ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this interesting topic and for providing me great guidance and support through the course of this research work. I am also thankful to Dr. Eric Hanson and Dr. Herman Lam for serving on my committee.

I would like to express my special thanks to Sharon Grant and Hyoungjin Kim for maintaining a well administered research environment and being so helpful in times of need. I am grateful to Shiby Thomas and Jennifer Sung for their invaluable help and fruitful discussions during the design and implementation of this work. Also I would like to thank all my friends for their constant support and encouragement.

I would like to thank the Office of Naval Research, the Navy Command, Control Ocean Surveillance Center RDT&E Division, and National Science Foundation for supporting this work.

Last, but not the least, I thank my family for their endless love. Without their support, this work would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

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

A FLEXIBLE DYNAMIC ECA RULE EDITOR FOR SENTINEL: DESIGN AND IMPLEMENTATION

By

Hung-Ju Chu

May 1998

Chairman: Dr. Sharma Chakravarthy

Major Department: Computer and Information Science and Engineering

A wide range of emerging applications that tend to be large and need timely response have prompted the development of active object-oriented database management systems (OODBMS). An active OODBMS uses event-condition-action (ECA) production rules that specify events to be monitored and their reaction to those events under certain conditions. The active OODBMS can be implemented under compilation-based or interpreter-based environment. One of the disadvantages of a compilation-based system is that we need to reedit, recompile, and link an application's source code to make any change visible. This implies that users have to painstakingly go through the procedure mentioned above to add, delete, or modify rules in a compilation-based active OODBMS. This becomes very unpractical especially if the users need to manipulate the rules frequently. Hence, managing the rules in an easy and efficient manner plays an important role in the success of an active OODBMS.

This thesis proposes a flexible design and implementation of Dynamic ECA Rule Editor for external rule support in Sentinel. The editor provides users with a friendly environment to manipulate the rules without changing, recompiling, or relinking any source code of their applications. Java language is used to implement the interface module that allows the editor to be used either from within a web browser or as a stand-alone application. To facilitate expansion and porting, OQL and a tiered client-server architecture are adopted to decouple the dependency among the three basic database application elements: presentation (interface logic), functionality (application logic), and data (database logic).

The OQL preprocessor has been modified to complete its functionality before it is applied to the implementation of the Rule Editor. Several bugs resulting from wrong C++ code translation have been fixed. The fixed clauses include UPDATE, DELETE, IN, WHERE, JOIN, and Subquery. INSERT clause and the capability of querying on transient objects have been extended. To facilitate the understanding of the OQL and to allow ad hoc queries, several interface programs has been implemented.

CHAPTER 1
INTRODUCTION

A wide range of emerging applications that need timely response entails developing active database management systems (DBMS). These applications, such as computer aided manufacturing, power distribution network, automated office workflow control, and air traffic control, need to continually monitor their database state and quickly respond with proper actions to certain events. Traditionally, to meet this requirement, either polling was used to determine whether the situation being monitored has occurred or each program that updates the database was augmented to check the situation being monitored. The former approach is not efficient because of the overhead of the polling. The latter approach compromises modularity and reuse of code. Active DBMSs have been proposed to solve these problems.

An active DBMS typically adopts event-condition-action (ECA) production rules that specify events to be monitored and their reaction to those events under certain conditions. This self-monitoring and self-reactive features seamlessly integrate with traditional DBMSs and elegantly provide the means to meet the various needs of today's applications. Most of the earlier research and commercial implementations of active databases focused on extending the active capability in the relational database systems. Recently, the focus has been moved to object-oriented database management systems (OODBMSs) to meet the needs of more complex applications, like image and graphic databases, scientific databases, and multimedia databases. The object-oriented paradigm provides more powerful capabilities to model complex objects.

An active OODBMS can be implemented under compilation-based or interpreter-based environment. The two systems have tradeoffs in terms of efficiency and productivity. One of the advantages of using the interpreter-based system is that it makes source-level debugging operations easier because all run-time errors refer to source-level. This environment readily accommodates incremental changes to code and dynamic specification of code. On the other hand, applications developed in the compilation-based environment can be run very efficiently because they are already in the binary-level. However, their source codes need to be compiled and linked before they can be run.

Sentinel is a compilation-based active OODBMS. As mentioned earlier, one of the disadvantages of the compilation-based system is that we need to reedit, compile, and link applications' source codes to make changes visible. This implies that the users have to painstakingly go through the procedure mentioned above to make any change on rules in a compilation-based active OODBM. This becomes impractical, especially if the users need to manipulate the rules frequently. Hence, managing rules in an easy and efficient manner plays an important role in the utility of an active OODBMS.

There is a need for a tool to manipulate the rules of existing applications without going through the steps involved in the compilation-based system. To meet the need, a Dynamic Rule Editor for Sentinel (DRES) [1] with Tcl/Tk interface was developed. However, the DRES has several limitations resulting from its underlying architecture and the design itself. They are summarized as follows:

- Interface logic and application logic are mixed together on account of the Tcl/Tk usage. It compromises the modularity and makes porting, maintenance, and expansion difficult. Besides, it limits that the interface must be run on the same machine in which the application logic is running. When multiple users access the editor simultaneously, multiple application logic modules need to be invoked during the whole sessions, which wastes system resources.

- A function-pointer data structure is maintained. The data structure is used to associate a rule with condition/action functions at run time. Whenever the user makes any change on rules, the structure needs to be compiled and made as library for later references, which is time-consuming and unnecessary.
- Applications still need to be relinked to make any change on rules visible.
- Rule database of DRES is represented by structures instead of classes. It does not take advantage of the OQL capabilities of an OODBMS.
- The editor only supports rules with primitive events (only method events). Rules with temporal, composite, or external events also need to be supported.

This thesis is primarily motivated by these limitations. It proposes a flexible design and implementation of Dynamic ECA Rule Editor for external rule support to Sentinel. The editor provides the users with a friendly environment to manipulate the rules without changing, recompiling, or relinking any source code of their applications. Java language is used to implement the interface module that allows the editor running either within a WWW browser or as a stand-alone application. To facilitate expansion and porting, Object Query Language (OQL) and a tiered client-server architecture are adopted to decouple the dependency among the three basic database application elements: presentation (interface logic), functionality (application logic), and data (database logic).

The remainder of this thesis is organized as follows: Chapter 2 presents an overview of current rule support in Sentinel. The rules supported by Sentinel can be classified into two types, namely internal (static) rules and external (dynamic) rules. How the external rules work through the Rule Editor is discussed in the last section of this chapter. Chapter 3 discusses some design issues, analyzes their advantages and disadvantages, and describes how they are applied in the design of the Rule Editor. In chapter 4, we discuss more details about the back-end design and implementation. Chapter 5 has a conclusion along with its discussion of the limitation and future direction of the Rule Editor. As OQL is used in the implementation of the editor and it is also one of the contributions of this thesis, it is discussed in the appendix as a separate chapter.

CHAPTER 2
OVERVIEW OF SENTINEL RULE SUPPORT

Sentinel is an integrated active OODBMS that incorporates ECA rules using Open OODB Toolkit [2] under C++ environment. ECA rules are seamlessly incorporated into C++ language through Snoop [3] language. Snoop defines event and rule specification, classifies events into a class hierarchy, and supports several event operators for constructing composite events.

2.1 Event Modifiers

Conceptually, an event is an atomic occurrence. It happens completely or not at all. From the point of view of implementation, it is necessary to choose a point (or several points) as an event along a sequential computation during a closed interval. Event modifiers [4] are used to create one or more events that correspond to points of interest in a closed interval. Snoop supports two modifiers, *begin-of* and *end-of*, to map an arbitrary interval on the time line into two events.

2.2 Event Classification

Events can be organized into a hierarchy of event classes based on their structure and behavior. Each event class represents a unique event type. For example, in an object-oriented database system, there may be a method, named **Insert**, defined in a class, named **Stock**. Every instance of the **Stock** class may potentially invoke its **Insert** method and produce an event, such as *begin-of* **Insert**. We can classify the events originating from the **Insert** method into an event class.

In a distributed environment, events can be broadly classified into local and global events. Local events are further classified into local primitive events and local composite events. Similarly, global events are further classified into global primitive and global composite events. This classification is illustrated in Figure 2.1.

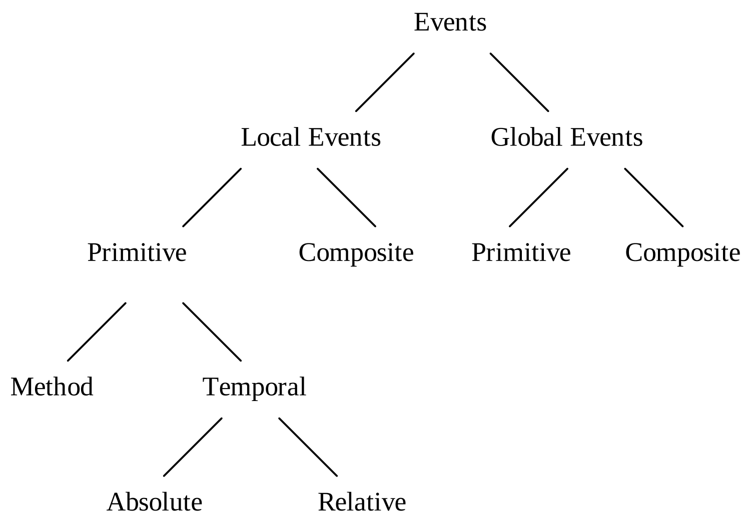


Figure 2.1: Event Classification

Local primitive events are predefined in an application and detected by a mechanism embedded in the system [5], while *Global primitive events* are defined and detected outside of the application and are referenced by an application in a distributed system [6]. *Primitive events* can be further classified into *method events* and *temporal events*. In the object-oriented paradigm, objects can be encapsulated by data (attributes) and methods (behavior). In Sentinel, every method of a class in the application potentially can be specified as an event by the event modifiers. They are referred to *as method events*. A *temporal event* is an instance of temporal event class [7]. It is defined by specifying a point on the time line. Essentially, there are two ways of specifying a point on the time line: using an absolute value and using an offset from a well-defined point on the time line. An *absolute temporal event* is specified with an absolute value of time. For example, <11:35:55/10/6/97> is an *absolute temporal event*. A *relative temporal event* is explicitly specified by a reference point and an offset time value. The reference point may be any event that can be specified in Snoop including an absolute temporal event. For example, <an-event-name + 5 sec> is a *relative temporal event*. Both the *method event* and the *temporal event* can be local or global depending on whether their definitions, detection, and references are within an application or not. A *composite event* is defined as an event obtained by the application of an event modifier to a *composite event expression* [8]. *A composite event expression* is defined as an event expression formed by using a set of primitive event expression, event operators and composite event expression constructed so far. Again, a composite event can be local or global. If at least one of the constituent events of the composite event is a global event, which is either composite or primitive, the composite event is regarded as a *global composite event*. Otherwise, it is a *local composite event*. Figure 2.2 shows the event operators supported in Snoop. The following section describes each of these operators and their semantics.

2.3 Event Operators

Snoop supports ten event operators so far. The following is the summary of the operators with brief explanations:

1. AND : Conjunction of two events, namely E1 and E2, denoted by E1 ^ E2, occurs when both events occur. The order of occurrence of E1 and E2 is irrelevant.

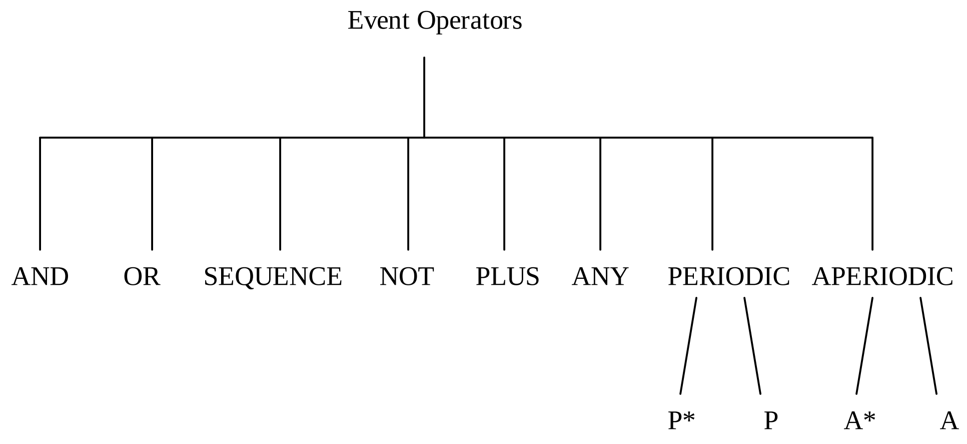


Figure 2.2 Event Operators

2. OR: Disjunction of two events, namely E1 and E2, denoted by E1 ∇ E2, occurs when either E1 or E2 occurs.

3. SEQUENCE: Sequence of two events, namely E1 and E2, denoted by E1>>E2, occurs when E2 occurs after the occurrence of E1.

4. NOT: NOT operator detects non-occurrence of an event, namely E2, in the closed interval formed by two events, namely E1 and E3, denoted by ¬E2[E1,E3].

5. ANY: The event , denoted by ANY(m,E1,E2,…..,En) where m <= n, occurs when m events out of n distinct events specified occur, ignoring the relative order of their occurrences.

6. A (Aperiodic): A aperiodic event, denoted by A(E1,E2,E3), is detected for every occurrence of E2 during the half-open interval formed by E1 and E3.

