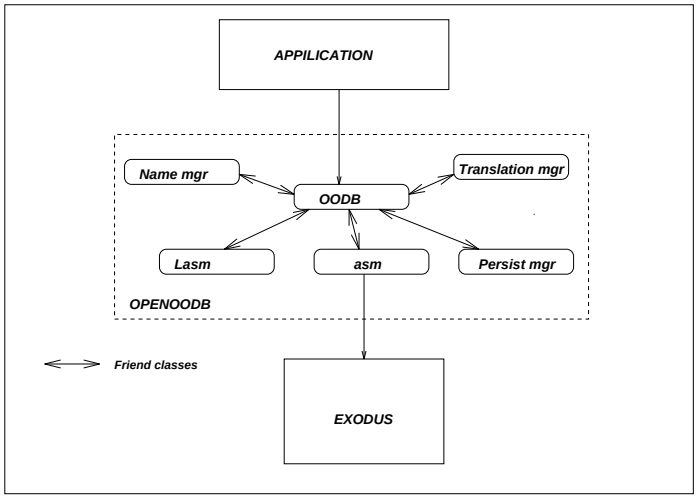

Figure 4.1. Functional Class Lattice of Open OODB

- **OODB main class.** This is the main interface class between the application and the system. In every application an instance of this class must be created at the very beginning. The constructor of this class, does the following: establishes connection with the Exodus storage manager, loads all the system tables into the main memory.

- **Preprocessor.** The Open OODB extends every application class with certain additional member functions to take care of object translation, sentry mechanism and persistence. This is achieved by the preprocessor. By default every application class is derived from the *wrapper* class. However, if the user does not wish to extend a certain class, the application class can be left as such and not derived from the wrapper class by use of the -*n* option of the preprocessor.

- **Cache manager.** Open OODB does not maintain a true Cache of its own. The tables implemented by the class *lasm* (which is the local address space manager) and the *C heap* serve as the cache. When objects are fetched from the persistent store they are placed in the C heap. Each persistent object is given a global identification (GID). The tables maintained by the local address

space manager map the local address to the GID when the persistent object is fetched.

- **Name manager.** The name manager has a *flat* namespace. This implies that we cannot have two objects with the same name in two different databases. To distinguish them, however, we can qualify them by the name of the database they are stored in.

- **Persist manager.** This is the policy manager which takes care of persistence. An object is made persistent by invoking the *persist* function. The persist manager first assigns a Global identifier (GID) and uses the services of the name manager to associate the name of the object with its GID. Secondly the persist manager determines the *transitive closure* which includes every object reachable from the original object. Only the root object (original object) is assigned the GID.

- **Translation manager.** The class implements the Object translation from an external to an internal computational format. In particular, the translation converts C++ objects (along with its transitive closure) into a particular format and vice versa. Also addresses (of objects) are swizzled by the Object Translation module when a persistent object is fetched. The fetched objects are allocated on the heap. If the persistent object contains a pointer to another persistent object, a surrogate for that object is created and pointer is swizzled to the address of the surrogate. Later, when a member function is called which accesses the state of the referenced object, that object is fetched.

- **Address space manager.** The address space manager, implemented by the class *ASM_Client*, serves as the interface to Exodus. It is responsible for establishing connections with Exodus, and also fetching and storing objects in

Exodus. This interface actually utilizes the client interface functions of Exodus. It consists of functions to create, modify and delete objects.

4.2 Active Capability of the Sentinel System

The Sentinel architecture [8] shown in figure 4.2 extends the passive Open OODB system [27, 17]. This approach has been an *integrated* one, i.e., support for primitive event detection and nested transactions is incorporated as part of Open OODB's kernel. In addition, support for composite event detection and rule management is added as separate modules.

4.2.1 Architecture

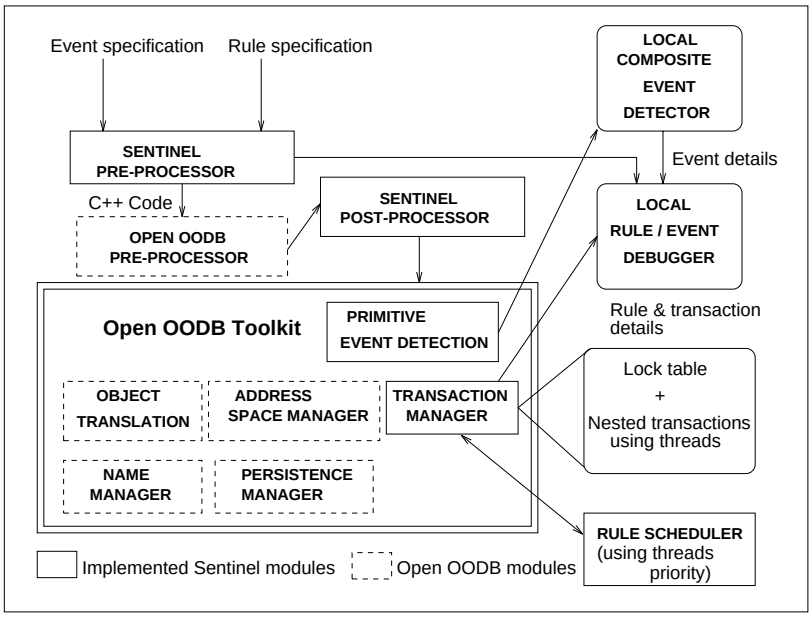

Figure 4.2. Sentinel Architecture

This section describes the overall architecture of Sentinel and its components, highlighting its extensions to the passive Open OODB system. These extensions include:

- Implementation of a full Sentinel C++ *pre-processor* (and a Sentinel *post-processor*) to transform the ECA rules specified either as part of a class definition or as part of an application; these processors are different from the C++ pre-processors used by the Open OODB in that Sentinel pre- and post-processors convert the high-level user specification of ECA rules into appropriate code for event detection, parameter computation, and rule execution, while the tasks of the Open OODB pre- and post-processors are of object level.

- Detection of *primitive events* by notifying the local composite event detector from within each wrapper method if that method is identified as an event. Sentinel post-processor adds this notification into the wrapper method, which is provided by Open OODB's Sentry mechanism.

- Implementation of a *local composite event detector* for detecting composite events (within an application) and parameter computation in various contexts [19, 7]. The event detector is implemented as a class and there is a single instance of this class per application.

- Implementation of a *transaction manager* for supporting nested transactions used for concurrent execution of rules. Light weight processes are used for both prioritized and concurrent rule execution. Nested transactions are supported in the client address space (as opposed to the server address space) and a separate lock table is maintained by Sentinel. This gives a two level transaction management (top level transaction concurrency is provided by Exodus at the server and the nested transaction concurrency by Sentinel for each client). There is no recovery at the nested subtransaction level.

- Implementation of a *rule debugger* for visualizing the interaction among rules, between events and rules, and between rules and database objects. This will be discussed further in 5.3.3.

Figure 4.3 shows how the class lattice of Open OODB has been extended. The classes outside the dotted box have been introduced for providing active capability. This figure also shows the kernel level enhancements to the Open OODB modules to accommodate nested subtransactions.

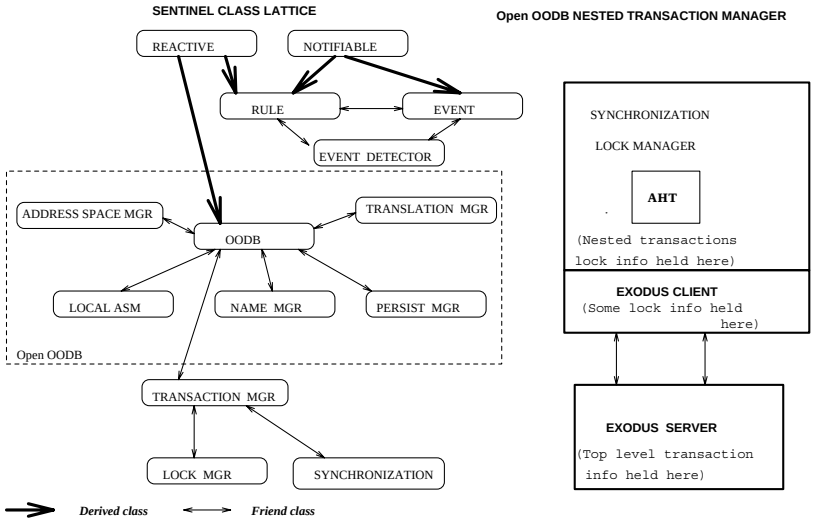

Figure 4.3. Class Hierarchy and Functionality of Sentinel

The architecture (new classes and modules) shown in Figures 4.2 and 4.3 supports the following features:

i) detection of primitive events,

ii) detection of local composite events,

iii) parameter computation of composite events,

iv) clean separation of composite event detection with application execution,

v) execution of rules in immediate and deferred coupling modes, and

vi) prioritized and concurrent rule execution.

event [**begin** (*eventName*)] [&& **end** (*eventName*)] *methodName*
event *eventName* = *eventExpression*

Event Specification Syntax

Figure 4.4. Sentinel Event Specification Syntax

rule *ruleName* ([*eventName* =] *eventExpression* | *eventName*,
conditionFunction , *actionFunction*
[[, *parameterContext*][, *couplingMode*]
[, *priority*][, *ruleTriggerMode*]

Rule Specification Systax

Figure 4.5. Sentinel Rule Specification Syntax

4.2.2 Rule Implementation

A high level event/rule format was introduced to allow users to specify events and rules at an abstract level. This event/rule format is preprocessed and changed into Sentinel system calls.

The syntax of a Sentinel event/rule specification is:

Event interface is specified to deal with methods as primitive events. This event interface specification is pre-processed by adding wrapper methods to notify the event detector when they are invoked. *Event expressions* specify primitive and composite events using event specification detailed in Snoop [11] which supports a number of event operators (e.g., *and, or, sequence, aperiodic*). The BNF of the event specification language can be found in [22].

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

Parameter_contexts are useful for different classes of applications. Four parameter contexts have been introduced in Sentinel: *recent, chronicle, continuous, cumulative.* The default context is assumed to be recent.

Coupling_mode refers to the execution points. Currently, *immediate* and *deferred* coupling modes are supported between event and condition-action pair. There are some implementation difficulties for supporting *detached* mode.

Priority classes are used for specifying rule priority. Sentinel provides a global conflict resolution mechanism among the priority classes and concurrent execution of rules that belong to the same priority class.

Rule specification is done at class definition time and as part of an application. Sentinel supports rule activation and deactivation at run-time.

(*Rule_trigger_mode*) is used for specifying the time from which event occurrences to be considered for the rule. Two options, *NOW* and *PREVIOUS* are supported, with NOW being the default.

The syntax of a rule is the same for both **class level** and **instance level** rules. A class level rule satisfies the inheritance property. Any class whose events are used in rules (either class level or instance level) need to be REACTIVE. When a user-defined reactive class is pre-processed, appropriate primitive events and rule declarations are generated and inserted in the application program. Since this rule will subscribe to an event expression that is specified on a class level, this rule will be notified whenever any object of this class invokes the method that are potential event generators.

4.2.3 Functionality of the Sentinel Modules

The control flow for supporting Sentinel's active features is further elaborated in Figure 4.6.

Detection of primitive events is based on the design proposed in [3]. Primitive events are signaled by adding a notify procedure call in the wrapper method by the

Sentinel post-processor. Both primitive and local composite events are signaled as soon as they are detected. However, the detection of a composite event may span a time interval as it involves the detection and grouping of its constituent events in accordance with the parameter context specified. A clean separation of the detection of primitive events (as an integral part of the database) from that of composite events allows one to

i) implement a composite event detector as a separate module (as shown in Figure 4.2) and

ii) introduce additional event operators without having to modify the detection of primitive events.