7. A*: It is a cumulative variant of A operator. A aperiodic-star event, denoted by A*(E1,E2,E3), is detected when E3 occurs provided E1 has already occurred. The occurrences of E2 are accumulated during the half-open interval formed by E1 and E3.

8. P (Periodic): A periodic event, denoted by P(E1,E2,E3) where E2 is a relative temporal event, is detected for every time period specified by E2 during the half-open interval (E1,E3].
9. P*: It is a cumulative variant of P operator. A periodic-star event, denoted by P*(E1,E2,E3) where E2 is a relative temporal event, is detected only once when E3 occurs provided the E1 has already occurred. The time specified in E2 is accumulated whenever E2 occurs.
10. PLUS: It is a variant of SEQUENCE operator. A PLUS event, denoted by E1 + [T] where [T] is a relative temporal event, is detected once a time interval [T] elapsed after the occurrence of E2.

2.4 Parameter Contexts

Snoop introduces the notion of parameter contexts [8] to capture application semantics for computing parameter or consuming event occurrences (of composite events) when they are not unique. These contexts are precisely defined using the notion of initiator and terminator events. An initiator of a composite event is a constituent event that can start the detection of the composite event whereas a terminator is a constituent event that can detect the occurrence of the composite event. A composite event may be comprised of several primitive events that can be initiators and terminators of another composite events. When the occurrences of several primitive events constitute a composite event, different possibilities for detecting composite events exist. By carefully analyzing several classes of applications, four parameter contexts are proposed in Snoop:

1. Recent : In this context, only the most recent occurrence of an initiator for any event is used. All other occurrences of a constituent event will not be used in

detecting another composite events and they will be deleted when the event started by the initiator occurs. The initiator of the event will continue to initiate another event until another initiator occurs.

2. Continuous: Each occurrence of the initiator of an event continuously initiates the event. In this context, a terminator may terminate one or more occurrences of the same event.
3. Chronicle: The initiator and terminator pair of a composite event is unique and deleted after the event occurred. They are paired based on chronological basis.
4. Cumulative: In this context, for each constituent event, all occurrences of the event are accumulated until the composite event is detected. In other words, the parameters of a composite event include the parameters of all the occurrences of each constituent event. All the occurrences of each constituent event are flushed whenever its associated composite event is detected.

2.5 Support for Rules

In the context of Sentinel, a rule consists of an event, a condition function of boolean type, an action function of void type, and a few attributes. Once an event is detected by the system, the associated condition function will be evaluated and the associated action function will be executed based on the result of the evaluation. Sentinel supports nested rules. When a rule's action raises an event that triggers other rules there is a nested execution of rules. As part of the rule semantics, it is necessary to know whether to execute condition and action functions immediately after the associated event has been raised. This is an issue about the coupling between the event and the condition-action pair [5]. Currently, there are three *coupling modes* supported by Sentinel, namely,

1. Immediate mode
2. Deferred mode
3. Detached mode

Sentinel supports multiple rules. An event can trigger several rules. Therefore, it is necessary to support rule execution mode that supports concurrent and prioritized serial execution of rules. Sentinel uses *Priority* classes for specifying rule priority. An arbitrary number of priority classes can be defined and totally ordered. A rule is assigned to a priority class by indicating its number or the name of the class. Sentinel provides a global conflict resolution mechanism among the priority classes and concurrent execution of rules that belong to the same priority class.

Sentinel also introduces two types of rule trigger modes for specifying the time from which event occurrences are to be considered for the rule. The two options are:

1. Now: Start detecting all constituent events starting from this time instant.
2. Previous: All previous constituent events are acceptable

The rules supported by Sentinel are specified in the classes (class-level) and the code (instance-level) of an application. They are referred to as internal (static) rules. The following section introduces the preprocessing of the internal rules.

2.6 Internal Rule Processing

The processing of the internal rules goes through the following steps:

- Designing and Editing Phase

  The user designs and edits his application by specifying the events and the rules for the application using Snoop, along with the class definitions. For example, the following code is the interface of a reactive class that specifies a class-level PLUS composite event and a class-level rule.

```
class STOCK: public REACTIVE
{
……
event end(e4) int get_price();
event e_plus = e4 + [0 months + 0 days + 0 hrs + 0 mins + 5 secs];
rule R1[e_plus, cond1, test_action3, RECENT];
…….
}
```

- Sentinel Processing Phase

The source code edited in the first phase is then processed by the Sentinel preprocessor (Spp). For example, the specification for the PLUS event and the associated rule is transformed into corresponding class constructors in the main function of the application by Spp as follows:

```
main{
{
…..
PRIMITIVE *STOCK_e4 = new PRIMITIVE( "STOCK_e4", "STOCK",
"end","int get_price()");
PRIMITIVE *STOCK_rel1 = new PRIMITIVE( "STOCK_rel1",
 "TEMPORAL", "","0 months + 0 days + 0 hrs + 0 mins + 5 secs");
PLUS *STOCK_e_plus = new
PLUS("STOCK_e_plus",STOCK_e4,STOCK_rel1);
RULE *R1 = new RULE("R1", STOCK_e_plus, cond1, test_action3, RECENT);
…
}
```

Note: PRIMITIVE and PLUS are two of the eleven event classes currently supported by Sentinel. The rest of them are REMOTE, AND, OR, SEQ, A, P, A_star, P_star, and NOT. They are all inherited from EVENT class.

These EVENT and RULE class constructors build an event-rule graph that is used for the event detection and the rule execution. The event detection logic and the rule execution logic are implemented in the body of these classes, which are beyond the scope of this thesis. They are discussed in other Sentinel papers [6][7][8][9].

- OpenOODB and C++ Compiler Processing Phase

  The code is then processed by OpenOODB preprocessor and finally compiled by a C++ compiler into an executable. The details are discussed in [7].

2.7 External Rules

As we have seen, the internal rules are part of the application source code. We need to go through the steps mentioned above to make any change to the rules visible. The idea of the external (dynamic) rules is mainly motivated by this limitation. Conceptually, the external rules are persisted in a database and inserted into an application when it is invoked. How we implement the external rules can be briefly described as follows: A function, namely load_dyn_rules, is inserted, by Spp, into the beginning of the application's main function. It will be executed whenever the application is invoked. Based on some persisted information outside of the application, the function creates events and rules by calling the RULE and EVENT class constructors. The Rule Editor proposed in this thesis is about the design and implementation of the ways to collect, persist, and use the information for creating the events and the external rules. One of the primary difficulties with the external rules is how the Rule Editor associates a rule with its condition/action functions at run time. This is solved by using a dynamic linking mechanism supported by Solaris operating system on which the external rule loader is running. We persist the function names and make the functions as DLL through the Rule Editor. The load_dyn_rules function uses the names to retrieve the function pointers by using the utilities, dlopen and dlsym, at run time. The details of the Dynamic Rule Loader and the way to avoid C++ name mangling will be discussed in section 4.5. Figure 2.3 illustrates the ways to create the internal and the external rules.

The Rule Editor mainly consists of three parts: (i) a way to collect the information (the GUI), (ii) a way to persist the information(the application logic and the rule database), and (iii) a way to use the information for creating the events and the rules (the Dynamic Rule Loader). Three design issues come to mind: the architecture to build the three parts, the choice of the language used to implement the GUI, and the database management system used to implement the rule database. The following chapter discusses these design issues and describes how they are applied to the design of the Rule Editor.

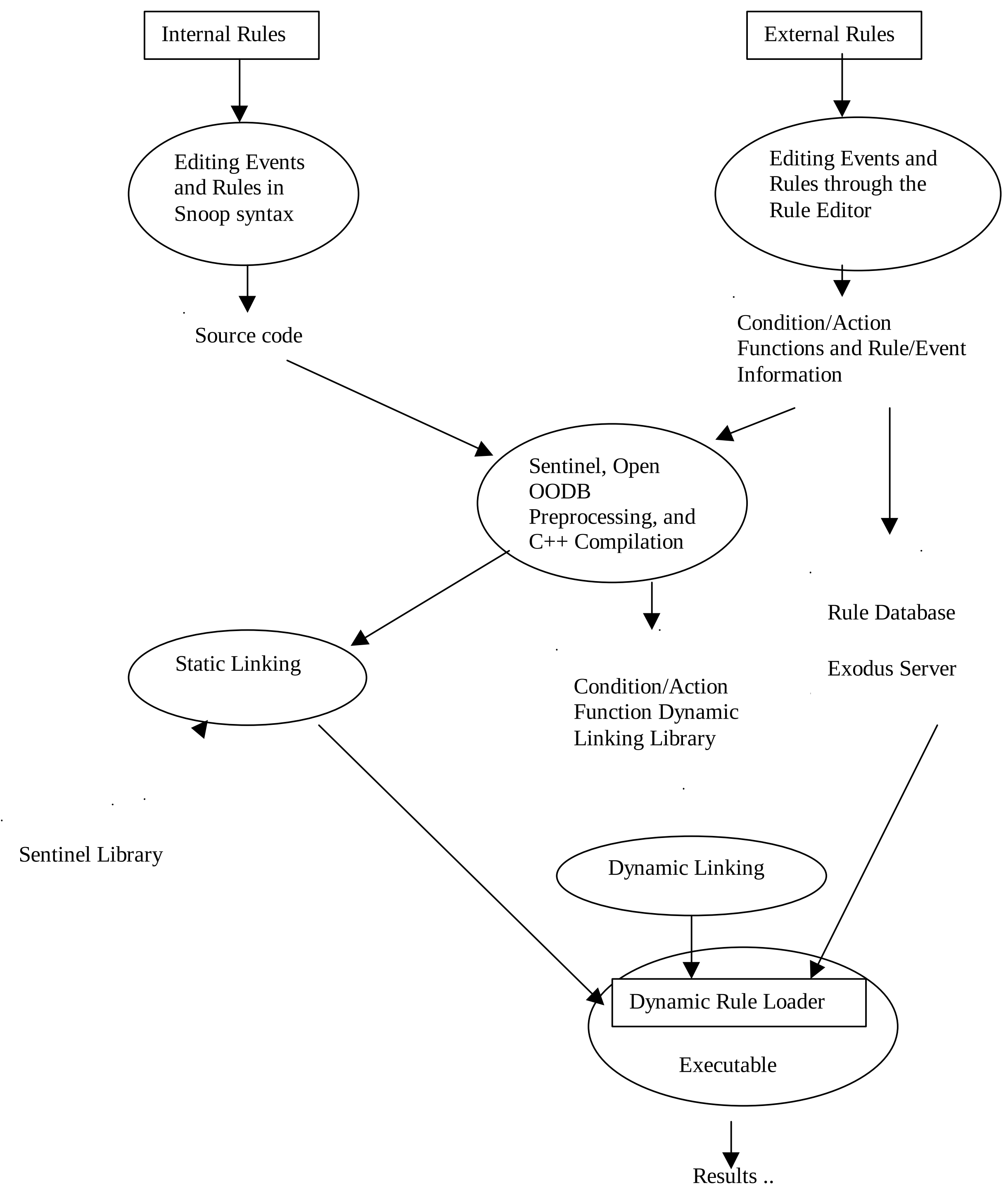

Internal Rules
External Rules
Editing Events and Rules in Snoop syntax
Editing Events and Rules through the Rule Editor
Source code
Condition/Action Functions and Rule/Event Information
Sentinel, Open OODB Preprocessing, and C++ Compilation
Static Linking
Condition/Action Function Dynamic Linking Library
Rule Database
Exodus Server
Sentinel Library
Dynamic Linking
Dynamic Rule Loader
Executable
Results ..

Figure 2.3 The Creation of the Internal and the External Rules

CHAPTER 3
DESIGN ISSUES

The Dynamic ECA Rule Editor for Sentinel aims at the following design goals: easy to maintain and expand, easy to use, efficient, concurrent access, portability, and usage from web. Based on these goals, this chapter discusses some design issues including the tiered client-server architecture, the use of Java language, and the Exodus Storage Manager. The last section describes how they are applied to the design of the Rule Editor.

3.1 Architecture

Modularity is the key feature to make programs easy to maintain and expand. To make an application highly modular, one solution is partition the application logic.

3.1.1 Client-Server Paradigm

The client-server paradigm is used by almost all network applications in the distributed computing world. The paradigm is basically motivated by the problem of rendezvous. To understand the problem, we can imagine how difficult it is to have two processes communicate to each other without one side waiting for the other side to contact. In the client-server paradigm, network applications are broadly divided into two categories, namely, a client and a server. A program that initiates a peer-to-peer communication is called a *client,* while a program that infinitely waits for a client's contact is called a *server*.

It is not necessary that a program is always a client or a server. It may play both the roles. It is not unusual to find a server for an application acting as a client for another. Hence, there exists the concept of tiers of the client-server relation.

3.1.2 Commonly Used Client-Server Architecture

Early GUI database applications usually adopt single-layered client-server architecture. The architecture is illustrated in Figure 3.1.

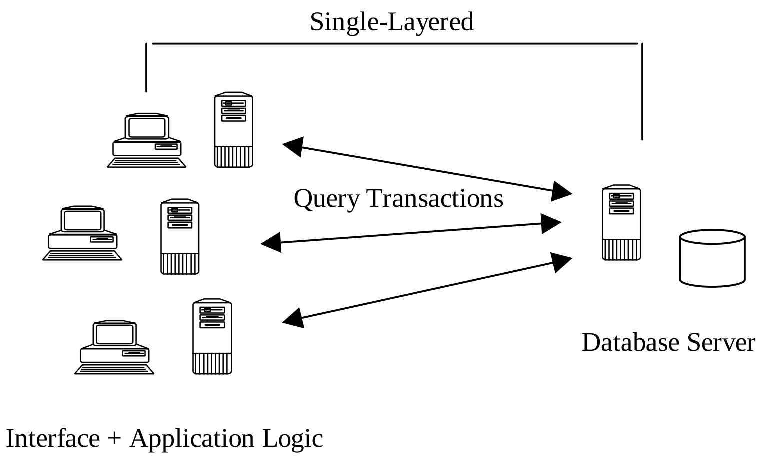


Figure 3.1 Commonly Used Client-Server Architecture

In the architecture, interface logic and application logic are deeply coupled to each other which results in the following limitations:

- Network traffic and CPU costs are expected to increase when multiple users access concurrently. Both interface logic and application logic are needed to be invoked during the whole session for every user when multiple users are accessing simultaneously, which increases the CPU and other resource costs. In addition, the remote database transport protocols used to carry the transaction usually results in large network traffic for every query. To mediate interaction between clients and the server among multiple query transactions, the effective bandwidth of network, and thus the query transaction processing speed, is reduced

- The coupling of interface logic and application logic compromises modularity and thus makes maintenance, expansion, and porting difficult.

As merging applications tend to be large and distributed, maintenance, cost, and efficiency become important issues. A double-layered architecture has been widely used as a solution to the above limitations.

3.1.3 Tiered Client-Server Architecture

By separating interface logic and application logic in the single-layered architecture, we get a tiered architecture, which consists of three tiers: interface logic (presentation), application logic (functionality), and database logic (data). It is illustrated in figure 3.2.

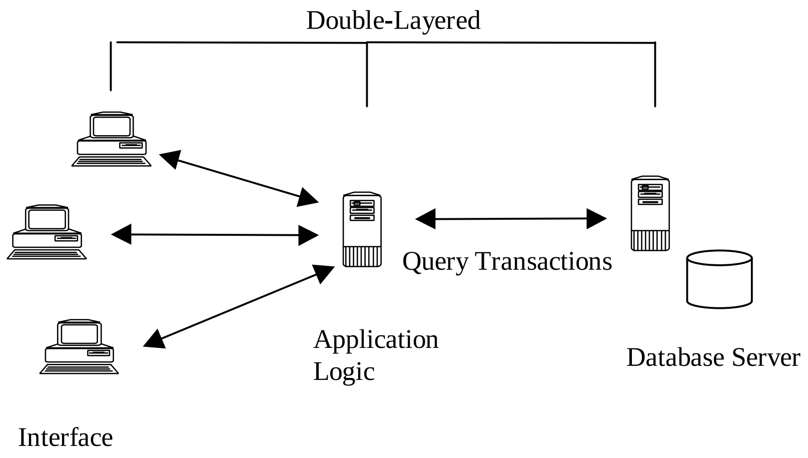


Figure 3.2 Tiered Client-Server Architecture

Well-defined boundaries in this architecture give more flexibility and modularity and improve system cost and performance. It is possible in the architecture to significantly improve the network performance by reducing the traffic caused by query transactions.

For example, in this architecture, the interface interacts with the application logic via standard protocol like socket (TCP/IP) or RPC. The application logic interacts with database server via standard database protocols. The application logic includes mediating requests from multiple clients, translating requests into database queries, passing queries to database server, and translating data from database server into clients' data in return. Significant traffic reduction between functionality module and database server can be expected. In addition, system cost reduction is possible. For instance, given the application logic (server) is running, only interface needs to be invoked per user when multiple users access simultaneously. From the modularity point of view, the separated modules are more manageable in nature. Software maintenance can be applied to each module individually which makes expanding and porting easier. As a result, long term development cost is reduced [10].

3.2 The Use of Java

To meet the requirements of multi-platform and usage portability, we adopted Java as the implementation language for the interface module of the editor. The reasons are described as follows: First, Java is an object-oriented programming language. Its syntax is similar to C and C++ that are "main-line", industry-proven languages. Second, Java has security, network, and machine independent features. It was initially developed to solve the problems of sharing software components in networked environment. It was designed to support multiple heterogeneous host machines. Without any modification, the compiled codes, known as the byte-code, of Java applications can be delivered to and safely run on multiple heterogeneous platforms. Java Virtual Machine, part of Java specification, is responsible for Java's cross-platform delivery, the small size of its compiled code, and Java's ability to protect users from malicious behavior. Third, Java has been integrated with the Word Wide Web (WWW). WWW showed how media-rich content could be accessed by clients on multiple platforms in uniform and simple ways. However, the content supported by the Web's HTML document format was too limited to satisfy the needs of current emerging complicated applications. HTML extensions, such as forms-based Common Gateway Interface (CGI) script, only highlighted those limitations. Embedding Java programs inside HTML pages, known as applets, has been implemented to overcome those limitations. With Java as the extension language, a Web browser is no longer limited to a fixed set of capabilities.

3.3 Exodus Storage Manager

Sentinel applications use Exodus Storage Manager [11] to manage their stored objects. As the Rule Editor and the Sentinel applications must share the same stored rule information for creating the external rules, Exodus is currently used as the storage manager of the rule database. Here we briefly introduce Exodus Storage Manager. Exodus Storage Manager is a multiple-user object storage system supporting versions, indexes, single-site transaction, distributed transactions, concurrency control and recovery. It has a client-server architecture. An application program that uses the storage manager may reside on a machine different from the machine or machines on which the storage server or servers run. The storage manager server is a multi-threaded process providing asynchronous I/O, file, transaction, concurrency control, and recovery services to multiple clients. It stores all data on volumes which are either Unix files or raw disk partitions. It provides objects for storing data, versions of objects for grouping related objects, and indexes for supporting efficient object access.

3.4 The Architecture of the Rule Editor

As we have seen in the earlier discussion, the tiered architecture is better than the commonly used client-server architecture in many ways. In order to apply the architecture to the design of the Rule Editor, the editor is decomposed into several manageable modules. The architecture and the functional modules of the Rule Editor are illustrated in figures 3.4 and 3.5 respectively.

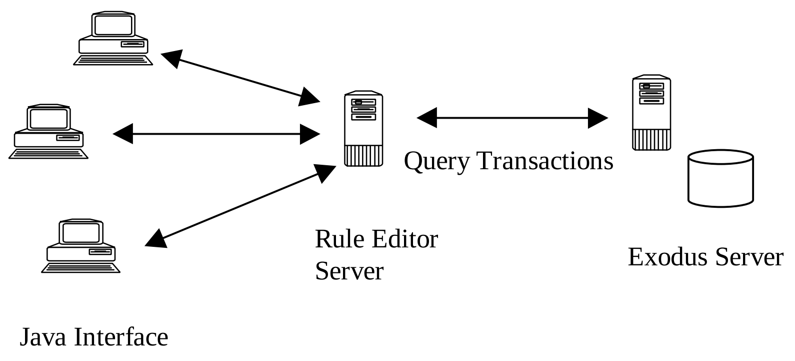


Figure 3.4 The Architecture of the Flexible Dynamic ECA Rule Editor

The functionality of each module is described as follows. The Java interface provides the users with GUI to collect, validate, wrap, and display information. As a client, it interacts with Rule Editor Server via TCP protocol. The Rule Editor Server has most of the application logic. Based on requests from multiple clients, the Rule Editor Server does corresponding jobs such as retrieving information from the rule catalog, generating the DLL, and persisting the rule information by acting as a client to the database server (Exodus server). At run time, based on the rule database and the DLL,

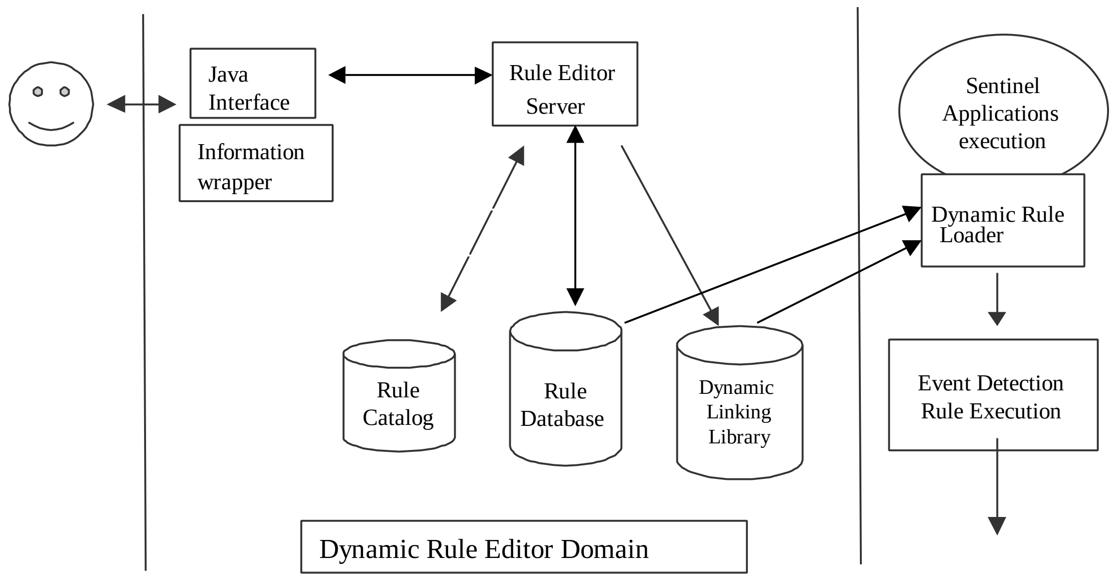


Figure 3.5 The Functional Modules of the Flexible Dynamic ECA Rule Editor

the Dynamic Rule Loader invoked by Sentinel applications creates events and rules by calling the EVENT and RULE class constructors. The event detection logic and the rule execution logic are implemented in these classes, which are discussed in other Sentinel papers [6][7][8][9].

The following chapter discusses the details of these modules.

CHAPTER 4
DESIGN AND IMPLEMENTATION DETAILS

In this chapter, we discuss details of the modules used in the Flexible Dynamic ECA Rule Editor.

4.1 Java Interface

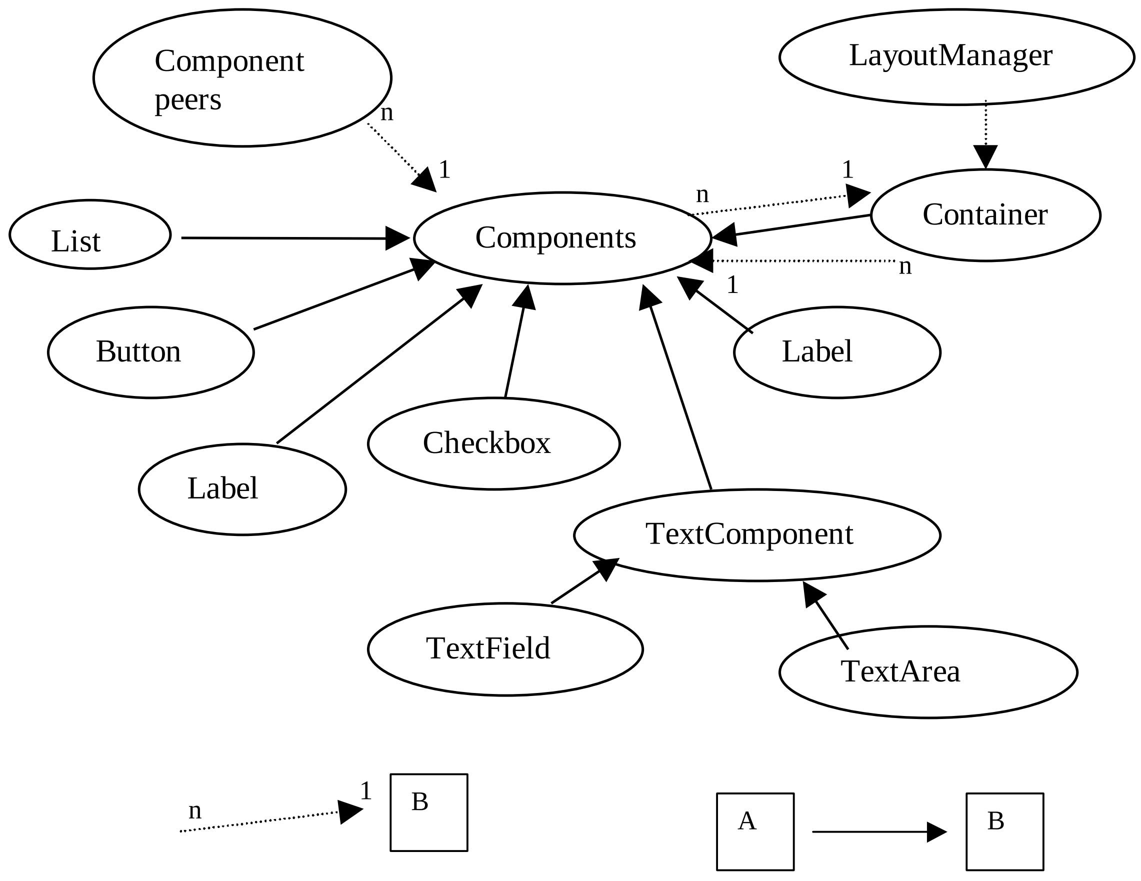
Anyone writing Java applets and Java graphic applications is going to work with the Abstract Window Toolkit, commonly referred to as the AWT, part of the freely distributed Java Developer's Kit (JDK). In the following subsection, we introduce the AWT and its underlying windowing system.