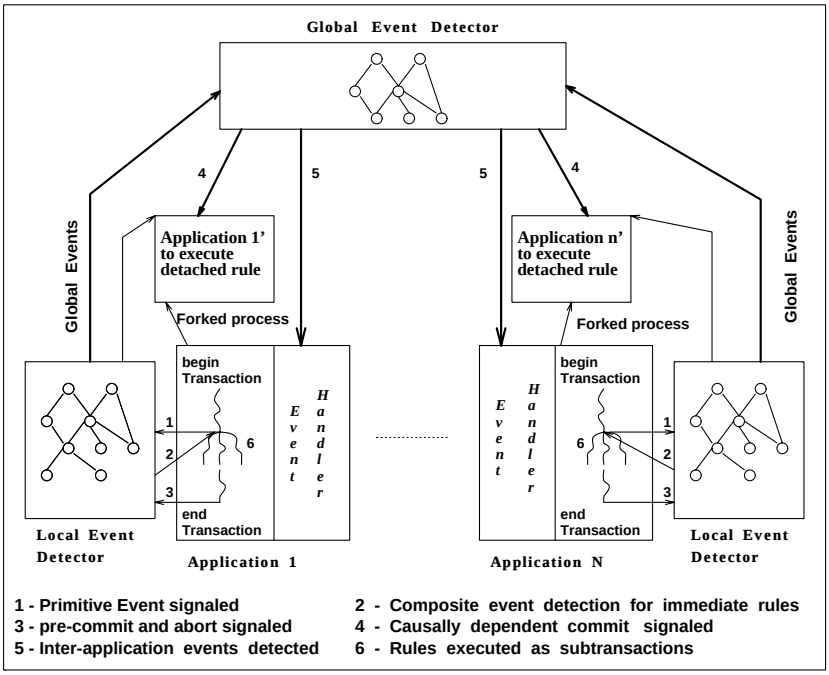

Figure 4.6. Local and Global Event Detector Architecture

Each application has a local composite event detector (Figure 4.6) to which all primitive events are signaled. The implementation uses *threads* (light weight processes), instead of processes, for separating composite event detection (as well as for the execution of rules) from application.

When a primitive event occurs it is sent to the local composite event detector and the application waits for the signaling of a composite event that is detected in the *immediate* mode. The local composite event detector and the application share the same address space and the event detector uses an event graph similar to operator trees [7].

For rule execution, a nested transaction manager (Figure 4.3) is implemented with its own lock manager. This is in addition to the concurrency control and recovery provided by the Exodus storage manager for top-level transactions. Each rule (i.e., condition and action portions of a rule) is packaged into a sub-transaction. A number of sub-transactions are spawned as a part of the application process. The order of rule execution is controlled by assigning appropriate priorities to each thread based on the priority of the rule and the priority of the triggering rule (if there is one). Support for multiple rule execution and nested rule execution entails that the event detector be able to receive events detected within a rule's execution in the same manner it receives events detected in a top level transaction. This is accomplished relatively easily by separating the local composite event detection from the application as shown in Figure 4.6. This separation also readily supports both online and batch (or after-the-fact) detection of composite events.

CHAPTER 5
DESIGNING GRAPHICAL USER INTERFACES FOR SENTINEL

5.1 User Requirements on Data Visualization

As indicated in Chapter 1, database users from different communities have drastically varied requirements on the appearance and functionality of visualization tools. Particularly, non-database experts usually expect the visualization tool to be easy to perform and understand, i.e., it should allow complex interaction between the user and the database to be conducted in natural and intuitive manners. Furthermore, results of database execution should be presented effectively and unambiguously.

With the advent of high speed graphical display devices, windowing operating systems, multimedia and other GUI advances in various computer disciplines, the users of database management system are expecting and demanding more from their user interfaces.

User interfaces for advanced database management systems differ from ordinary visualization tools in that they should provide the users a way of interaction with the underlying DBMS in addition to being capable of displaying the entire spectrum of different types of data effectively.

The design of a user interface should observe the following rules:

i) *Usability* is an important aspect of GUI design since the principal function of a graphical user interface is to facilitate the human-computer dialogue.

ii) *Simplicity* and *User-friendliness* are of ultimate importance for users to reduce the learning curve.

iii) *Portability* has become an imperative in present day's software arena.

During the past years windowing systems have gained the position of the underlying programming environment of most graphical user interfaces. This is due to their interactive nature from the human-machine relation perspective. In a data visualization objects that appear before the user can be thought of as "widgets" while various behaviors are assigned to screen visuals. These behaviors might be elicited when pointed at with a picking device (typically, a mouse) via a *callback routine*. Such actions can trigger additional database interactions and visualizations, thus affording a mechanism with which to do database browsing.

The emergence of active databases has added a new dimension to the interaction between applications and DBMSs. Traditionally all these interactions have been driven by the application. Visualization tools following this driving force are application-oriented. In such applications the user normally submits requests to the DBMS and receives responses from the system in *synchronous* or conversational fashion. In contrast, in an active database application, it is difficult for the user to predict what event(s) might happen and what actions the system might take as a consequence; he/she receives responses from the DBMS in an *asynchronous* and passive way. In some circumstances the user may not receive any reply if no rule fires. This kind of human-computer interaction distinguishes itself from traditional ones in passive database environments.

In the context of active databases, the functionality of data visualization will switch its emphasis from traditional data navigation/browsing to reflecting the changes resulting from event triggering/rule firing.

In addition to the new requirement imposed by active databases and general rules for GUI design, different user perspective may also create a diversity of design choices.

Non-database expert users are more interested in application related data than system related data. In other words, they only expect to carry out requests in the

application and receive results out of the system, without understanding the details of the execution process. The user interface is meant to highlight information related to the application without it being clustered with irrelavant data reflecting the states of the underlying system.

Application developers, on the other hand, need to understand the details of the system's functioning modules, trace the execution, discover existing or potential errors and correct the errors if necessary. Their interests can go well beyond a specific application's running behavior. More information of the system kernel should be presented expressively via the user interface to ensure that the user make a good understanding of it.

The following sections elaborate the design of an application level GUI and a system level rule debugger for the Sentinel active DBMS, illustrating the observations discussed above.

5.2 MDP - An Application Level GUI

MDP, Plan Monitoring Application, is a military application utilizing the active capability of the Sentinel DBMS.

5.2.1 Description of the MDP Application

In this application there are two kinds of critical situations that may create alerts when they occur:

- *non-weather related*

  The navy consists of various units (termed as *unit objects*), positioned at various locations for performing some task. Each unit has a *readiness* status indicating whether they are in a position to perform certain operations. Readiness can be defined based on personnel, training, supplies etc. and is maintained in terms of ratings (e.g., 1 signifying Combat Ready and 5 signifying Overhaul). A

readiness rating of 2 or below is desired for any unit. As a crisis arises a *plan* has to be prepared to deal with it. A set of units have to be assigned to carry out each plan. Once this is done any change to either the plan or a unit's readiness status has to be monitored continuously.

- *weather related*

  There are a fixed number of geographic regions in which the weather will be monitored. The weather is described by wind speed, wave height and the date on which this weather is valid. When severe weather condition is reported for a certain region, for example:

  a) `wind_speed > MAX_WIND_SPEED` or
  b) `wave_height > MAX_WAVE_HEIGHT`

  where `MAX_WIND_SPEED` and `MAX_WAVE_HEIGHT` are predefined limits, the following action will be taken: identify units in the affected region and warn commanders of the severe weather. Two kinds of weather-related tasks can be done: weather monitoring and weather forecasting.

It is clear that the active capability of Sentinel can be used to achieve automatic monitoring of the above situations.

5.2.2 Design of the MDP Graphical User Interface

Currently, MDP is a single-user application, which means every user runs an instance of MDP locally. In this environment the user interface is appropriate to be designed as *tightly coupled* or *integrated* with the system. The Sentinel DBMS and the underlying Open OODB are easily accessible by MDP as libraries. The benefit of this design approach is that the system and the visualization program run in the same address space hence system data can be readily shared by parameter passing and procedure invocation. Thus the user interacts with the Sentinel system in a simple and asynchronous manner: he/she submits one or many *reports* at a time

and wait for the system to process the messages, detect events and carry out actions according to rules that would fire. Results of actions are sent back to the front end for the user to visualize.

Since there are two types of critical situations to monitor (weather related and non-weather related) they must be treated differently to suit user's need.

In our design, non-weather related result message is shown as plain text. For weather related messages in addition to text output we have arranged a 2-D raster image representing the geographic area we are currently monitoring, which is a partial world map containing most of the Pacific Ocean and its Asian and American coast lines. A region is defined as a rectangular area in the map. Currently there are 10 regions of interest. When severe weather change emerges in a region and some appropriate alert should be issued the region will be highlighted around its border in an eye-catching color. Newly created color rectangle will blink to enhance user attention until a newer update happens.

Corresponding to different severity a region can be highlighted in one of 3 colors: violet, orange, red, representing the weather condition in the ascending order of gravity.

Figure 5.1 shows the layout of the MDP graphical user interface. It contains a group of subwindows and a set of action buttons.

1) *Regions Monitored*: this is a scrolled list containing the names of all regions being monitored.

2) *Region Information*: this is a scrolled text pane showing the position (longitude/latitude) of a region.

3) *Incoming Reports*: this is a scrolled list of possible reports that user may submit.

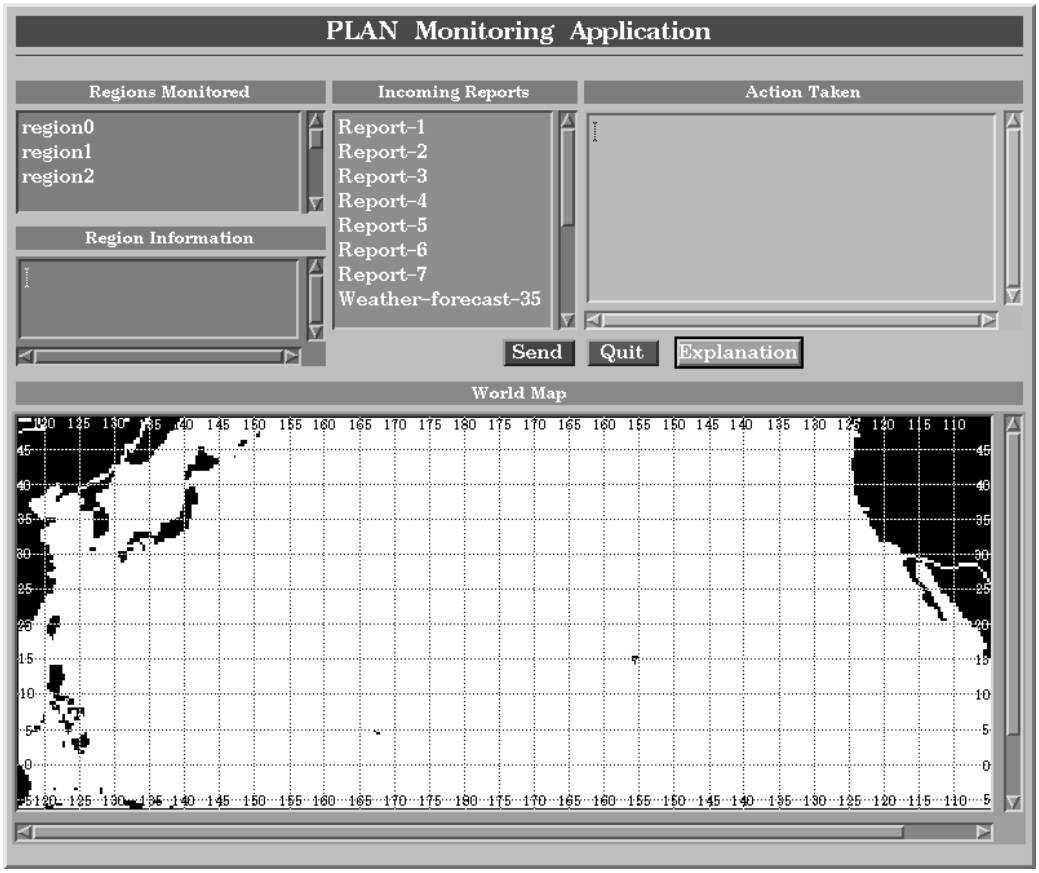

Figure 5.1. MDP Layout

4) *Action Taken*: this is a scrolled text pane that gives the result of execution or action taken by the MDP system according to the report(s) submitted.

5) *World Map*: this is a canvas containing a portion of a gridlined world map. All regions are inside this map. At different execution stages a region's border rectangle may be highlighted and its color may change to one of the following colors:

blue, when the region is picked statically in the Regions Monitored window to retrieve its position information.

violet, orange and red, represent different severity of weather in that region, in ascending order.

6) *View Button*: for browsing the content of reports.

7) *Send Button*: for submitting report(s) after selecting one or more reports in the Incoming Reports window.

8) *Quit Button*: for exiting from the MDP application.

9) *Explanation Button*: for "help" purpose. When this button is clicked there pops up a dialog window with a scrolled text pane containing explanation information.

In the future we plan to distribute the execution of MDP over many distant sites. Each site runs a copy of the Sentinel system and MDP. These sites can send messages (including plain text, graphics and possibly audio data) to a remote message server via a network communication interface. Once the interface is developed MDP can be seamlessly upgraded to run distributedly without drastic architectural reformation.

5.3 Sentinel Rule Debugger - A System-Level GUI

5.3.1 Background, Observations and Objective

Using ECA rules in active database systems for real-life applications involves implementing, debugging, and maintaining large numbers of rules. Experience in developing large production rule systems has amply demonstrated the need for understanding the behavior of rules especially when their execution is non-deterministic.

Availability of rules in active database systems and their semantics creates additional complexity for both modeling and verifying the correctness of such systems.

There has been an obvious lack of suitable mechanisms to examine whether the active behavior designed for an application corresponds to the intended one. Even though the active capability in terms of rules can be used beneficially in a wide range of applications, it is not a trivial task to understand, debug and maintain a large number of rules. The *static analysis* of database production rules has been discussed in [28, 1].

The **static analysis** methods are used for determining whether arbitrary sets of database production rules are guaranteed to terminate (*Termination*), produce a unique final state when multiple rules are triggered at the same time (*Confluence*), produce a unique stream of observable actions (*Observable determinism*). Termination is analyzed by constructing a directed triggered graph for a set of rules R. If there are no cycles then rules are guaranteed to terminate. Rules in R are said to be confluent if every execution graph of R has at most one final state. Also partial confluence is analyzed by analyzing confluence for a subset of rules in R. The algorithms presented in [28, 1] are conservative: they may not always detect when a rule set satisfies these properties. However, they isolate the responsible rules when a property is not satisfied.

Run-time analysis is in terms of interaction among rules, between rules and events and between rules and database. Runtime analysis augments the static analysis and gives the user a better understanding of the system. In addition to static analysis it is also necessary to provide a runtime analysis of rules for the following reasons.

- We need to have an expressive rule language to accommodate a diversity of applications. However the expressiveness adds complexity. One solution is to

find an appropriate *declarative* rule language which would allow detection of rule inconsistencies at compile time. This is static analysis. Although these languages are appropriate for specifying conditions over database states, some problems arise when trying to describe reactive capability declaratively. For example, reactions could be arbitrary operations other than database operations. It may be difficult to formalize these operations in order to do static analysis. Hence the expressiveness of the rule language can be limited to focus on a particular application (e.g. integrity rules which have restricted reactions only to update/restore the state of a database once a violation occurs). As the number of rules increases it becomes difficult for the administrator to foresee possible interactions between rules. A rule debugger would be of immense help in this regard.

- The nature of rule execution is dynamic. The rules are fired in response to a certain event. Due to the event based nature of rule execution the user does not know in advance when a rule would be fired. Hence there ought to be some sort of explanation mechanism which allows the users to ascertain what rules are activated and when. There needs to be some visualization of the runtime trace of rule execution.

- Rules are eligible for firing if appropriate events are raised. We could have multiple rules being triggered by an event and these rules can be executed in any order. These rules form a *conflict set*. The user can focus on this set and change priorities if required. Presentation of event-rule context is a run-time issue.

- There can be nested execution of rules. Sometimes the nested execution may lead to potential cycles i.e. a set of rules repeatedly triggering each other (which

may have been determined by static analysis). It will be helpful to recognize and highlight these potential cycles at runtime.

- It is also very helpful to show how the execution of rules effects the state of the database. This could be shown in terms of the objects modified by the rules. This makes sense in an object-oriented context since we have object identifiers to keep track of the objects.

We conclude that both static and run-time analysis are important in an active database management system. In this thesis we concentrate on the runtime analysis of rules.

It is clear that we cannot adopt the model of traditional debuggers (of imperative languages) readily in the context of active rules. The reason is twofold:

- The conventional debuggers show the sequence in which the tracked units (instructions in a programming language) executed. The situation is *context independent* since the user is already aware of the context. For example, the instructions of a program are executed in a fixed sequence given by the programmer. If there is an error such as an overflow, non-existent pointer dereferencing, etc., then the program fails and with the help of the debugger the user can locate where the error has occurred.

- In a conventional program debugging environment, the factors considered are variables, subroutine calls, exceptions (stack overflows), pointer referencing / dereferencing, etc. The debugger aids the user in this process by furnishing low-level details such as the line number of the program where the error occurred, variables accessed, etc.

By contrast, the rules in an active database system are executed without any user intervention. There is no means by which the user can know in advance which rules

will be fired. Hence the debugger geared for active rules has to be *context dependent*. The context for rules is determined by the events which caused them to fire [12].

When we consider debugging in the context of an Active database system, we need to take into account the following factors.

- **Database component.** Different from conventional program execution, the database operations are performed on database objects and carried out in well defined atomic units namely transactions. In order to ensure atomicity and the correctness of the operations, locking schemes are used according to the transaction model adopted. These locking mechanisms enforce concurrency control mechanisms when several transactions compete for access to database objects. To get a complete picture of the states of a database system we need to trace the *transactions*, *database objects* and *locks* held (especially when there are concurrent operations).

- **Event/event relationships.** The active feature of a database system adds another dimension to the process of debugging. We have to consider the interactions *among events*, *among rules*, *between rules and events* and *between rules and database objects*. Events are divided into two categories: primitive and composite. A composite event is defined recursively as an event expression formed by using a set of primitive event expressions, event operators, and composite event expressions constructed up to that point. In an active database application, the dependency graph of all primitive and composite events form a structure similar to decision tree. To ease the understanding of the interrelations between the events there is a need to visualize this tree structure.

- **Event detection.** The core functional module of the Sentinel active database system is the local composite event detector. Since it is difficult for the user

to know which event(s) would be triggered and which rule(s) would fire in advance, it would be desirable to keep track of the event triggering and rule firing sequence to help the user understand the result. Hence extracting information such as *parameter contexts* and *time stamps* from the event detector, and displaying the *global event history graph* will significantly enhance the user's understanding.

- **Rule/event interactions.** When event(s) are raised appropriate rule(s) fire. An event may trigger several rules simultaneously. It is desirable to visualize the *context* of rule firing, i.e., which event causes which rule(s) to fire. This interaction can happen in a reciprocal fashion: a rule may raise an event which may cause some other rule to fire and so on. This may result in a nested execution of rules.

- **Rule/database interactions.** In the process of concurrent rule execution the extended Open OODB lock manager will handle concurrency control among nested transactions. Rule/database interactions are conveyed in terms of locks acquired/released on database objects.

- **Rule execution.** The nested transaction model is used in Sentinel rule execution. Condition checking and action running are packaged in a light weight process (thread) as a subtransaction. This implementation supports sibling concurrency.

- **User intervention.** In an active database application the user is not meant to direct the execution intentionally. However at times the system developers and analyzers wish to trace the system behavior more closely or need to compare different results with different initial conditions for debugging purpose. For this the user may desire to change the asynchronous nature of execution by running

the visualization program in a step-by-step mode, or changing the states of the rule system. Another case of possible user intervention appears when the number of events and rules grow and the user's interest becomes only a subset of the entire event/rule set. In this case, it is appropriate to reduce the number of objects to trace to suit the user's need. The above situations show it is helpful to allow the user to be able to interrupt the execution, input changes to existing event/rule attributes and choose a subset to monitor at certain "break points".

These requirements are initiated by application developers and system analyzers in contrast to non-database experts or end-users. The two communities of users differ in their interest.

From the above observations we depict that objective of a rule debugger for an active object-oriented database management system is itemized as follows.

- The rule debugger should concentrate on the high level details such as relationships between primitive events and composite events, rule-event interactions, interaction between rules and interaction of rules with database objects rather than the low-level details as in a programming environment.

- The debugger should show the execution of rules graphically preserving the triggering order, current status, and other relevant details. This is particularly useful when there is a nested execution of rules.

- In addition to the rule trace, the context (i.e., the events raised, whether the event was raised from within a rule or the top level transaction) should be shown.

- The debugger should allow the user to input changes and monitor only certain rules/events of his interest. This is useful in applications having a large number of rules.

- Finally, it should be possible to visualize application execution either at run-time or after the execution of an application for the analysis of rule execution. This should be accomplished without having to change either the architecture of Sentinel or the visualization tool.

5.3.2 Design Choices

Besides the issues concerning general design of graphical user interfaces discussed in Chapter 3, the following design alternatives also have an impact on the overall architecture and functionality.

- **Problem-oriented vs. program-oriented:** One approach to debugging a program is to provide a high-level description of the expected behavior of the program to the debugger. The expected behavior is in terms of what the program is intended to do rather than the low level details of the program such as program variables, subroutines, etc. For example, in an approach adopted in debugging of parallel programs [15], the expected behavior is formulated in such a way that it aids the debugger in bringing out or detecting certain problems in the application domains e.g. deadlock, starvation etc in distributed programming. This would help the programmer to compare the actual behavior and the expected behavior at execution time. The expected behavior is specified abstractly in terms of control flow, data flow and synchronization events. This is in contrast to the conventional debugging approach (of conventional programs) as exemplified by *dbx* [21] where the program behavior is modeled at low-level, in terms of the source code entities such as subroutine names, line numbers,

etc. As categorized in Hseuh and Kaiser [15] the former approach is termed as *problem-oriented* and the latter low-level approach is termed as *program-oriented*. When we try to debug/visualize rule execution and event detection in an active database management system, the problem-oriented approach is desirable. This is due to the fact that rule execution is dynamic as rules are fired in response to a situation without any user intervention. Due to the event based nature of rules the user does not have any priori knowledge of rules that will be fired in a given execution of an application. Hence the "context" of program/application is not available to the user as in the case of a conventional debugging model. The "context" has to be provided by the system. The expected behavior of rule execution is specified in the event specification and rule specification language [19, 9], which details that rule(s) are fired in response to event(s). Even though the user may be able to input changes to the system at "break points" the rule execution context is still provided by the system, only with altered result.