4.1.1 Abstract Window Toolkit and Peers

As the programming interface for applet and graphic development, the AWT provides a generalized set of classes. They are mainly composed of the following four classes: the **Component** class, **Container** class, **Graphics** class, and **LayoutManager** interface. **Component** class is an abstract class for GUI components such as menus, buttons, labels, lists, and so on. **Container** is an abstract class that extends Component. Classes derived from **Container**, most notably **Panel**, **Applet**, **Window**, **Dialog**, and **Frame**, can contain multiple components. **LayoutManager** is an interface that defines methods for positioning and sizing objects within a container. **Graphic** class is an abstract class that defines methods for performing graphic operations in a component. Every component has an associated **Graphic** object.

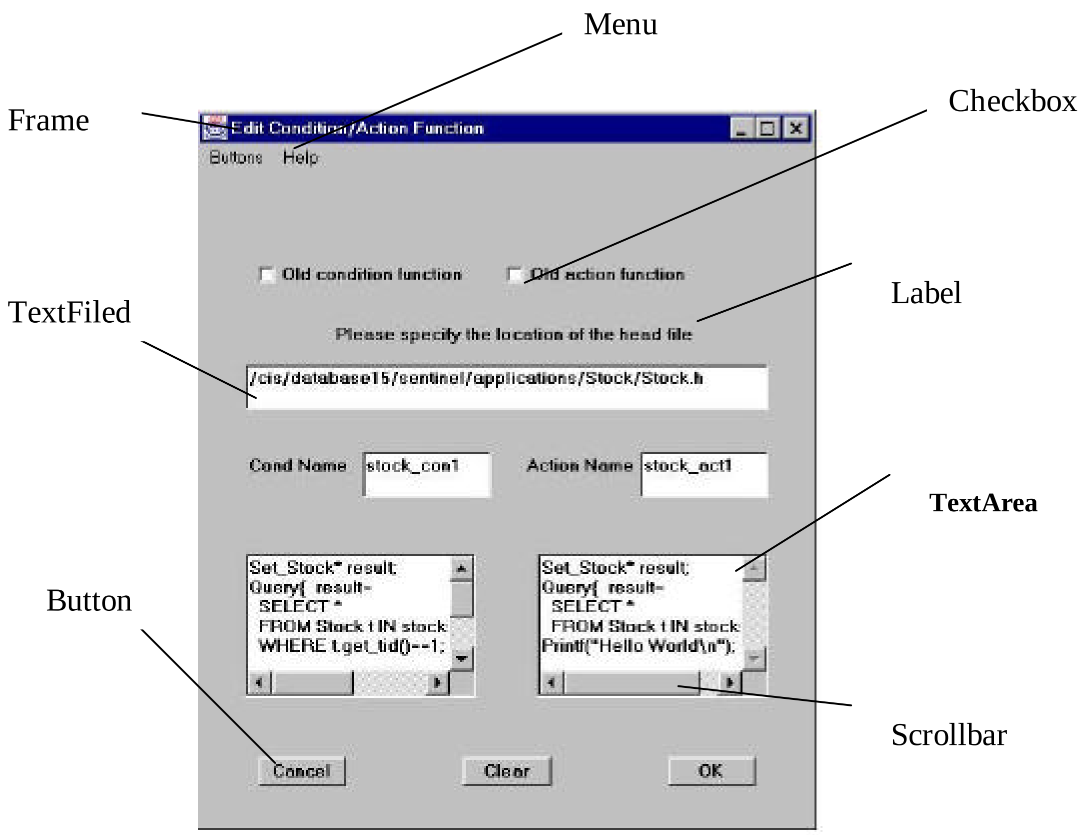
The AWT is a platform-independent windowing toolkit. This feature is made possible by a set of AWT classes known as *peers*. *Peers* are native platform-dependent GUI components that are manipulated by AWT classes. The AWT delegates the actual rendering and behavior of frames, menus, panels, labels, buttons and so on, to the native platform-dependent windowing system.

Figure 4.1 shows a class diagram of the relationship between components, containers, layout managers, and peers. Figure 4.2 illustrates a window sample from the Rule Editor GUI to demonstrate some Java AWT Components.



B has n As. A is inherited form B

Figure 4.1 AWT Component Class Diagram



/cis/database15/sentinel/applications/Stock/Stock.h

Cond Name stock_con1 Action Name stock_act1

```
Set_Stock* result;
Query{ result=
 SELECT *
 FROM Stock t IN stock
 WHERE t.get_tid()==1;
```

```
Set_Stock* result;
Query{ result=
 SELECT *
 FROM Stock t IN stock
Printf("Hello World\n");
```

Figure 4.2 Java AWT Components

4.1.2 Java Applet

Applets are Java programs that execute within a Java-enabled WWW browser. The Java applets are launched by embedding the **APPLET** HTML tag in a web page and then viewing that page with the browser. For example, to execute the Rule Editor through a browser, we have to embed the following code in the web page:

```
<APPLET CODEBASE ="http://www-pub.cise.ufl.edu/~hungju/RuleEditor/Class"
  CODE=RuleEditorApplet.class WIDTH=350 HEIGHT=60>
</APPELT>
```

The HTML tag CODEBASE ensure that the applet (the Rule Editor's interface) is downloaded form that codebase machine. When applets are executed in a Java-enabled browser, the browser provides a great deal of the infrastructure necessary for the applets' execution. For example, when the Rule Editor web page is visited, the browser calls init() and start() methods to initialize and start the display of the editor's interface. When the editor is no longer displayed, the browser calls stop() and destroy() to remove the display and release system resources.

There is one notable security restriction with an applet. It is that all communication must be with the host that delivered the applet to its current host. This implies that the Rule Editor Server must be run on the Web server machine.

4.1.3 Java Application

The Rule Editor is a system tool for Sentinel. It will be unsuitable that it must always be used through a browser (unless we develop our own Java-enabled browser). A standalone Java application is also a desired way to implement the Rule Editor. From the developer's point of view, there are primarily two differences between a Java applet and a Java application:

- An application must include a main() method.
- If the application requires a window, it must extend the AWT **Frame** class.

The other notable distinction between the two kinds of Java programs is that an application dose not have the same security restrictions that an applet dose. For example, whereas an applet cannot write or modify files, an application can perform file I/O. Applications are not constrained by the security restrictions that the Java-enabled browser enforces on applets.

It is desirable to have the editor run in the both modes. There are a number of ways to write a Java program so that it can execute either form within a browser or as a standalone application. One way to do this is to center an Applet within a Frame, as in the following example.

```
        …………
1       public class RuleEditorApplet extends Applet {
        …………….
2           public static void main (String args[]) {
                RuleEditor  app = new RuleEditor ( ….. );
                …………………….
            }
            public void init()   {  ……………  }
            public void start () {   …………    }
            public void stop ()      {   ………….   }
            public void destroy () {   …………   }

3         class RuleEditor extends Frame {
            public RuleEditor (….)
4           RuleEditorApplet applet = new RuleEditorApplet();
            Applet.start();
            ………………
            }
5           public boolean handleEvent(Event event) {
            if (event.id == Event.WINDOW_DESCTROY)
              System.exit(0);
            Retrun false;
          }
        }
```

Putting the main() method (line 2) in the RuleEditorApplet class (line 1) enables the program to be executed with the same argument, whether it is going to run as an applet or as an application. To run it as an applet, we need to embed the code presented in previous subsection in the web page. In this case, the main() method is completely ignored; instead, the browser invokes the init() and start() method to start the applet. Line 3 begins the definition for the RuleEditor class-- the application class. This class provides the Frame for the application to run in. The work starts in line 4, where a new applet is created and centered in the Frame. Line 5 simply sets up a simple event handler for existing the application. It demonstrates the way that a Java program handles the events. The events may result from a mouse click on a graphic component. Figure 4.3 depicts the Rule Editor invoked as a standalone application and figure 4.4 illustrates the use of the Rule Editor through WWW as an applet.

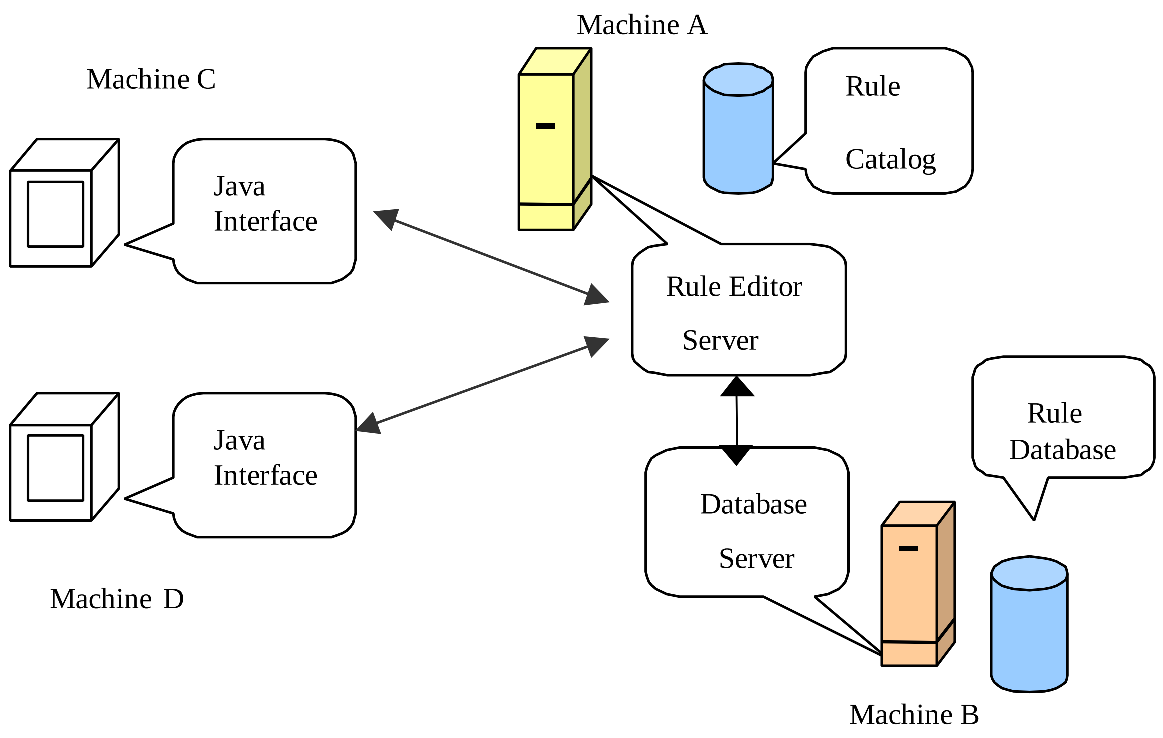


Figure 4.3 The Rule Editor Runs As a Stand Alone Application

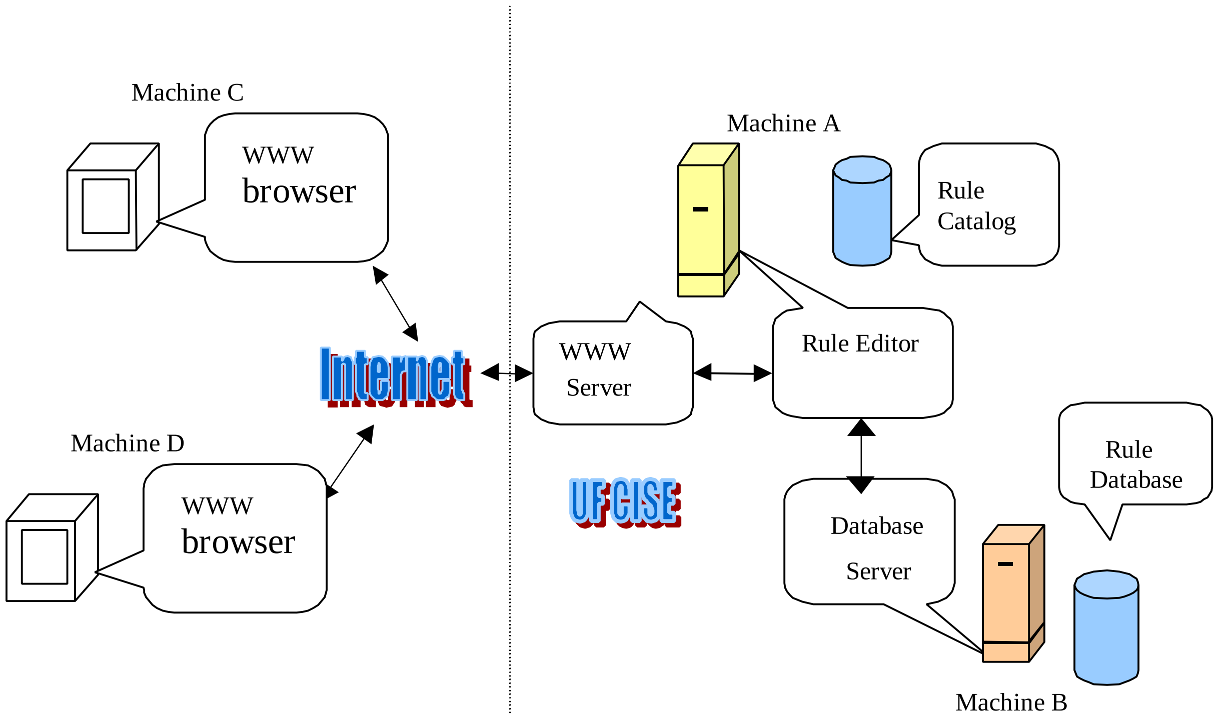


Figure 4.4 The Rule Editor Runs Through WWW As an Applet

4.1.4 Self-guided Interactive GUI

To facilitate the use of the Rule Editor, we adopt the self-guided style to design the interface module. The GUI of the Rule Editor presents the users with a sequence of windows in proper format to guide them through the necessary steps involved in manipulating a rule. The flow chart of the Rule Editor Interface is illustrated in Figure 4.5. The interface details and its back-end are discussed in the following subsection.

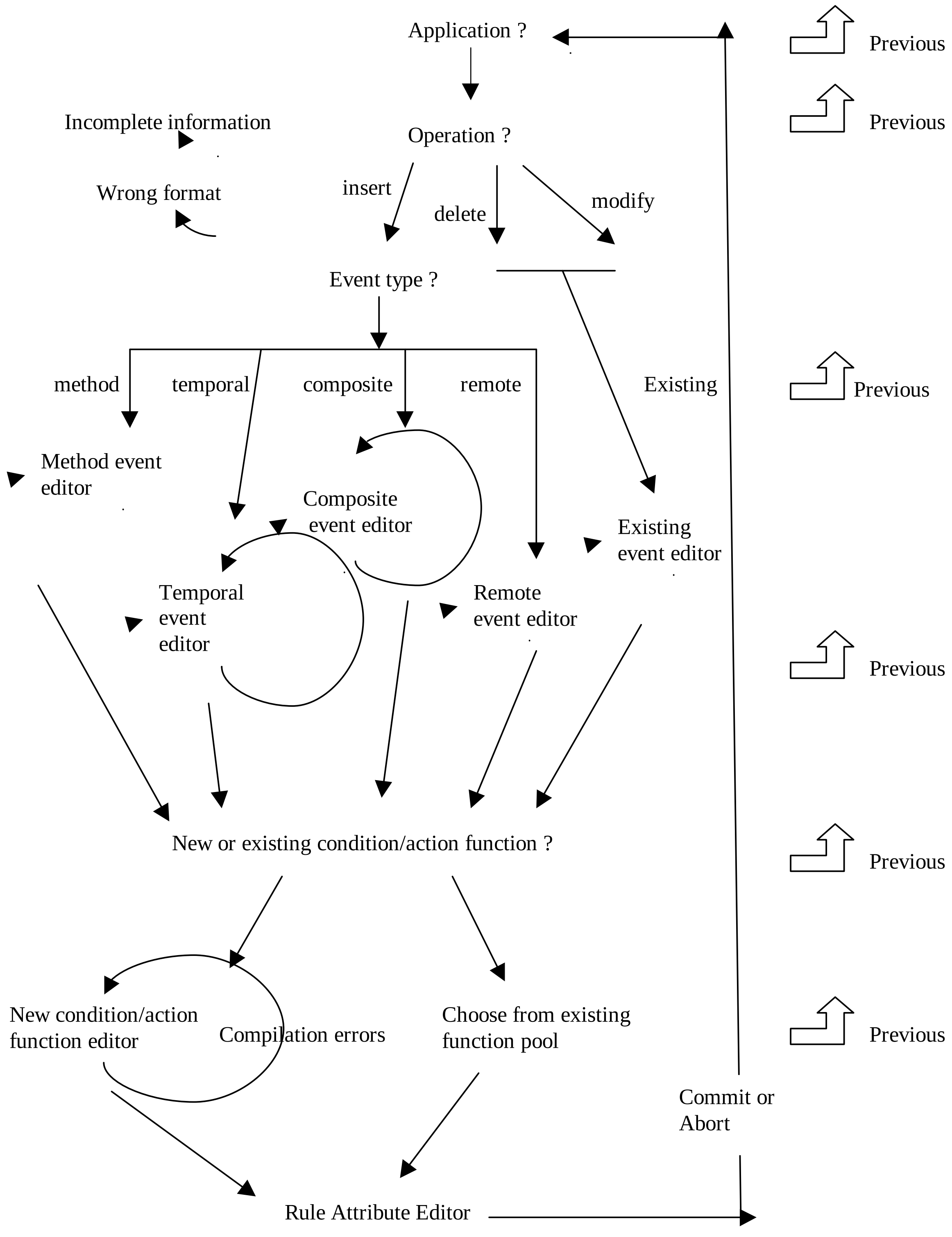


Figure 4.5 The Flow Chart of the Rule Editor Interface

4.1.5 Interface Logic

To make the interface more manageable and expandable, instead of using a complicate window, we assemble a chain of widows from a set of widows for each type of operations. Another reason to do so is to ensure that every window will show up properly on various platforms having different resolutions. Each window is implemented by a Java class inherited from **Frame** class in AWT and is responsible for certain type of necessary information needed to manipulate a rule. Each window and its components will show up, hide, or get disposed based on the users' choices. The interactive process will not go further and a warning window will show up if the collected information is not complete or its format is incorrect. The warning windows are shown in figure B.13 in the appendix. There is a base window, named logo, implemented by **RuleEditor** class that accommodates all global variables where all necessary information for a rule are stored. The logo window is illustrated in Figure B.1 in the appendix. All other windows update the variables based on the collected information. We can consider every operation on a rule as a transaction. It should be done in all or nothing semantics. To ensure the atomicity of an operation, there are three basic buttons, "Previous", "Cancel", and "OK" on every window to control the flow of the interface. Updates on the variables would be undone if "Previous" or "Cancel" is clicked. The content of those variables are sent to Rule Editor Server only when users click "commit" button on the last window of the window chain.

The following are the possible windows that form the window chains (The windows are represented by capital letters in parentheses.).

(A) Choosing applications: The window displays the names of all existing applications. This information is received from Rule Editor Server based on the system file named applications.txt in the system rule catalog directory. The user is asked to select an application for which he wants to work on by double clicking one of the given applications in the list. The window is shown in Figure B.2 in the appendix.

(B) Choosing operations: A user is asked to select one of the three operations, insert, delete, or modify a rule, by checking one of the three radio buttons. The window is illustrated in Figure B.3 in the appendix.

(C) Choosing event types: This window displays all types of events supported by Sentinel. They are classified into four groups, primitive, composite, remote, and existing events. The window is shown in Figure B.4 in the appendix. The primitive events are divided further into method and temporal events. When an event of any type is selected, a corresponding window will pop up to allow the user to edit the selected event. A composite event is constructed from the ten operators and represented by a tree (The semantics of the ten operators were discussed in chapter 2). Internal nodes in the tree represent operators and leaf nodes represent primitive events. To construct a composite event, a user needs to check one of the ten radio buttons that represent the corresponding operators. An operator editor window, say W1, will pop up for editing the corresponding type of composite event, say C1. After editing C1, the user may click "Continue" button in W1 to continue to extend the composite event. In this case, C1 is a subtree of the composite event to edit. System will give C1 an alias name for later reference. The procedure will be repeated until the whole tree that represents the desired composite event is formed. It is finished by clicking "OK" button in the last operator editor window.

(D) Choosing an existing event: This window displays three lists of existing events, which are method, temporal, and composite event lists. The information is received from the Rule Editor Server based on the rule database. A user is asked to select one of them as the event for the rule to work on. The window is illustrated in Figure B.5 in the appendix.

(E) Method event editor: This window displays a list of all methods defined in the active classes of the selected application. The information is received from Rule Editor Server based on the method signature file in the system directory. The file is generated by the Sentinel preprocessor. Every given method is a potential event. A user is asked to select one of them by double clicking on it. He is also asked to check one of the two radio buttons to specify whether the event should be detected at the beginning or at the end of the execution of the method. The window is illustrated in Figure B.6 in the appendix.

(F) Absolute temporal event editor: This window provides an entry to user for editing the time string. A fixed format of the time string is presented to guide users to input a correct one. A warning window will pop up if the given time string is invalid. The window is depicted in Figure B.7 in the appendix.

(G) Remote event editor: This window provides two entries to the user for editing the machine-application-name string and class-event-name string. Fixed formats of the strings are presented to guide users to input the correct formats.

(H) Operator editor windows: This category contains a set of windows using which composite events are edited. Each window corresponds to an operator that has two or three operands (events) depending on its semantics. Each window shows all possible operand (event) lists received from Rule Editor Server and previously edited information. The possible types of the operands for an operator are method events, relative temporal events, and composite events represented by alias names. A user may double click one item in the list to select the corresponding event as an operand to the operator being edited. If the selected events need more information, some irrelevant entries will get hidden and others will show up to facilitate the user to input only the required information. Some of these windows are depicted in Figure B.8 -- B.12 in the appendix.

(I) Using existing condition and action functions: By checking two checkboxes, "Use old condition" and "Use old action", this window displays two lists of functions previously defined in the selected application. One is for condition functions and the other is for action functions. The information is based the system rule catalog managed by the Rule Editor Server. A user is asked to select one condition function and one action function by double clicking on it.

(J) Editing new condition and new action functions: This window provides two editing areas and three entries to the user for editing condition/action functions, entering condition/action names and specifying the location of the header file associated with the functions. The condition function is of type boolean and the action function is of type void. A user may edit C++ code and OQL in the editing area whenever they make sense. The window is illustrated in Figure B.15 in the appendix.

(K) Using an existing condition function and editing a new action function. The window is shown in Figure B.14 in the appendix.

(L) Editing a new condition function and using an existing action function.

(M) Editing the attributes of a rule: This window provides one entry and four lists to the user for entering a rule name and specifying the attributes of the rule. These attributes include CONTEXT, PRIORITY, TRIGGER modes, and COUPLING modes. The options available are the following:

- CONTEXT: It can be one of these options: RECENT, CHRONICLE, CONTINUOUS, and CUMULATIVE.
- PRIORITY: It can be any integer value from 1 to 50.
- TRIGGER: It can be either NOW or PREVIOUS.
- COUPLING: It can be one of these options: IMMEDIATE, DETACHED, or DEFERRED.

(The semantics of these options were discussed in Chapter 2.)

This window is the last window of any window chain. It provides Cancel and Save buttons for the user to quit or finish the operation (transaction). The window is illustrated in Figure B.16 in the appendix.

(N) Selecting an existing rule: This window displays a list of rule names associated with the event selected in the previous window. The information is based on the rule database via the Rule Editor Server. The user is asked to select one of them by double clicking on it. Up to this point, the user either deletes or modifies a rule. In case of deleting a rule, the window (O) will show up and the user will be asked to save or cancel. In case of modifying a rule, the content of condition/action functions associated with the rule will show up in the window (J).

(O) Showing all information associated with the selected rule. The information includes all components of the event and the contents of the condition/action functions.

A window chain is established by the following three information: the type of the operation (insert, delete, or modify), the event type, and the category of the condition/action functions (existing or new). The following are possible combinations of the first two information and their corresponding window chain (The window chains are represented by integer numbers in parentheses.):

(1) Inserting a rule with a new method event:

(B) → (C) → (E)

(2) Inserting a rule with a new temporal event:

(B) → (C) → (F)

(3) Inserting a rule with a new remote event:

(B) → (C) → (G)

(4) Inserting a rule with a new composite event:

(B) → (C) ←→ (H)

(5) Inserting a rule with an existing event:

(B) → (C) → (D)

(6) Modifying an existing rule:

(B) → (D) → (N)

(7) Deleting an existing rule:

(B) → (D) →(N) → (O)

The following are the possible options for the third information and their corresponding window chain:

(8) With an existing condition and an existing action function:

(I) → (M)

(9) With an existing condition function and a new action function:

(K) → (M)

(10) With a new condition function and an existing action function:

(L) → (M)

(11) With a new condition function and a new action function:

(J) → (M)

The window chains associated with all operations supported by the Rule Editor are illustrated in Figure 4.6.

4.2 Information Wrapper

Information wrapper is a component of the Editor Interface that translates the information collected from the user to a format that is used by the RE Server. The purpose of this translation is to ensure three things: (i) the information is correctly transported to the server, (ii) the server easily persists the received information in terms of the database classes, and (iii) the information stored in the database is successfully used in creating rules by the Dynamic Rule Loader at run time.

$$\begin{tikzcd}
 & & (8) \arrow[llddd, bend right] \\
 & & (9) \arrow[lllddd, bend right, to=A] \\
(A) \arrow[r] \arrow[dd] & \{(1),(2),(3),(4),(5),(6)\} \arrow[ruu] \arrow[ru] \arrow[r] \arrow[rd] & (10) \arrow[ll, bend left] \\
 & & (11) \arrow[llu, bend left] \\
 & (7) \arrow[luu] &
\end{tikzcd}$$

Figure 4.6 Possible Window Chains Supported by the Rule Editor

To avoid incompatible problems between the client and the server, which are written in different languages, all information sent to the Rule Editor Server is represented as strings. The following two sections introduce the two main portions of the information to be processed. One is about the string representation of the condition/action functions. The functions are wrapped so that they can be successfully compiled to generate a dynamic linking library at the server side without the C++ compiler mangling their names which is necessary for creating a rule with its condition and action functions. The other one is about the string representation of a composite event.