- **Integrated vs. loosely-coupled:** When the rule debugger is an integral part of the underlying database management system, it is termed as *integrated*. The integrated approach favors rule debugging per application as opposed to global rule debugging (across applications). The integrated approach makes it easy to show the modification of data (attributes) in certain object-oriented environments since we are operating within the same address space. In contrast in a loosely-coupled approach the rule debugger would be envisaged as a separate system which would be linked to the underlying database system by means of some communication mechanism (sockets, pipes, RPC, etc.) or by using the file system as a message passing channel. This approach favors global monitoring of rules, that is, we could have event detection and execution of rules across

applications. An application may be interested and respond to events generated in some other application. For example, a stock broker application may respond to the event of decrease in IBM stock in a stock exchange application. The main advantage of this approach is that since it can be built as a separate module independent of the active database system, it will not be constrained by the features of the latter. Also it is likely for different applications to run on different sites in a distributed manner. The main disadvantage with this approach particularly in an object oriented context is that we are operating in different address spaces. This makes tracking of data modification difficult since it would involve issues of accessing remote objects.

- **Online vs. post-execution analysis:** We can visualize events and rules when they occur at run-time (*online*) or we can do so later via a stored event/rule log(*post-analysis*). The advantage of the first approach is it has the real sense of debugging – debugging time and run-time are identical. As a result of online debugging the user is able to interact with the system during execution and possibly change the result of execution. By contrast when post-analysis is adopted there exists a lag of debugging time with respect to run-time. It is probably desirable to have both options in a debugging environment. To support online debugging we need to have some mechanism by which there is instantaneous notification of event occurrences and firing of rules. In an integrated approach this could be easily achieved by accessing variables and invoking procedures; In an loosely-coupled approach this can be done by means of socket mechanism.

- **Using preprocessing vs. not using preprocessing:** The preprocessor parses static event/rule specifications in an application at compile time. This information can be gathered for analysis and display. In this approach the rule debugger may show all predefined events, rules and their relationships

before run-time. However, to support it the preprocessor needs modifications. Alternatively, the rule debugger can obtain event/rule information exclusively at run-time as event and rule objects are instantiated. This approach does not utilize the compile-time information and can not support static analysis.

- **Batch vs. step:** The user can choose to debug the rule execution in a step-by-step fashion, i.e., the system halts every instant when there is an event raised, a rule firing, a transaction commit, etc. Alternatively, the application can run consecutively until it finishes (batch mode). Both options may be desirable because the step mode enables the user to watch the system change in a finer scale and interact with the system in the middle of execution, while the batch mode suits the situation when the user wish to visualize and understand the result of execution as a whole.

- **Interactive vs. non-interactive:** The debugger may support "break points" and allow user intervention during run-time, or it only displays results. Clearly the interactive approach is more desirable for a debugging tool, although there may exist a number of difficulties in implementing this feature.

5.3.3 Existing Sentinel Rule Visualization Tool

Design, Architecture and Implementation

The departure point for our work is from the first version of Sentinel Rule Debugger designed and implemented by Ziauddin [26]. The Rule debugger was implemented to monitor rules and events *within an application*, i.e., there is no global visualization of event detection and rule execution. A *problem-oriented*, *loosely-coupled*, *post-analysis* and *non-interactive* approach was adopted. The way of communication between the debugger and the active database system was achieved via log files which

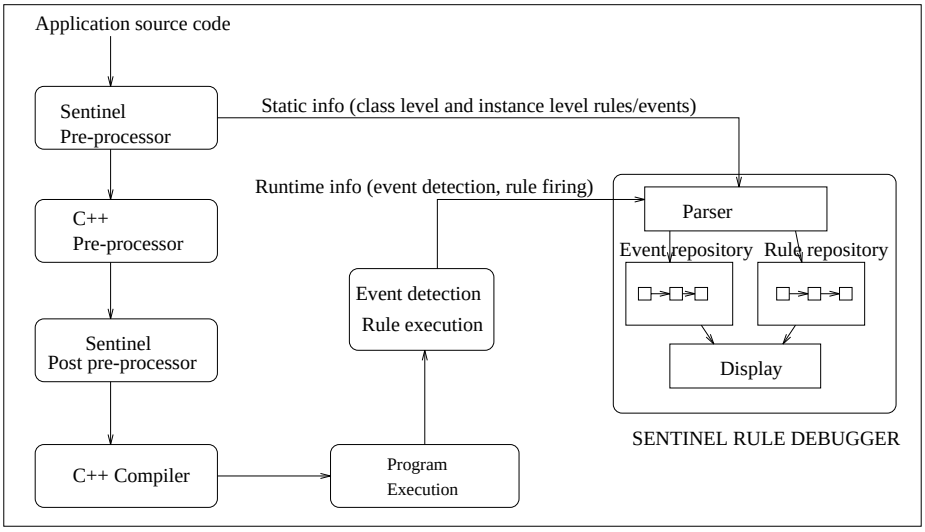

Figure 5.2. Functional Modules of the Non-interactive Sentinel Rule Debugger

contain information of rule/event definitions, rule/event object id's, the actual occurrence of events and firings of rules, and other transaction related information. From the rule/event definition the information concerning which rule subscribes to which event and hence rule-event interactions can be obtained. The transaction related information manifests rule-database interactions. The information in the log files is furnished by the pre-processor at compile-time, and the event detector and rule manager at run-time. After execution of the application finished the Rule debugger reads these files to obtain the trace of event occurrence and rule execution. Since the debugging is done as a post-analysic *there is no actual run-time visualization.* Run-time visualization is simulated by showing the occurrence of events and rule firings as and when they happened.

The architecture for the visualization tool is dictated by the architecture for event detection in Sentinel. The functional architecture of the visualization tool is shown in Figure 5.2.

The input to the rule debugger consists of

- *class and instance level rule/event definitions*: This information is supplied by the user in the application program using the event [9] and rule [2, 19] definition language. The preprocessor gathers this static information in the form of a file.

- *event detection information*: This is the run-time information obtained on the occurrences of events and the creation of event objects. This information is provided by the local event detector and written to the run-time log file.

- *rule firing information*: This is furnished to the visualization tool in the same way as the event detection information. In addition, the transaction in which rules were fired and the information concerning locks acquired/released on database objects are also furnished.

The visualization tool's parser reads the static information and stores the event and rule information in the in-memory event and rule repositories, which are linked list structures.The data structure which captures the nested execution of rules is an n-ary tree. The root node represents the top-level transaction of the application. When this transaction triggers a rule and since rules are executed as subtransactions, the child node of the top-level transaction represents the first rule fired. This node in turn could trigger another rule and it is represented as the child node (subtransaction of subtransaction) and so on. The transaction tree grows in a top-down way: it starts from the top-level transaction and spans to the descendents.

Layout and Functionality

The user interface window consists of the following parts:

- **Trace:** This is one of the buttons in the Rule debugger. When this button is activated by the mouse click the run-time trace of the transactions is shown. The trace is shown on step-by-step basis or in a continuous mode, depending

on the user's choice. In the step mode after each node is drawn, the display routine checks for a *buttonpress event* (mouse event) and afterwards the next node is drawn. Each node represents a subtransaction which maps to a rule. The color of the nodes changes at each step of the execution. A green node represents an *active* transaction, yellow for *suspended*, and red for *committed*. A sample trace is illustrated in Figure 5.3.

- Besides the *Trace* button a group of push buttons provide the following functions.

  CONTMODE: select the continuous tracing mode.

  STEPMODE: select the step tracing mode.

  CLEAR: clear the drawing-area windows.

  QUIT: exit from the program.

- **Rules** This is a list with all the rules defined in a particular application. When an item is chosen a dialog window pops up showing the description of the rule.

- **Events** This has a similar functionality as that of Rules. It lists the events to be monitored in the application.

- **rule visualization window** This is a drawing-area window displaying the rule execution (nested transaction) tree during a trace.

- **Execution console window** This is a drawing-area window displaying execution information in plain text. The information includes all transaction / subtransactions with their IDs, events that were raised and rules that were fired. This window records the event detection, subtransaction and rule firing sequence.

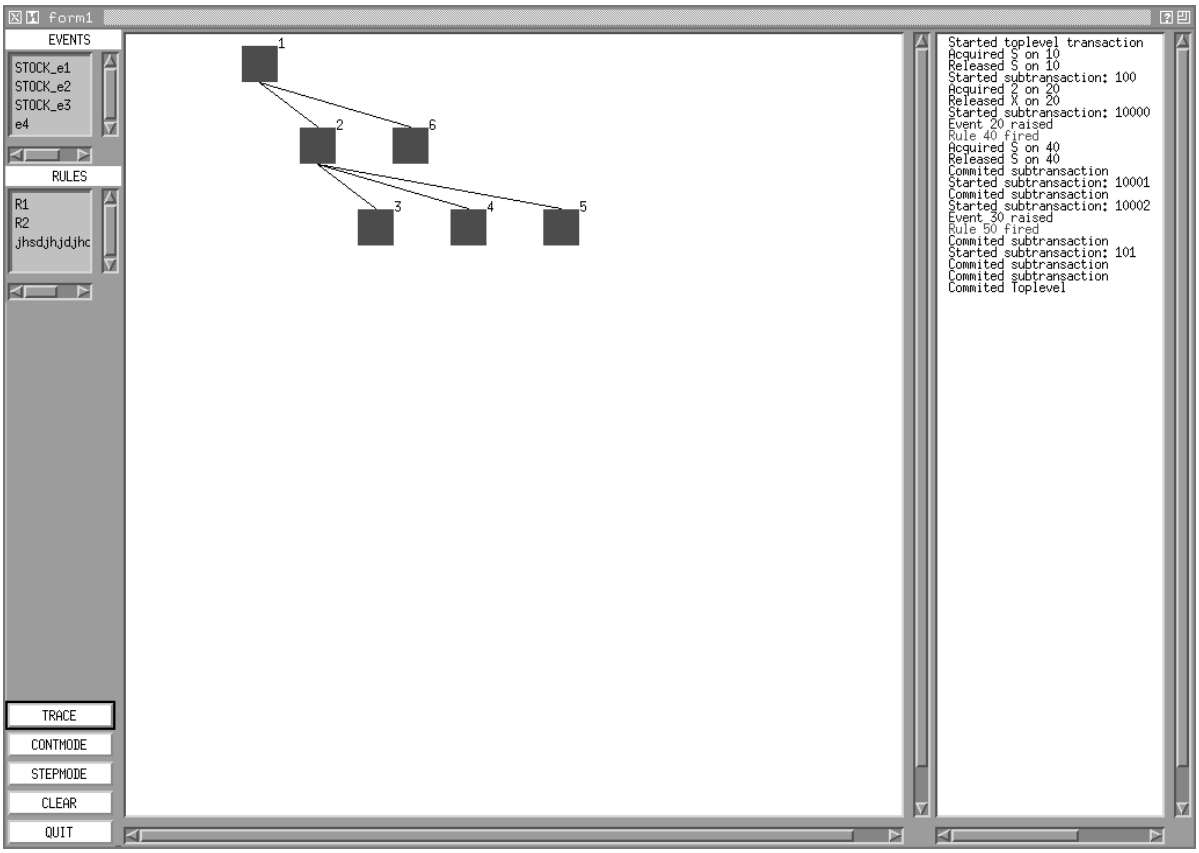

Figure 5.3. Layout of the Non-interactive Sentinel Rule Debugger

Limitations of the Existing Rule Debugger

The following lists some restrictions of the rule debugger described above. Both of design and implementation issues are covered.

- The visualization of rule execution in the existing Sentinel Rule Debugger is done in post-execution analysis using log files. There is no actual run-time visualization. On account of this restricted debugging nature, the rule debugger allows no user interaction with the system at run-time.

- All event detection information is supplied in the console window in text form. It is difficult for the user to envision the complex event-tree structure and the context of composite event detection.