4.2.1 C++ Name Mangling and Condition/Action Functions

As we know, C++ supports name overloading. Most C++ compilers generate a unique internal name for each distinct function. They usually encode the function argument types in a suffix appended to the source name of the function. Condition and action functions are implemented in C++, OQL, and Snoop. They will be sent to Rule Editor Server and processed by Sentinel preprocessor and C++ compiler. For dynamic linking, the Rule Editor needs the internal function names to get the function pointers from dynamic linking library for constructing a rule. There are two ways to get the internal names. One is to know how the C++ compiler "mangles" the source function names. This approach has the following disadvantages: (1) The mangled name is difficult to understand and not portable. (2) Programmers need to know more about the compiler internals, which may not be practical. One way to solve this problem is to prevent the C++ compiler from mangling the function names by tagging the functions with directives that belong to a "foreign" language. For example, we can tell the C++ compiler the functions are C functions by preceding the key word " extern "C" ".

The Rule Editor Interface prepares two template functions for condition and action functions to ensure that they are in right format for compilation. The skeleton entries are as follows:

- Condition function:

```
#include <Sentinel.h>
#include <?>
extern "C" {
int ?(L_OF_L_LIST* dummy) {
? }
}
```

- Action function:

```
#include <Sentinel.h>
#include <?>
extern "C" {
void ?(L_OF_L_LIST* dummy) {
? }
```

}

The question marks (?) will be replaced with the information given by the users. The first question mark will be replaced by the location of the header file where the schema of OQL in the function body is defined. The second question mark will be replaced by the function name given by the user. The question marks in the bodies of the functions will be filled by the actual code given by the user. The completed information will be sent to the Rule Editor Server as a string preceded by some message identifier and ended by '^' for compilation. The result of the compilation will be sent back in the following format:

- Success: "Object file already exists" (if using an existing function) or
  " Object file has been created successfully." (if editing a new function)
- failure: "Errors in compilation".
  (Actual error messages form compilers……….)

By comparing the first word of the result with "Object" and "Errors", the client (interface) takes the appropriate actions, like displaying error messages, asking users to reedit, or continue to the next step.

4.2.2 The String Representation of a Composite Event

A composite event can be represented by a tree in terms of the classes defined in the rule database schema. A leaf node in the tree stores all the necessary information for creating the corresponding primitive event and an internal node keeps its corresponding event operator type and the pointers to its child nodes. To avoid the incompatibility between the client and the server processes, the event tree is represented by a sequence of strings at the client side. To facilitate the reconstruction of the event tree on the server side, the client (the Rule Editor Interface) assigns every string, which represents a node in the tree, an integer as an identifier. These integer identifiers are sequential and unique. They represent the node pointers in the tree. In a string, information components are separated by a ";". The root node of the event tree is represented by the last one of the sequential strings. For example, the following is the string representation of a composite event, rooted at a periodic-star event, denoted by P*(E1,E2,E3):

- E1: "##PRIMITIVE-METHOD-EVENT##;1;STOCK;begin;int buy_stock(int qty)\n"
- E2: "##TEMPORAL-EVENT##;2;11 months + 11 days + 11 hrs + 11 mins + 11 secs\n"
- E3: "##TEMPORAL-EVENT##;3;11:11:11/11/11/11\n"
- P*: "##PERIODIC*-EVENT##;4;1;2;3\n"

Section 4.4.3 discusses how the Rule Editor Server uses above format to reconstruct the composite event tree.

4.3 Rule Editor Server

The Rule Editor Server is a C++ process running on Solaris operating system. The jobs performed by the Rule Editor Server are summarized as follows:

- Initializing the rule database
- Providing clients with the rule catalog and the rule database information
- Compiling the condition/action functions, reporting results of the compilation
- Making the object files of the condition/action functions into a dynamic linking library
- Reconstructing the event trees and persisting the event and rule information
- Moving the source code files of the condition/action functions to the system directory for later reference
- Cleaning up the temporary directory

When Rule Editor Server is invoked, the first thing it will perform is to initialize the rule database. If the rule database does not exist, the Rule Editor Server creates one and initializes it based on the system file named "applications.txt" in the rule catalog directory. The file records all applications' alias names and their corresponding Exodus storage group numbers. On the other hand, if the rule database already exists, the Rule Editor Server updates it based on the system file, deleting out-of-date applications and adding new applications. After initializing the database, it listens to clients in a loop. Whenever a client initiates a connection to it, it forks a child process to serve the client. The following section discusses the message-driven services provided by the server.

4.3.1 Message-driven Services

Rule editor server recognizes the given Exodus storage group number and message identifier and performs the corresponding actions. The possible messages are summarized as follows:

- Messages → Actions
- "give_me_app_inf\n" → Send all applications' alias names and Exodus sorage group numbers to the requesting client. The information is based on the system file "applications.txt".
- "give_me_event_list\n" → Send the name list of the given application's existing events to the requesting client. The information is based on the rule database.

- "@xsg\n"[1] → Send the given application's method signatures, which are potential method events, to the requesting client. The information is based on the application signature file in the rule catalog system directory.
- "$xsg\n" → Send the name list of the given application's existing condition functions to the requesting client. The information is based on the application's rule catalog.
- "~xsg\n" → Send the name list of the given application's existing action functions to the requesting client. The information is based on the application's rule catalog.
- "compile_functions\n" → Compile the given functions in the temporary directory of the given application and send the results of the compilations to the requesting client.
- "commit\n" → Add the given condition/action functions to dynamic linking library (if they are new), move the source files of the functions to function pools belonging to the application (if they are new), and persist the given rule information in the rule database. To facilitate porting, all database dependent code of the Rule Editor Server go through OQL. They are implemented as functions with names preceded by the key word "DRE_". Section 4.4.4 will discuss some OQL statements that manipulate the rule database. The following are the commit type identifiers recognized by the Rule Editor Server.

  Type 1 : "##INSERT##NEW_METHOD_EVENT##\n"
  Type 2 : "##INSERT##NEW_TEMPORAL_EVENT##\n"

[1] The "xsg" means the Exodus storage group number of the application that the requesting client is working on. For example, 8604.

Type 3 : "##INSERT##NEW_COMPOSITE_EVENT##\n"
Type 4 : "##INSERT##NEW_REMOTE_EVENT##\n"
Type 5 : "##INSERT##EXISTING_METHOD_EVENT##\n"
Type 6 : "##INSERT##EXISTING_TEMPORAL_EVENT##\n"
Type 7 : "##INSERT##EXISTING_COMPOSITE_EVENT##\n"
Type 8 : "##INSERT##EXISTING_REMOTE_EVENT##\n"
Type 9 : "##MODIFY##TEMPORAL##\n"
Type 10 : "##MODIFY##COMPOSITE##\n"
Type 11 : "##MODIFY##METHOD##\n"

- “commit_delete\n” →Remove the given condition/action functions from the dynamic linking library space (if they are not shared), remove the source files of the functions to function pools belonging to the application (if they are not shared), and remove the given rule information from the rule database based on the given type identifier. The commit type identifiers recognized by the Rule Editor Server are summarized as follows

Type 1 : "##DELETE##METHOD_EVENT##\n"
Type 2 : "##DELETE##TEMPORAL_EVENT##\n"
Type 3 : "##DELETE##COMPOSITE_EVENT##\n"
Type 4 : "##DELETE##REMOTE_EVENT##\n"

- “abort\n” → clean up the given functions (both source files and object files) in the temporary directory of the given application.

4.4 Rule Database

The rule database is stored in and managed by Exodus Storage System [11]. The schema of the database consists of several classes that contain all information for creating the events and the rules for all applications. This section introduces the structure of the database, some OQL statements that manipulate the database, and how the Rule Editor Server transforms the strings received from the client into the event tree in terms of the classes defined in the schema.

4.4.1 The Structure of the Rule Database

The database contains a set of applications. Each application has four sets of events. They are composite events, denoted by C_events, method events, denoted by M_events, temporal events, denoted by T_events and remote events, denoted by R_events. Every event in the four sets has a set of rules associated with it. Figure 4.4 illustrates the structure of the database.

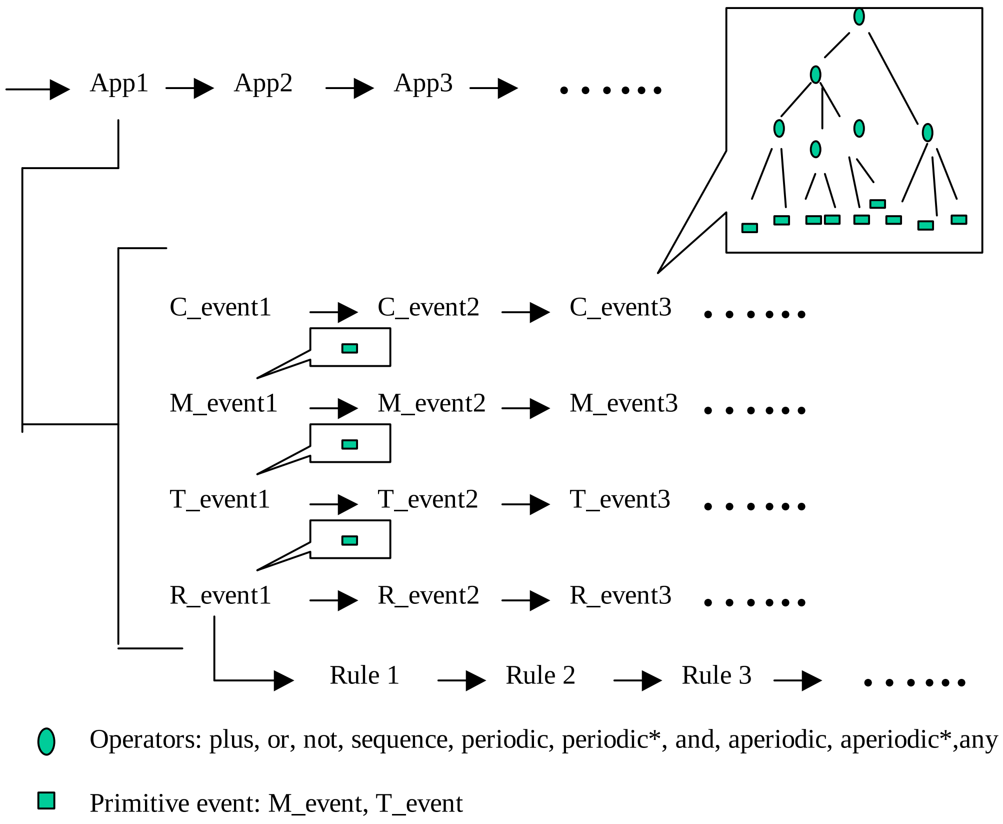


Figure 4.7 The Structure of the Rule Database

The following subsection discusses the outline of the classes by which the structure is implemented.

4.4.2 Rule Database Schema

The following are the interface of the classes defined in the database schema:

```
class AppNode {
private:
  Set_EventNode comp_events;
  Set_EventNode method_events;
  Set_EventNode temporal_events;
..Set_EventNode remote_events;
  char* app_name;
  int xsg;
public:
……………
  int insert_rule(char* , char*, RuleNode*, EventNode*);
  int delete_rule(char* , char*,char *);
  int modify_rule(char* , char*, char*, RuleNode*);
  void create_events_rules(void*);
……………
};

DECLARE Set<AppNode>

class EventNode {
private:
  char*  event_name;
  char*  method_signature;
  char*  begin_end;
  char*  class_name;
  char*  event_type;
  char*  machine_app; //for remote event
  char*  className_eventName; //for remote event
  char* comp_event_info;
  EventNode *child_event1, *child_event2, *child_event3;
  Set_RuleNode rules;
public:
……………….
  int insert_rule(RuleNode*);
  int modify_rule(char*,RuleNode*);
  int delete_rule(char*);
..EVENT* create_event(void*);
……………….
};

DECLARE Set<EventNode>

class RuleNode {
private:
  char*  rule_name;
  CONTEXT  context;
  Char*  coupling_mode;
  Char*  trigger_mode;
  Char*  priority;
  Char*  condition_name;
  Char*  action_name;
public:
…………..……
..void  create_rule(EVENT*,void*);
…………….
};

DECLARE Set<RuleNode>
```

Note: The **DECLARE** is a macro facility supported by OpenOODB from Texas Instruments. The macro will be expanded by its preprocessor named CoolCpp to a parameterized collection class [12]. For example: DECLARE Set<AppNode> will be expanded to a class named **Set_AppNode**. Appendix A.3 will discuss more details about the parameterized collections.

The classes **AppNode, EventNode**, and **RuleNode** are used to internally store and represent the relevant information of an application, an event, and a rule respectively. The information will be used by the Dynamic Rule Loader for creating the events and the external rules at run time. The rule database is an instance of **Set_AppNode** class. Every member of the set is an instance of the **AppNode** class. The **AppNode** class has four data members of type **Set_EventNode**. Every member among the four sets of events belonging to the application is an instance of **EventNode** class. The **EventNode** class has three pointers to its child event nodes, all necessary information needed to create an event, and a data member of type **Set_RuleNode**. If an instance of **EventNode** is M_event, T_event, or R_event, its child event pointers point to NULL. If an instance of **EventNode** is C_event, its child event pointers point to other event subtrees that represent another C_events or other type of events. Every set of rules belonging to an event is an instance of **Set_RuleNode** class. Every member in the set is an instance of **RuleNode** class that stores all necessary information needed to create a rule.

4.4.3 The Reconstruction of an Event Tree

When the Rule Editor Server receives the sequence of the stings that represent a composite event, it uses the strings to construct an event tree in terms of the classes defined in the schema. The tree internally stores all information needed for creating the composite event at run time. The way that the server reconstructs the tree is described as follows: It creates an object of **EventNode** for every string based on the information carried in the string. It declares an array of **EventNode** and assigns those objects to the array using the integer ID of each string as the index. The last element of the array is the root of the tree. Figure 4.5 illustrates the reconstruction of the event tree using the example in section 4.3. The following section introduces how the event tree and all rule information are persisted by using OQL.

4.4.4 Persisting Information Using OQL

The database dependent code in the Rule Editor Server is implemented by OQL. The following are two examples of OQL for manipulating the rule database:

```
Query{
UPDATE (app.insert_rule(event_group,event_name,rule_ptr,event_ptr))
FROM AppNode app IN applications
WHERE app.get_xsg()==xsg  ;}
```

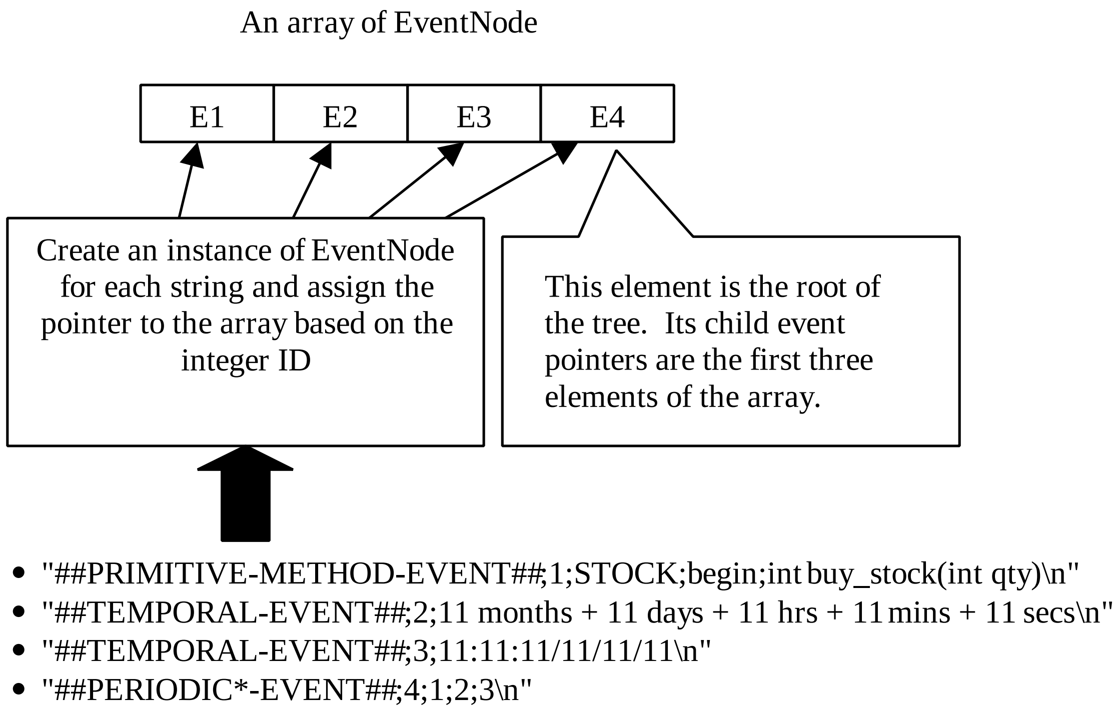

An array of EventNode
E1
E2
E3
E4
Create an instance of EventNode for each string and assign the pointer to the array based on the integer ID
This element is the root of the tree. Its child event pointers are the first three elements of the array.
"##PRIMITIVE-METHOD-EVENT##;1;STOCK;begin;int buy_stock(int qty)\n"
"##TEMPORAL-EVENT##;2;11 months + 11 days + 11 hrs + 11 mins + 11 secs\n"
"##TEMPORAL-EVENT##;3;11:11:11/11/11/11\n"
"##PERIODIC*-EVENT##;4;1;2;3\n"

Figure 4.8 An Example of the Event Tree Reconstruction

This query inserts a rule into an application whose Exodus storage group is "xsg". The "applications" is the alias name of the application set in the database. Rule editor server fetches the set from Exodus server based on the name. The "app" is an instance of the **AppNode** and a range variable over the application set. "insert_rule" is a member function of **AppNode**. It takes four parameters: "rule_ptr" of type pointer to **RuleNode**, "event_ptr" of type pointer to **EventNode**, "event_group" of type string, and "event_name" of type string. If "event_ptr" is not NULL , the event is a new event. "event_ptr" is the root of the event tree, which has been discussed in the previous section. Here is a piece of code from the implementation body of **AppNode**'s "insert_rule" function:

```
//Insert the new rule to the rule set of the new event.
event_ptr->insert_rule(rule_ptr);
//The event is a C_evnet.
if (!strcmp(event_type,"COMPOSITE")){
….
//Insert the event to the composite event set of the given application.
if(!comp_events.Insert_Member(*event_ptr)) {
…..}
}
```

The first line inserts the new rule pointed by the rule_ptr to the rule set belonging to the new event pointed by the event_ptr. The rest of the code inserts the event pointed by the event_ptr to one of the four sets belonging to the given application based on the "event_type". The comp_events is a data member of **AppNode**. On the other hand, If the event_ptr is NULL, the rule to be inserted uses an existing event. The following query is to find the event node among comp_events based on the "event_name"

```
Query{
      result = select *  from EventNode* event in :comp_events
      where !strcmp(event->get_event_name() , event_name)
;}
```

Note: The ':' in ":comp_events" means the set of objects are already in application address space.

4.5 Dynamic Rule Loader

At run time, when a Sentinel application is invoked, it executes "load_dyn_rules" function. In the function, it performs the following two steps:

- It retrieves the rule database information of the given application (Exodus group number) from Exodus server. The following is a piece of code used in this step:

  ```
  Set_AppNode* applications_ptr;
  Query{
  Applications_ptr =
  select *  from AppNode* app in applications
  where app->get_xsg() == xsg;}
  ```

- It traverses all data structure and creates events and their associated rules, based on the information stored in the structure, by calling the EVENT and RULE class constructors. A handle to the DLL is needed by the Dynamic Rule Loader for retrieving the condition/action function pointers which are two of the parameters of the RULE class constructor. The following is the code that gets the handle.

  ```
  void*  dyn_lib_handle= dlopen(dyn_lib_location , RTLD_LAZY);
  ```

The following is the code that calls " create_events_rules(dyn_lib_handle)" method for every instance of AppNode among the application set retrieved in step 1. Actually, the application set contains only one member whose Exodus group number is equal to "xsg".

```
for(applications_ptr->First_Member(i); !applications_ptr->Is_End_Of_Set(i);
    applications_ptr->Next_Member(i)){
    (applications_ptr->Get_Member(i)).create_events_rules(dyn_lib_handle);}
```

In the implementation body of " create_events_rules(dyn_lib_handle)", it traverses the four event sets belonging to the application. The following is the code that traverses the composite event set and calls "create_event" method for every member in the set.

```
for(comp_events.First_Member(i);!comp_events.Is_End_Of_Set(i);comp_events.N
ext_Member(i)){
      (comp_events.Get_Member(i)).create_event(dyn_lib_handle ); }
```

The implementation body of " create_events(dyn_lib_handle)" creates a certain type of event and all the rules based on the event by calling the corresponding event constructor and the rule constructor. For example, the following is the code that creates a primitive event and its associated rules.

```
if(!strcmp(event_type,"PRIMITIVE")){
PRIMITIVE *primitive_event= new
PRIMITIVE(event_name,class_name, begin_end,method_signature);
for(rules.First_Member(i);!rules.Is_End_Of_Set(i);rules.Next_Member(i)){
```

```
       (rules.Get_Member(i)).create_rule(primitive_event,dyn_lib_handle);
}
return primitive_event;
```

The following is another example that creates an AND composite event and all rules based on the event.

```
if(!strcmp(event_type,"AND")){
AND *and_event=NULL;
…...
 and_event = new AND(event_name,child_event1->create_event(dyn_lib_handle),
child_event3->create_event(dyn_lib_handle ));
….
 for(rules.First_Member(i);!rules.Is_End_Of_Set(i);rules.Next_Member(i)){
       (rules.Get_Member(i)).create_rule(and_event,dyn_lib_handle );        }
return and_event;
```

The following is the code from in the implementation body of "create_rule" method present in the **RuleNode** class.

```
int (*fptr_cond)(L_OF_L_LIST*)=NULL;
void (*fptr_act)(L_OF_L_LIST*)=NULL;
fptr_cond=(int(*)(L_OF_L_LIST*))dlsym(dyn_lib_handle,condition_name);
fptr_act=(void(*)(L_OF_L_LIST*))dlsym(dyn_lib_handle,action_name);
…..
RULE *rule_ptr = new RULE(rule_name, event_ptr, fptr_cond,fptr_act , context);
….
```

It creates a rule by calling RULE constructor with some parameters. The "rule_name" and "context" are the data members of the **RuleNode**. The "event_ptr" comes form the caller. It is the return value of the event constructor in the caller's code. The condition and action function pointers, "fptr_cond" and "fptr_act", are got by calling dlsym utility with two parameters. The dynamic linking library handle is passed as an argument by the caller. The "condition_name" and "action_name" are the data members of the **RuleNode** class. The functions pointed by "fptr_cond" and "fptr_act" will be executed once the event pointed by "event_ptr" is detected. The event detection logic and rule execution logic are beyond the scope of this thesis. They are discussed in other Sentinel papers [6][7][8][9].

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

This thesis proposes a flexible design and implementation of a Rule Editor to extend Sentinel system to support external rules. We have seen how the three-tiered architecture being better than the traditional two-tiered approach in many ways. To apply the architecture to our design, the editor is basically decomposed into four modules: the interface, the Rule Editor Server, the rule database, and the Dynamic Rule Loader. This well-defined boundaries give more flexibility and modularity to the system that provides more room for improving performance and maintenance. We use Java to implement the interface module, which allows the editor to run either from within a WWW browser or as a standalone application. To facilitate porting, we decoupled database logic from the Rule Editor Server by using OQL and separating all the database dependent code as individual functions. In the Dynamic Rule Loader module, we used dynamic linking library to invoke condition and action functions, which avoids the relinking of applications.

5.2 Limitation and Future Work

The editor proposed in this thesis only supports the Sentinel system based on OpenOODB. The design of the editor is only to make porting easier rather than to support other database management systems. In the future, we plan to incorporate CORBA to extend the editor to support external rules under multiple heterogeneous database management systems.

APPENDIX A
OVERVIEW OF OQL FOR SENTINEL

OQL for Sentinel is an extended version of OQL[C++] implementation that accompanied the OpenOODB toolkit from Texas Instruments. They are introduced in section A.2 and section A.1 respectively. In section A.3, we present an OQL application, hospital database, to demonstrate the usage of OQL and its preprocessing. Section A.4 introduces more OQL examples based on the application. The limitations of OQL are discussed in section A.5. To facilitate the understanding of OQL, some OQL demo programs have been designed and implemented. They are discussed in the last section.

A.1 Introduction to OQL[C++]

OQL[C++] is an object query language that applies to OpenOODB. It is an example of the binding of a query language with C++. It extends C++ with query capability using well known SELECT-FROM-WHERE structure of SQL block. The following section introduces the format of OQL[C++].

A.1.1 The Format of OQL[C++]

A OQL[C++] query statement block [12] is represented as follows:

```
SELECT <objects>
FROM <range variable> IN <set>
WHERE <predicate>
```

The SELECT clause identifies the type of the objects in a set to be returned by the query. The FROM clause declares range variables as well as the target set to be queried. The

WHERE clause specifies the predicate that defines the properties to be satisfied by the objects to be retrieved. The following is a simple example:

- Retrieve the patients whose family doctor is Mary.

```
SELECT p
FROM Patient p IN patients
WHERE p.family_dr( ).name( ) == "Mary";
```

Patient is a class name defined in a C++ program. The SELECT clause states that the objects returned by the query belong to the class Patient. In other words, the objects in the result set of the query are Patient objects that behave exactly as Patient objects defined in a C++ program. p is a range variable over the instances of the patients set. "patients" is an alias name associated with a user-defined set object of type Set_Patient which contains persistent objects of type Patient. The family_dr() is the class Patient's public member function of type Physician class. The name() is the public member function of Person class inherited by class Patient and class Physician. The expression p.get->physician ( ).name( ) is called a path expression.[1] The dot "." notation means the invoking of a member function, which follows the C++ convention.

The path expression allows navigation through the object composition which enables the formulation of predicates on values deeply nested in the structure of an object. It provides a uniform mechanism for the formulation of a query that involves object composition and inherited member function. The predicates in the WHERE clause can be defined using comparison operators Θ, which belongs to one of {==,<,<=,>,>=,!=}, and logical operators &&(AND), || (OR), and ! (NOT). Let p.M1(A1).M2(A2). ..... .Mn(An) be a path expression. Every Mi(Ai), where 1 <= i <= n,

[1] The paragraph describing the definition of path expression is quoted from [13].

must be single-valued. If a function Mk(Ak) is a set-valued, then the programmer must break the path expression by defining a subquery on the set p.M1(A1).M2(A2). ... .Mk(Ak) [13]. For example,

```
Query{
   Result = SELECT p
        FROM Patient p IN Patients
        WHERE p.get_name() == "Bob" &&
        EXISTS ( SELECT r
                    FROM  Medical_record r IN :p.get_medical_records()
                    WHERE r.get_date() < "10/10/89" && r.get_diagnosis() == "flu" );
   }
```

In " Medical_record r IN :p.get_medical_records( )"[2] the path expression p.get_medical_records( ) is set-valued (i.e., it returns a set of medical records), therefore, it is necessary to define a variable r to range over the member of this set. Note that no direct comparison of private values of an object is allowed. The syntax of comparison and logical operators is the same as C++. Atomic term are t1 Θ t2, t1 IN s1, s1 CONTAINS s2, t1 Θ ALL s1, t1 Θ ANY s1, and EXIST s1.[3] The t1 and t2 are single-valued path expressions or constants, s1 and s2 are sets or set-valued path expressions. Θ is the comparison operator. A predicate is a Boolean combination of atomic terms.