- Rule firing context is shown only in text. There needs to be a more expressive demonstration of the event-rule relationship.

- The user is not able to choose a subset of events / rules to trace. As the number of traced objects increases the information presented to the user will tend to cluster.

- No dynamic analysis is done by the rule debugger. Specifically, there is no indication of the existence of potential cycles in rule execution.

- Use of different colors to represent different execution stages has enhanced the quality of illustration, but it also degrades portability — the visualization cannot be run on monochrome displays.

5.3.4 Interactive Rule Debugger - A Revised Design

To overcome the deficiencies in the existing rule debugger, we have improved the design without the sacrifice of losing any existing functionalities. The resulting debugging environment is termed *SIEVE* – **S**entinel **I**nteractive **E**CA Rule **V**isualization **E**nvironment. The following is an outline of proposed improvements.

- **Online choice** allows the user to run the debugger in the interactive mode in addition to the post-analysis mode. When the debugger runs in online mode the debugging is achieved at run-time. If the users set up break points in an application, he/she is able to input changes to the system in the course of execution and visualize the effect of these changes.

- **Event tree** demonstrates the structure of primitive and composite events defined in an application, helps the user better understand the event-event relationships.

- **Rule firing context** is shown by connecting the event-rule pairs across the event tree and rule execution tree (subtransaction structure).

- **Event detection context** is retrieved from the event detector at run-time and displayed in the form of *global event history graph* .

- **Pruning of the event and rule trees** is enabled to let the user choose a subset of events or rules of interest to trace. Thus one can avoid the interference of unwanted information.

- **Cycle checking** is done continuously during execution to alert the user of potential non-terminant rules.

The interactive debugging mode presents new requirements to the rule debugger's architecture. To preserve the design of existing modules and support code reuse, we

have implemented a communication interface using BSD sockets. Each module of the Sentinel system that needs to interact with the rule debugger will pass messages through the communication interface without changing its inner structure and functionality. These modules include the *Transaction Manager* and *Lock Manager* within the extended Open OODB *Kernel* and the *Local Composite Event Detector* and *Rule Manager* of the Sentinel active DBMS.

The implementation and functionality of SIEVE is further elaborated in Chapter 6.

CHAPTER 6
IMPLEMENTATION

In this chapter, we discuss various issues related to the implementation of both application-level and system-level GUIs for Sentinel, with emphasis on the implementation aspects of the interactive rule debugger - SIEVE. We begin in Section 6.1 with an overview of the X Window System and the OSF/Motif toolkit. Then in Section 6.2 and 6.3, the implementation of the MDP user interface and the Sentinel Rule Debugger are discussed in detail respectively.

6.1 An Overview of X/Motif

6.1.1 The X Window System

The X Window System is an industry-standard software system that allows designers to develop portable graphical user interfaces. One of the most important features of X is its unique device-independent architecture. X allows programs to display windows containing text and graphics on any hardware that supports the X protocol without modifying, re-compiling or re-linking the application.

One salient difference between X and other windowing systems is that X does not define a particular user interface style. In X only a flexible set of primitive window operations is provided without dictating the "look and fee"l of any particular application's user interface. Applications depend on higher level libraries built on top of the basic X protocol to provide components like menus, push buttons and dialog boxes. Open Software Foundation's (OSF) Motif widget set is used for this purpose in our applications.

The architecture of the X Window System is based on a client-server model. A single process known as the *server*, is responsible for all the input and output devices such as the display monitor, the keyboard and mouse. The X server typically runs on a workstation (or personal computer) with a graphics display. An application that uses the facilities provided by the X server is known as a *client*. The client communicates with the X server via a network connection using an asynchronous byte-stream protocol. The X architecture hides most of the details of the device-dependent implementation of the server and the hardware it controls from the clients. When a client application needs to use a service provided by the X server, it issues a request to the server. Typical client requests include window creation and destruction, reconfiguration and text or graphics displaying.

6.1.2 Libraries for Developing X Applications

X applications normally use libraries that provide an interface to the base window system. The most widely used low-level interface to X is the standard C language library known as *Xlib*. Xlib defines an extensive set of functions that provide complete access and control over the display, windows and input devices. Besides Xlib there are some higher-level toolkits (based on Xlib) available including the *X Toolkit*, which consists of two parts: a layer known as the *Xt Intrinsics* and a set of user interface components known as *Widgets*. The best-known OSF/Motif widget set implements user interface components including menus, scroll bars and buttons, etc., while the Xt Intrinsics provides a framework that allows the designer to combine these components and produce a complete user interface.

Each of the Motif widgets provides certain appearance, functionality and structure. The widgets are divided into classes based on general functionalities.

Another toolkit, the *Xpm* widget set is based on Xlib as well. It is used mainly for displaying X pixmaps. Xpm complements Motif's functionality in handling color

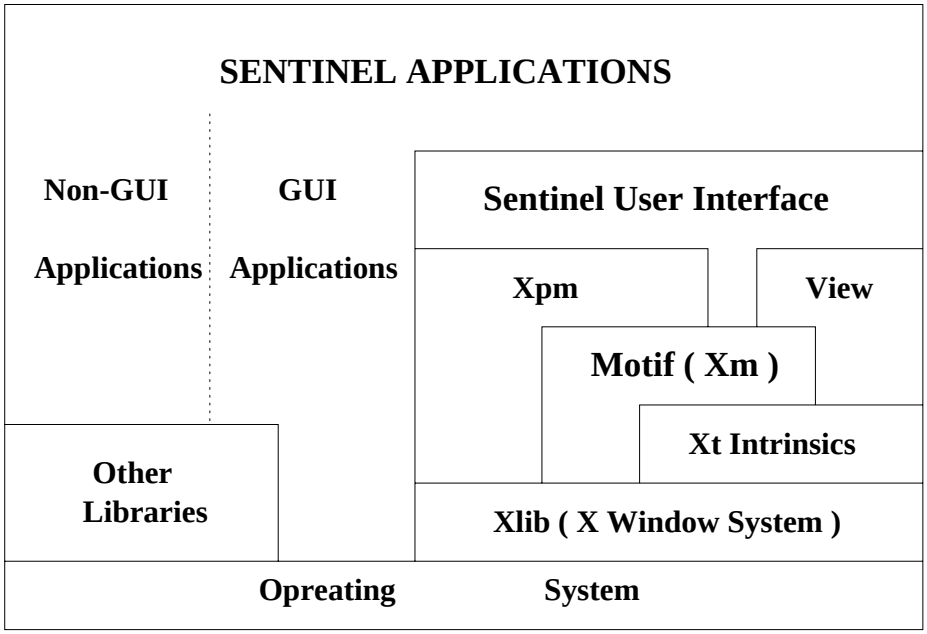

Figure 6.1. Underlying X Libraries of Sentinel Graphical User Interfaces

graphics. Motif can only load and store X11 bitmaps which are single depth black-and-white pixmaps. In the implementation of the interactive rule debugger for Sentinel the Xpm toolkit is used to manage push buttons with color images.

Developed by BBN, the *View* library is a toolkit based on Motif and Xt Intrinsics. View is suitable for user interface implementation in and object-oriented programming environment. Using the View library one can fit the design with the o-o paradigm therefore avoid the problem caused by the fact that all of the Xlib, Xt and Motif functions are written in C. Another benifit of using View is that upon the instance creation by invoking the View class constructor the necessary steps of top-level shell widget creation and application context retrieveing involved in traditional Motif programming have been included, thus reduces coding complexity. In both of our GUI applications the View library is used to exploit the above advantages.

Figure 6.1 shows the architecture of Sentinel user interfaces with their underlying libraries and operating systems.

6.1.3 Application-System Interaction in X/Motif

The essence of X programming is the handling of asynchronous events. Events can occur in any order, in any window, as the user moves the pointer, switches between the mouse and the keyboard, moves and resizes windows, and invokes functions through user interface components.

Xlib provides many low-level functions for handling events. For example, in the interactive rule debugger the "up" and "down" mouse events of the first and third mouse buttons are traced to handle user requests such as locating cursor position and zooming of a certain area in the drawing canvas.

Xt simplifies event handling by having widgets handle many events themselves, without any application interaction. Once all of the widgets for an application have been created and managed, a last statement, *XtAppMainLoop* will turn control of the application over to Xt Intrinsics and let Xt handle the dispatching of events to the appropriate widgets. However, an application can gain back the control for certain events if it registers for the events with callback routines. In this case Xt will pass the control to callback routines in which user specified action can be carried out.

6.2 Implementation of the MDP User Interface

- **Support for Temporal Events** Temporal events are a special kind of primitive events which are time-related. Temporal events can be categorized into absolute and relative event subclasses. An *absolute* temporal event is specified with an absolute value of time and is represented as: $< timestring >$. For example, 2 p.m. on August 15th, 1995 is specified as $< (14:00:00)08/15/1995 >$. A *relative* temporal event is specified by a reference point and an offset.

  Currently Sentinel supports temporal events using Unix signal handling services. The temporal event handler is a separate module independent of the

local composite event detector and the primitive event subscription module which is inside the extended Open OODB kernel.

As described in 6.2, MDP interacts with Sentinel asynchronously. When a temporal event is defined, the temporal event handling module is called to set timer. At the same time the user interface manages a *warning dialog* in green color stating that temporal event will happen either at an absolute time or after a definite time interval, depending on the event's nature. As the timer runs out a signal is triggered and the event handler starts executing. Reply message is sent to the front-end inside the event handler. At this point the MDP user interface notifies the user of the temporal event by changing the color of the warning dialog and the text content in the dialog.

- **Timeout Event - Blinking graphics** As mentioned in section 6.2, a newly arriving region-related action message will cause an appropriate rectangle box on the world map canvas to blink. This is achieved by utilizing Xt Intrinsics' signal handling services through invocation of the call *XtAppAddTimeOut*. This call registers a timer procedure that is called after a specified amount of time. In our case, when a new region-related message arrives the user interface installs a timer procedure which draws the target rectangle in a color. This timer procedure is set to run after 0.25 second. Inside this timer procedure another timer procedure is installed with a same time interval (0.25 second). The latter performs in the same way as the first procedure except that it draws the rectangle in white. These two timer procedures repeatedly invoke each other after waiting for 0.25 second. As a result the user envisions a blinking rectangle. The blinking will be stopped when a new message arrives by calling *XtRemoveTimeOut*.

6.3 Implementation of SIEVE

6.3.1 Layout and Functionality

The visualization tool's interface and functionality has been designed to provide as much information as possible in an uncluttered manner. Some information (e.g., event detection and rule execution) is provided as part of the visualization whereas other information (e.g., objects held by a rule/subtransaction) is provided on a demand basis.

The display area of SIEVE contains the following:

1. *Primitive Events*

2. *Composite Events*

3. *Rules*

   These are three scrolled lists containing the names of all primitive events, composite events and rules defined in an application. This information is provided by the preprocessor via a log file.

4. *Output*

   this is a scrolled text pane that gives the description of current execution of the system. Also, when an item inside a scrolled list is chosen by a mouse click, its detailed description will appear in this pane. The description includes, depending on what it is – for a primitive event, its method, modifier (when to notify in the execution of the method function – begin or end) and OID; for a composite event, its type, OID and constituent events (which can be primitive or composite); for a rule, its condition, action, priority, coupling mode, context, OID and the name of the associated event.

5. *Canvas*

   this is a drawing-area widget displaying the event tree, nested transaction tree and rule firing context graph. During a trace, the upper half of this drawing-area window shows the event tree, with leaf nodes (primitive events) on the top. Composite events link with their component nodes with straight lines. Initially, since no event has been detected/raised, all nodes are in the color of grey.

   The lower half of the drawing-area displays rule execution tree, which demonstrates the nested nature of transactions. Each node stands for a subtransaction. Different colors are used for the three states of sub-transactions: green for running, yellow for suspended and red for committed. Whenever a rule is fired a line connecting the transaction node of the rule and the triggering event is shown, and the color of the triggering event is changed to brown.

6. *Start Button*

   for starting debugging.

7. *Step Button*

   for choosing step debugging mode. In this mode the system will halt every time a system update happens. These updates include event detection, rule firing, start and commit of a transaction or subtransaction, acquirement and release of a database object, and encounter of a break point.

8. *Continuous Button*

   for choosing continuous debugging mode. In this mode the application will execute continuously until it meets the end or a break point.

9. *Help Button*

   for online help.

10. *Select Button*

    for selecting a subset of events or rules to trace.

11. *Global clock*

    shows global time in event detection.

12. *Dialogs*

    accept choices to be input at break points

6.3.2 GUI Implementation Notes

1. **Dialogs**

   Various popup dialog boxes are used to assist the user in making choices in different stages of execution. For example, At the beginning when the user click the "Start" button a question dialog box pops up with a toggle box to inform the user to choose between "Passive" and "Online" options. If "Passive" is chosen then the interaction between the visualization program and the Sentinel system is done using log files on a post-analysis base. Otherwise the two communicate via their socket interfaces. Another example is, when a break point is encountered the user has the following choices: (zero or many can be chosen)

   - *add a new event*
   - *add a new rule*
   - *update an existing event*
   - *update an existing rule*

Once a certain choice is made there may be additional more choices. For example, to update an existing rule, the next choice may be one or a combination of the following:

- *enable/disable the rule*
- *change its priority*
- *change its context*
- *change its coupling mode*

Prompting multiple choices and reading user response is achieved by dialogs. Dialogs help the user make decision intuitively and avoid errors. To further enhance user-friendliness we have added the "cancel" option at every stage where the user needs to make decisions.

2. **Mouse Events**

When the size of the event tree and subtransaction tree becomes large, the number of rectangles representing event and subtransaction nodes may fill the entire canvas area. The straight lines connecting events and rules add more complexity to the appearance of the picture. With all the lines, rectangles, text information and colors tangling together the user may have difficulty visualizing clearly. To overcome this potential problem we have implemented two features that will help the user in viewing the canvas:

- **graphical information querying**

  If the user move the mouse cursor inside a rectangle and click the first (left) mouse button, the corresponding information about the node will show in the output text pane. This gives the user a better knowledge of the nodes in the canvas. To support this feature, we let the first mouse

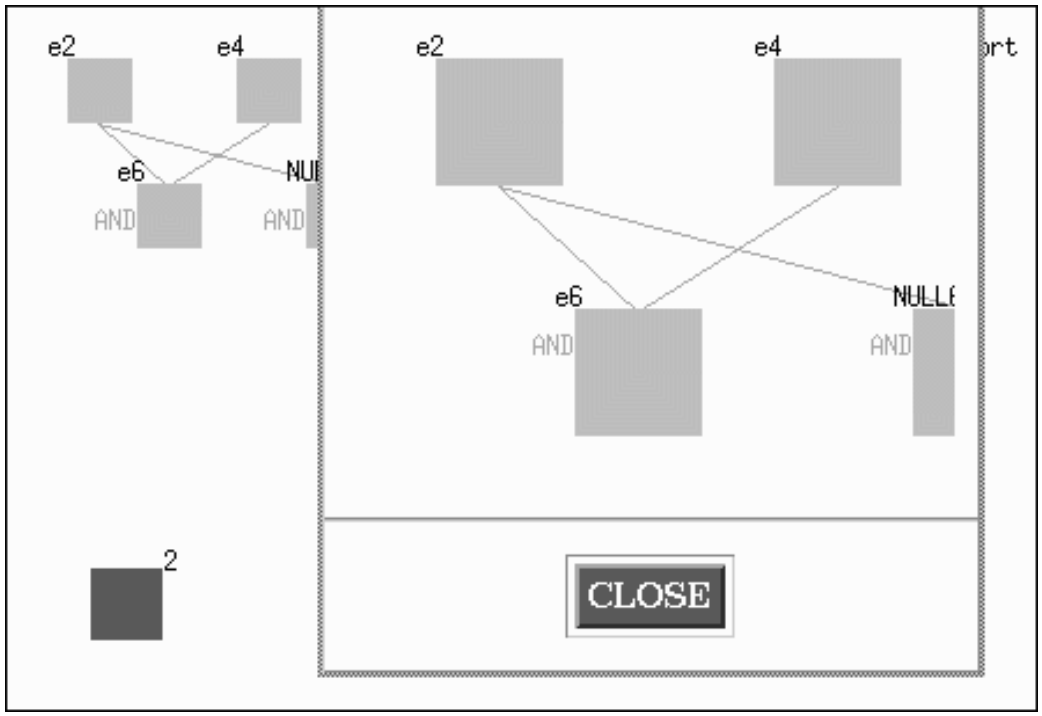

Figure 6.2. Zooming Feature of the Sentinel Rule Debugger

button event associate with an action procedure that catches the position of the mouse cursor and searches through the event tree and transaction tree data structure for a position match, then prints the information of the matched node on the text pane.

- **zooming**

  The other feature lets the user choose a small area of the canvas and zoom it in to a scale twice finer as the original scale. if the user presses down the third (right) mouse button at a starting point, holds and moves the mouse cursor to a second position and releases it, the area enclosed in the two points will be zoomed. The enlarged image will appears in a popup dialog box containing a drawing-area canvas.

  Figure 6.2 shows a part of the original canvas and the zoomed area.

3. **Auto-zooming**

   As the number of nodes drawn in the canvas increases the image size may exceed the boundary of the canvas. Though we can use scroll bar to move out-of-sight portions of the image back into the window, the entire picture can not be viewed at the same time, and the image size always has an upper limit. We have implemented an *automatic zooming* feature to solve this problem. Whenever the image size approaches the size of the canvas window, the drawing scale will be adjusted in such a way that the new image drawn using the reduced scale will remain in the canvas window.

4. **Pruning of the event and subtransaction trees**

   The user may be interested only in a subset of all events defined in an application. To avoid tracing all events exhaustively we have added a pruning feature to allow the user to specify certain events and/or rules to trace and let unrelated information screened away. This is accomplished by forming two reduced trees containing only the nodes in the selection. This is possible since events and rules are objects and moreover there is a one to one correspondence between rules and transactions. When the trees are truncated in this way a child node may not be an immediate subtransaction of its parent, but rather a descendant of the parent transaction; and sibling nodes may not be real siblings which stand for concurrent subtransactions.

   Figure 6.3 shows the entire rule graph for an application. Figure 6.4 shows the graph after pruning. In the pruned graph, only transaction nodes that are related to rules R2 and R3 are present.

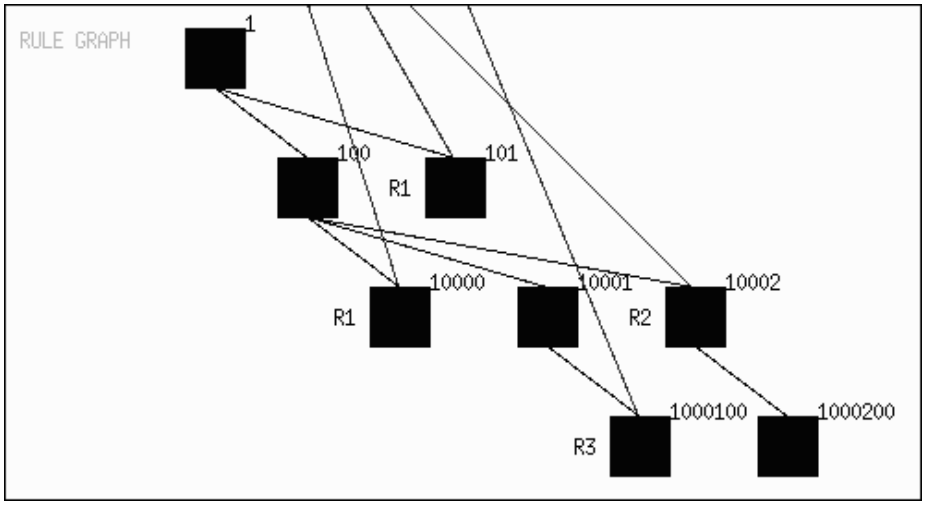

Figure 6.3. Before Pruning

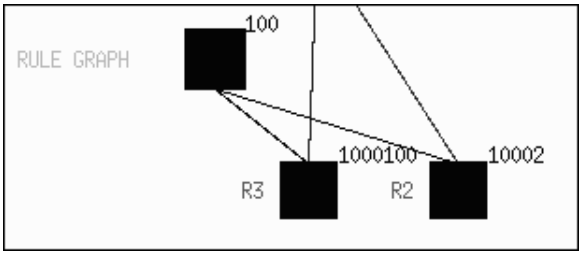

Figure 6.4. After Pruning

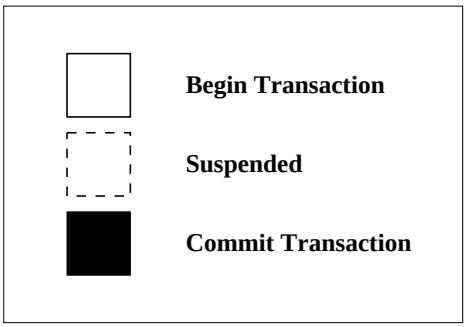

Figure 6.5. Graphical Representation of Subtransaction States for Monochrome Displays

5. **Hardware Portability**

   SIEVE enhances hardware portability by distinguishing between color and monochrome displays automatically and treating them differently. This is accomplished by the checking the default depth of the display. For color monitors the color representation of different subtransaction states remains consistent with that of the older implementation [26], but for monochrome displays the graphical representation has been modified as shown in Figure 6.5.

6.3.3 Implementation Issues on GUI-System Interactions

In this thesis we have implemented a debugger for active rules in an object-oriented context. The rule manager supports the event-condition-action paradigm. Apart from tracing the execution of rules the Rule debugger also keeps track of the events. As mentioned in Diaz et al. [12] tracing of events gives important hints to the user: the event-rule cycle allows the user to know not only which rules fired but also which event(s) caused the rule(s) to fire. The occurrence of the events sets the context for the rule execution. The following features of the rule manager of Sentinel have influenced the development of the Rule debugger.

- Events and Rules in Sentinel are treated as first class objects. Hence rule and event definitions are identified by their object identifiers, described by attributes and are manipulated through methods.

- Both events and rules can be defined either at class level or at instance level. The class level rules fire for all objects of that particular class when a certain situation (event-condition) is satisfied while the instance level rule fires only for a specific object.

- Events can be either primitive or composite. Currently, Sentinel supports primitive events as behavior invocations: either as global functions or as member methods. The system supports composite events by incorporating all the operators defined in Chakravarthy and Mishra [9].

- The condition and action are packaged into a single thread of execution. This thread is executed as a subtransaction of the parent triggering transaction. Hence we could typically have a cascaded firing of rules and there is a one to one mapping between the subtransactions and the rules.

- The nested transaction model supported by Sentinel allows sibling concurrency.

The functional architecture of the Rule debugger is as shown in Figure 6.6.
The input to the debugger is threefold:

- **Static information** Class instance level rule and event definitions are supplied by the user in the application program using the event [9] and rule [2, 19] definition language, which is incorporated on top of C++. As the definitions are preprocessed the preprocessor supplies static information in the form of a log file.

  The debugger requires the following information with respect to an event:

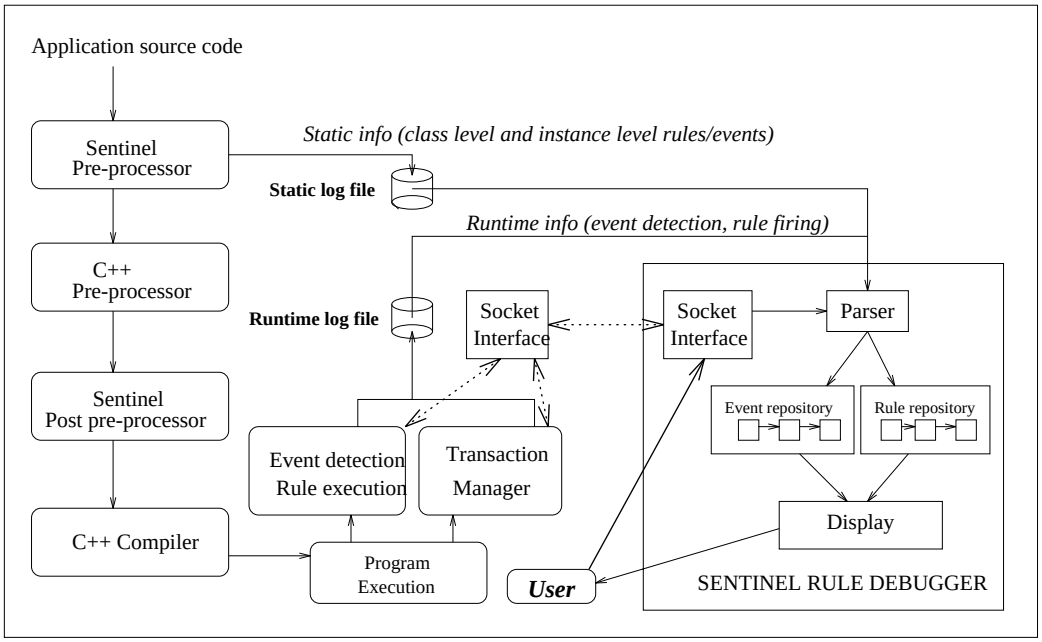

Figure 6.6. Functional Modules of the Revised Sentinel Rule Debugger

1. The user-supplied *name* of the event.
2. If it is a class level event then the *classname* with which the event is associated.
3. The *type* of the event (*primitive* or *composite*).
4. The signature of the *method* on which the event is defined along with the event modifier. (whether it is raised *before* or *after* the invocation of the method).
5. If the event is composite then the *event expression*.

Also the following information is provided to the debugger with respect to a rule:

1. The user supplied rule *name* and the *classname* to which it is associated if it is classlevel rule.

2. The signature of the methods implementing the *condition* and *action* of the rule.

3. The *context* for which the rule is fired (recent, continuous, cummulative, chronical), *coupling mode* (immediate, deferred, detached), rule *trigger mode* (now, before) and the rule *priority*.

- **Event detection information** This is the runtime information on the occurrence of events and the creation of event objects. In addition to the name and OID the event detector also provides *context parameter* of composite events to the rule debugger. This information is useful for demonstrating the *global event history graph.* [1] The way of transmitting information, depending on the debugging mode the user specified – either passive or online, can be different in implementation – by using files or stream sockets, but the appearance of the user interface will not differ. This approach hids the intricacies of implementation from the user.

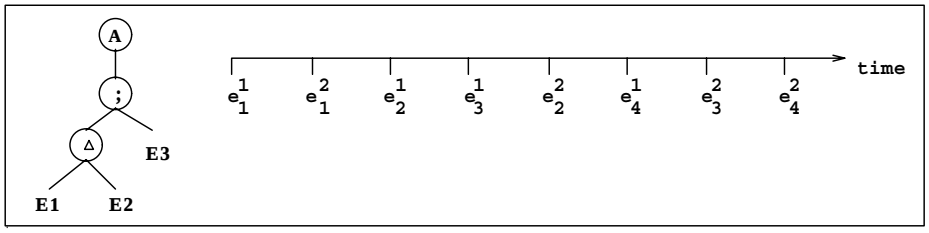

Figure 6.7. Global Event History

[1] Global event history (event-log) is a set of all primitive event occurrences and is denoted by H. Each primitive event occurrence is represented as a set in the log.

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

Illustrated in Figure 6.7 is the event expression `A = (E1` $\triangle$ `E2) ; E3` and the occurrences of different instances of event E1, E2 and E3 as well as the event graph for A.

- **Rule firing information** This is furnished to the rule debugger in the same way as the event detection information. The rule object identifiers and the user supplied rule names are associated when the rule objects are created.

- **Transaction information** The TID of the transaction in which a rule is fired is provided to the rule debugger by the transaction manager in Open OODB kernel. This association between the rule and the transaction helps the user visualize the nested transaction approach Sentinel adopted in rule execution.

- **Locking information** The ids of database objects which were accessed in the process of rule execution are provided by the lock manager. This information helps the user visualize rule-database interaction.

- **Break points** The user can indicate several break points in the application. When the preprocessor parses the application code, a new primitive event is defined as "break_point". At an actual break point this event is explicitly raised, detected and the rule debugger is notified of the break point. Then the debugger accepts input requests from the user and transfers these requests (textual strings) to Sentinel for processing. In implementation the rule debugger collects all of the user-input request messages and dispatches them one after another to the Sentinel socket interface **at once**. The parser in the system processes the textual messages and executes them accordingly as if they were statements of the application. As a result, the execution of these requests may trigger more events, fire more rules and alter the database, but the front-end has no knowledge of what may happen and only waits for output from the system asynchronously. This approach is strictly consistent with the event-condition-action nature of rules i.e. active behavior. Adding break points to the application enables the user to change and monitor the behavior of events and

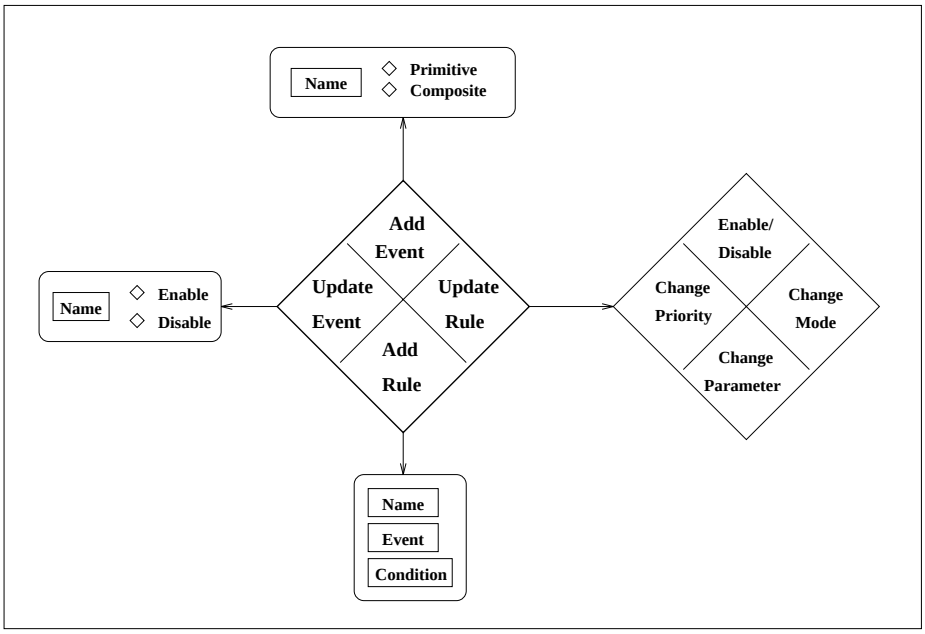

Figure 6.8. Control Flow at a Break Point

rules. This feature increases the interaction between the visualization front-end and the underlying system from the user's perspective.

Figure 6.8 shows the control flow at a break point.

SIEVE offers *uniform supports* for

- *step/batch mode tracing of rules:* The user can switch between step and batch mode during execution. When the step is chosen, the unit of consecutive execution is the interval between any event, rule or break point. When batch mode is chosen, the unit of trace is the entire application (if break points are ignored) or the interval between two break points (when break points are accepted). The user can enable/disable break points at run-time.

- *using information from preprocessing/post-analysis/run-time execution:* The user can specify if preprocessor should be used by the debugger. Also, he/she can choose from post-execution and run-time analysis before and a trace and

change this preference afterwards. SIEVE utilizes either files or sockets accordingly, but the appearance of the front-end remains the same.

- *context dependency on system/user:* The displayed rule execution information may be the result of the original application or dynamic input from the user at break points. SIEVE treats different context in the same asynchronous way.

The implementation requires the following changes to existing Sentinel modules:

- *preprocessor:* In order to support predefined break points in an application, such additional feature is added to the preprocessor that it recognizes break points and maps them into a kind of explicit primitive event. This approach enables the event detector to notify the user interface at run-time when break points occur. Moreover, the preprocessor creates a data structure which contains all procedures (pointers) that are potentially conditions/actions with their names (strings). This one-to-one correspondence between the names and pointers makes it possible to search pointers by name at run-time. This feature is extremely useful for run-time rule creation, when the condition and action part need to be defined dynamically without recompiling.

- *event detector:* A parser is added to process user input coming from SIEVE and translate the textual messages into appropriate execution statements. Besides, a socket interface is used for communication with SIEVE.

- *transaction manager:* A socket interface is needed to send transaction-related information to SIEVE.

6.3.4 An Example: Stock Demo

Consider the following code example:

- *before preprocessing:*

```
{
  ......
  event end(e1) int sell_stock(int qty);
  begin(e2) void set_price(float price);
  event e3 = e1^e2;
  rule R1[e3, cond1, action1, CUMMULATIVE, DEFERRED];
  ......
}
```

- *after preprocessing:* For the above application, the log file would contain the following information:

```
Event Stock Primitive e1 [int sell_stock(int qty)] end
Event Stock Primitive e2 [void set_price(float price)] begin
Event Stock Composite e3 e1 AND e2
Rule R1 Stock cond1 action1 CUMULATIVE DEFRRED NOW e3
......
break
```

The preprocessed code for the events and rules is:

```
{
//pre-define break_point as a primitive event;
break_point = new PRIMITIVE("break_point", "OODB", "end", "break");
......
//instantiate events and rules that are defined in the application;
PRIMITIVE * Stock_e1 = new PRIMITIVE("e1", "Stock", "end", "sell_stock");
PRIMITIVE * Stock_e2 = new PRIMITIVE("e2", "Stock", "begin", "set_price");
AND * Stock_e3 = new AND("e3", Stock_e1, Stock_e2);
RULE * Stock_R1 = new RULE("R1", Stock_e3, cond1, action1, CUMULATIVE);
Stock_R1->set_mode(DEFERRED);
......
//explicitly raise break_point event to notify the front-end;
Notify(NULL, "OODB", "break", "end", system_list);
......
}
```

- *after execution:* The run-time log file used for post-analysis may has the following contents:

```
......
e2
......
e1
e3
R1
......
BREAK
......
```

Figure 6.9 shows a snapshot of the canvas when subtransaction 10001 has just started. Composite event STOCK_e4 has triggered rule R1, whose corresponding subtransaction 10000 has committed. The top-level transaction 1 and subtransaction 100 are being suspended to allow for nested transactions.

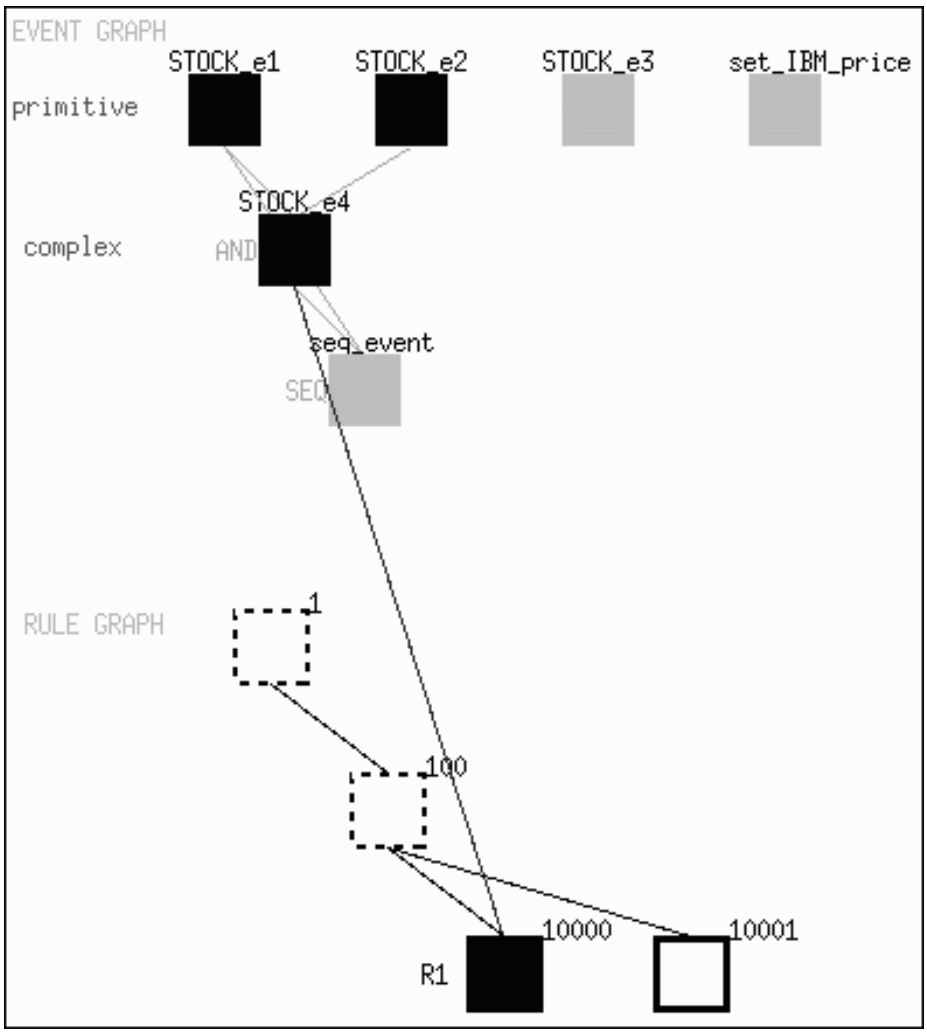

Figure 6.9. A Snapshot of Trace

CHAPTER 7
CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this thesis we have addressed the visualization issues on active databases. Emphasis has been focused on the taxonomy between application-level and system-level user interfaces to Sentinel, an active object-oriented database management systems.

As an illustrative example, we have designed and implemented MDP, an action-oriented application-level user interface. We have discussed the impact of the application's requirement on the architecture, layout, functionality and the interaction with the rule system.

A more innovative contribution of this thesis is the extension of the rule debugger to support interactive event/rule visualization. The debugging tool – SIEVE supports visualization of event detection and rule execution both at run-time and in batch post-analysis mode. We have revised the overall architecture to reflect and monitor the execution status of key modules of the Sentinel system, with emphasis on the context of event-rule relationships. In order to assist the user in understanding the rule mechanism the rule debugger is augmented with explanation features by furnishing the user with relavent information including

- event objects and tree structure
- transaction/rule objeccts and tree structure
- event detection context parameters
- global event history graph

- lock information of database objects being held/released by subtransaction

The visualization tool enables the user to interrupt the execution at "break points" and input feedbacks to the rule system, therefore conters the difficulty of run-time tracing caused by the asynchronous nature of ECA rules. The user-initiated changes include

- add new events
- disable/enable existing events
- add new rules
- disable/enable existing rules
- change rule priority, context, and coupling mode

The functionality of the interactive rule debugger has been further improved in the following aspects:

- portability among hardware platforms
- on-line help
- automatic zooming
- illustrative dialogs

7.2 Future Research

The near-future goal of the extension to the Sentinel OODBMS is to upgrade it to support global active database applications in which events can be defined distributedly accross many user applications. The overall architecture is proposed in Tamizuddin [26] and illustrated in Figure 7.1. Each application has a local event

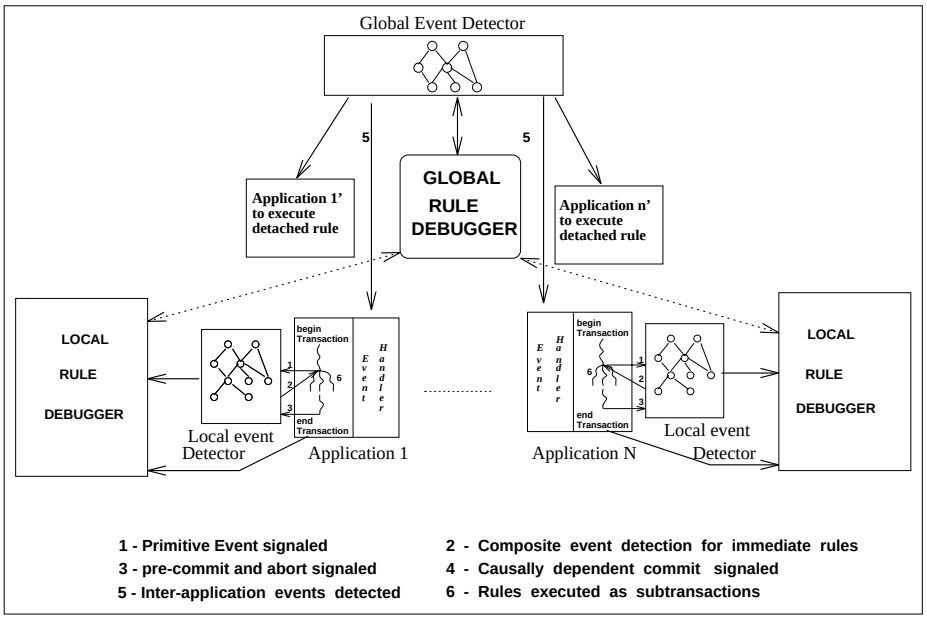

Figure 7.1. Overall Architecture for Global Event Detection and Rule Visualization

detector to which all the primitive events are signaled. In addition, each application has a *global event handler thread* that handles the execution of rules whose events span across applications (which can be running at distant address spaces). The core functional module that is needed for accomplishing global event detection will be the *global event detector*, which is a separate process running remotely. When a primitive event occurs, it is notified to the local event detector. The application waits for the signaling of any composite event of immediate mode. In the mean time the global event detector communicates with the local event detectors via remote procedure calls (*RPC*s) to receive local events, detects the occurrences of global events and notify the appropriate local event detectors of these occurrences.

The role of the rule debugger in this environment is to monitor global events as well local events. It needs to communicate with not only the local event detector and rule manager of each application but also the global event detector. This requirement raises several design and implementation issues listed in the following:

- Choice of communication protocol and architecture

- Network security and authentication
- Clarity of display – avoiding the clustering of information

Besides global visualization we also plan on incorporating static analysis tools as part of the visualization toolkit so that runtime execution can be compared with static analysis. The preprocessor may play an important role in supporting static analysis.

Also, we plan to add interactive query language capability to view database states. The user interface will be tuned to perform data-oriented tasks. Ideally (i.e., for the long term), it would be useful to have a cumstomizable visualization toolkit to which the user can specify expected behavior and the tool provides a visual feedback on how the actual execution differs from the specification and offers guidance for correction.

REFERENCES

[1] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proceedings, International Conference on Management of Data*, pages 59–68, May 1992.

[2] E. Anwar. Supporting Complex Events and Rules in an OODBMS: A Seamless Approach. Master's thesis, CIS Department, University of Florida, Gainesville, FL, November 1992.

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

[4] M. Arya, N. Grady, G. Grinstein, P. Kochevar, D. Swanberg, V. Vasudevan, L. Wanger, A. Wierse, and M. Woyna. Database Issues for Data Visualization: System Integration Issues. In *Proc. IEEE Visualization Workshop*, San Jose, 1993.

[5] H. Behrends. Simulation-based Debugging of Active Databases. In *Proc. IEEE Workshop on Research Issues in Database Engineering, RIDE 94*, Houston, 1994.

[6] P. Borras, J. C. Mamou, and D. Tallot. Building User Interfaces for Databases Applications: The O2 Experience. *SIGMOD Record*, 21(1):32–38, March 1992.

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

[8] S. Chakravarthy, V. Krishnaprasad, Z. Tamizuddin, and R. Badani. ECA Rule Integration into an OODBMS: Architecture and Implementation. Technical Report UF-CIS-TR-94-023, University of Florida, Gainesville, FL, Feb. 1994. (In ICDE-95, Taiwan, March 1995.).

[9] S. Chakravarthy and D. Mishra. An Event Specification Language (Snoop) for Active Databases and its Detection. Technical Report UF-CIS TR-91-23, University of Florida, Gainesville, FL, Sep. 1991.

[10] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. Technical Report UF-CIS-TR-93-007, University of Florida, Gainesville, FL, March 1993. (Revised and extended version of UF-CIS-TR-91-23.).

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

[12] O. Diaz, A. Jaime, and N. W. Paton. DEAR: A DEbugger for Active Rules in an Object-Oriented Context. In *Proc. of the 1st International Conference on Rules in Database Systems*, September 1993.

[13] O. Diaz, A. Jaime, N. W. Paton, and G. al Quimari. Supporting Dynamic Displays Using Active Rules. *SIGMOD Record*, 23(1):21–26, Mar. 1994.

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

[15] W. Hseush and G.E. Kaiser. Modelling Concurrency in Parallel Debugging. In *ACM S IGPLAN Notices*, number 3, pages 11–20, March. 1990.

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

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

[18] R. King and M. Novak. Building Reusable Data Representations with FaceKit. *SIGMOD Record*, 21(1):11–17, March 1992.

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

[20] M. Kuntz. The Gist of GIUKU Graphical Interactive Intelligent Utilities for Knowledgeable Users of Data Base Systems. *SIGMOD Record*, 21(1), March 1992.

[21] M. Linton. A Debugger for the Berkley Pascal System. Master's thesis, University of California at Berkeley, June 1981.

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

[23] P. Neeta. An X-based Graphic Browsing Interface for OSAM*. Master's thesis, CIS Department, University of Florida, Gainesville, FL, 1992.

[24] J. Paredaens, M. Andries, M. Gemis, and M. Gyssens. An Overview of GOOD. *SIGMOD Record*, 21(1):25–29, March 1992.

[25] A. Sharma. On Extensions to a Passive DBMS to Support Active and Multimedia Capabilities. Master's thesis, CIS Department, University of Florida, Gainesville, 1992.

[26] Z. Tamizuddin. Rule Execution and Visualization in Active OODBMS. Master's thesis, CIS Department, University of Florida, Gainesville, FL, May 1994.

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

[28] J. Widom and S. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proceedings, International Conference on Management of Data*, pages 259–270, May 1990.

BIOGRAPHICAL SKETCH

Jun Zhou was born on January 29, 1968, at Shanghai, China. He received his Bachelor of Science degree in physics from the University of Science and Technology of China, Hefei, China in July 1990. He also received his Master of Science degree in physics from Tulane University, New Orleans, in May 1993. He will receive his Master of Science degree in computer and information sciences from the University of Florida, Gainesville, in August 1995. His research interests include active, object-oriented databases and graphical user interfaces.