A.1.2 The Features of OQL[C++]

Because of the special binding of OQL[C++] with C++, OQL has the following features:

1. It uses C++ type system as an object data model instead of inventing a new one. This feature eliminates unnecessary data structure mappings between C++ and the database, facilitates type checking. In traditional DBMSs, database applications

are written using embedded query statements in a general purpose language (PL). The embedding usually has limitations in matching between database data model and PL type system. For example, PL dose not support a set type, so additional mechanism like cursor is needed to retrieve a set of data. Moreover, the lack of type checking span both PL codes and query codes and hence increases the opportunity for inconsistencies to be introduced in database [13].

2. It adopts the combination of static and dynamic type checking. Type checking is necessary in database application to ensure that the object transferred between the application space and the database space are consistent. There are two kinds of type checking: static (at compile-time), dynamic (at run-time). OQL adopts a combination of static and dynamic type checking: Its applications statically type checked by a C++ compiler. Run-time type checking is also used in OQL to ensure that data transferred between persistent object base and the application's address space are type-correct [13].
3. It adds a set type to C++ using C++ template. OQL[C++] supports explicit collection type that are defined using C++'s parameterized classes.
4. It allows C++ code in the formulation of queries. For example, it allows user-defined set_valued functions as targets for queries. It allows user-defined boolean_valued functions as query predicates. It allows constructor function calls to create objects of a new type
5. The binding of OQL and C++ allows computational completeness.

[2] The ":" in ":p.get_medical_records( )" is added by Sentinel to differentiate persistent objects and transient objects.
[3] Only t1 Θ t2, t1 IN s1, and EXISTS s1 are implemented so far.

6. It inherits all good features of object-oriented data model like complex objects, object identity, inheritance, aggregation, data abstraction, and polymorphism.
   - Complex objects, Object identity, Inheritance, and Aggregation: Like other object-oriented language, C++ supports inheritance and composition (aggregation) relation among classes to represent complex objects. The feature of object identity allows complex objects to be supported and allows subobjects to be shared. OQL inherits those benefits.
   - Data abstraction: All predicates in OQL[C++] queries are formulated in terms of the objects' public interface. Data abstraction allows OQL[C++] queries on composite objects using a uniform mechanism (i.e., path expressions) without regard to the way composition is implemented.
   - Polymorphism of queries: OQL should be able to operate on semantically different collection types (e.g., set, bag, array). Furthermore, queries should be allowed on any type of collections, including heterogeneous, transient, and persistent collections.[1]

A.1.3 OQL[C++] Preprocessor

The OQL[C++] preprocessor architecture mainly consists of two modules: parser and code generator. An OQL statement is recognized by the preprocessor that parses and translates them into C ++ code with appropriate storage manager function calls, such as "fetch" and "persist" [11]. The resulting code is then compiled by a standard C++ compiler. The implementation details of the preprocessor is discussed in [14].

[1] Only the set type of homogeneous objects is implemented so far.

A.2 Introduction to OQL for Sentinel

OQL for Sentinel is an extended version of OQL[C++]. As the original version of OQL[C++] was not complete, its preprocessor had been modified before it was integrated into the Sentinel preprocessor. We found and fixed several bugs spanning over JOIN, SUBQUERY, UPDATE, DELTET, IN, and WHERE clauses. Most of them were due to wrong C++ code translation. Besides, we also extended two of its limitations. One was INSERT clause, which allowed new objects to be inserted or existing objects to be transferred from an old set to a new set. The other one was the capability to query on transient objects by using the symbol ":" to determine whether the target objects are transient or persistent.

A.3 An OQL Application and Its Preprocessing

A.3.1 Hospital Database

The hospital database is the first OQL application for Sentinel. It was implemented, based on the clinic schema proposed in [13], to test the functionality and the limitations of OQL[C++]. This application is presented in this section to demonstrate the usage of the OQL and its preprocessing. Figure A.1 illustrates the relationship amongst the classes in the schema.

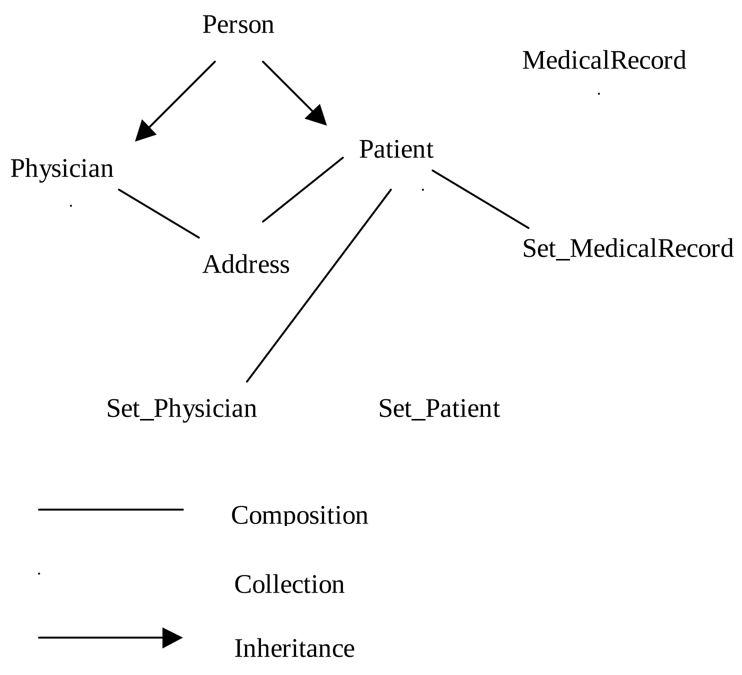


Figure A.1 Hospital Database Schema

Both **Patient** and **Physician** classes are inherited from **Person** class. A patient object has a set of objects of **MedicalRecord** and a set of objects of **Physician**. The following is the interface of these classes.

```
        #include "OpenOODB.h"
        #include <defmacro.h>[1]
        #include "Set.h"
        #include "List.h"

……………….

        class MedicalRecord {
        ……………
          };

        …………
```

[1] All application files using parameterized classes and queries should consistently include the **defmacro.h** file in all source files. The **defmacro.h** file should be included before the first template file is included. Applications planning to use persistence should include the **defmacro.h** file after the **OpenOODB.h** file is included.

```
DECLARE Set<MedicalRecord>

class Patient : public Person {
……………………..
public:
……………
Set_MedicalRecord records();
Physician&  family_dr();
……………
};

…………

DECLARE Set<Patient>
….…….
```

The following is a piece of code from the implementation body of the schema:

```
//MedicalRecord class implementation
  MedicalRecord::MedicalRecord() {………..}

  ..........

  IMPLEMENT Set<MedicalRecord>
  IMPLEMENT List<MedicalRecord>

  …………………..

  // Patient class implememtation
  Set_MedicalRecord Patient:: records(){………}

  …………………...

  IMPLEMENT Set<Patient>

………………………
```

Note: The **DECLARE** and **IMPLEMENT** are macro facilities supported by OpenOODB from Texas Instruments. The macro will be expanded by CoolCpp preprocessor to parameterized collection classes [12]. Open OODB currently supports two parameterized collection classes: List<Type> and Set<Tyep> [13]. They are defined and implemented in "Set.h" and "List.h" that come with Open OODB toolkit.

The following is a simple OQL statement that retrieves a set of patients whose patient id is equal to the variable "id".

```
…………………
Set_Patient *r = NULL;
Query {
r = SELECT *
FROM Patient* patient1 IN set_patient
WHERE  patient1->pat_id()==id;
}
………………
```

Note: The key word **Query** will be recognized by OQL preprocessor (CoolCpp) and the statement demarcated by "{" and "}" will be translated to the corresponding C++ code.

A.3.2 Preprocessing

All programs using the parameterized collections or OQL need to be preprocessed by the CoolCpp preprocessor before they are compiled by a C++ compiler. The following lines are extracted from the Makefile of the hospital application.

```
$(CCC) $(CCCFLAGS) -E $(INCLUDES) $*.C > /tmp/$*1.C
$(COOLCPP) -C $(INCLUDES) /tmp/$*1.C /tmp/$*.C
$(PPCC) $(CCCFLAGS)  $(INCLUDES) -w -c  /tmp/$*.C -o $@
```

First, the standard C preprocessor is invoked to include all system include files. The standard C preprocessor ignores all **DECLARE**, **IMPLEMENT**, or **Query** key words. Second, the CoolCpp preprocessor is invoked to expand all **DECLARE, IMPLEMENT** macros and OQL statements delimited by "Query {" and "}". At the end of this stage, the resulting files contain no #include, **DECLARE**, **IMPLEMENT,** or OQL statements. Finally, the C++ compiler is invoked.

The following is the code expanded by " DECLARE Set<Patient>"

```
class List_Patient { ……………… }

class Set_Patient {
……………..
void First_Member ( Iterator & ) ;
int Insert_Member ( Patient & ) ;
Patient & Get_Member ( Iterator & ) ;
void Next_Member ( Iterator & ) ;
int Is_End_Of_Set ( Iterator & ) ;
………………….
protected :
List_Patient & Get_List_Node ( Iterator & ) ;
List_Patient * header ;
……………………..
};
```

The following is the code translated from the OQL example in A.3.1.

```
Set_Patient * r = 0 ;
Patient *patient1;
Set_Patient *result_12400 = new Set_Patient ();
Set_Patient *set_12400;
set_12400 = (Set_Patient*) OpenOODB->fetch( "set_patient" );
if(set_12400 != 0) {
  Iterator  iter_12400;
  for(set_12400->First_Member(iter_12400);
        !set_12400->Is_End_Of_Set(iter_12400);
        set_12400->Next_Member(iter_12400) ) {
       patient1 = (&set_12400->Get_Member(iter_12400)) ;
       if( patient1->pat_id()==id ) {
         result_12400->Insert_Member( set_12400-> >Get_Member( iter_12400 ) );
            }
       }
r = result_12400;
}
else {
        fprintf(&__iob[2],"set_patient is not a named set\n");
        r = result_12400;
}
```

A.4 More OQL Examples

The following are some OQL statements that demonstrate the functionality of OQL.

- Retrieve the patient whose id is 125.
  This query example illustrates that a range variable can be declared as a pointer just like in C or C++.

```
SELECT *
FROM Patient* patient1 IN patients
WHERE  patient1->pat_id()== 125
```

- Retrieve John's patients whose names are Bob.
  This query example illustrates nested query using IN clause.

```
SELECT *
FROM Patient patient1 IN patients
WHERE patient1.family_dr().Name() == "John" &&
patient1
IN (select * from Patient patient2 IN patients
     where patient2.Name() == "Bob" )
```

- Retrieve the physicians of the patients in "patients" whoever has "lung CA".
  This query example illustrates nested query using EXISTS clause.

```
SELECT *
FROM  Physician physician1 IN physicians
WHERE
EXISTS(
SELECT *
FROM Patient patient1 IN patients
WHERE
patient1.family_dr().doc_id() == physician1.doc_id()
AND  EXISTS(
SELECT *
FROM MedicalRecord record1 IN :patient1.records()
WHERE record1.diagnosis()=="lung CA"))
```

Note: The ':' in ":patient1.records()" means the set of objects are transient.

- Retrieve John's patients whose names are not Bob.
  This query example illustrates nested query using NOT XXXX IN clause.

```
SELECT *
FROM Patient patient1 IN patients
```

```
WHERE patient1.family_dr().Name() == "John" &&
NOT patient1
IN (select * from Patient patient2 IN patients
where patient2.Name() == "Bob" )
```

- Retrieve the physicians of the patients in "patients" who never had "lung CA".
  This query example illustrates nested query using NOT EXISTS clause.

```
SELECT *
FROM  Physician physician1 IN physicians
WHERE
NOT EXISTS(
SELECT *
FROM Patient patient1 IN patients
WHERE  patient1.family_dr().doc_id() == physician1.doc_id()
AND  EXISTS(
SELECT *
FROM MedicalRecord record1 IN :patient1.records()
WHERE record1.diagnosis()=="lung CA"))
```

- Query all patients who live in the same city as doctor "John".
  This query example shows the join in OQL.

```
SELECT patient1
FROM Patient patient1 IN patients,
Physician physician1 IN physicians
WHERE patient1.family_dr().Name() == "John" &&
physician1.Name() == "John"  AND
!strcmp( patient1.home_address().city(), physician1.home_address().city())
```

- Retrieve only the name and age information of patients whose age are between 10 and 40.
  This query example shows projection.

```
SELECT test (p.Name(), p.age())
FROM  Patient p IN patients
WHERE p.age() > 10 AND p.age() < 40
```

- Query doctor John's patients who have "flu".
  This query example shows that user defined functions are allowed to appear in the WHERE clause in SELECT-FROM-WHERE structure.

```
SELECT *
FROM Patient patient1 IN patients
WHERE compare_equal(patient1.family_dr().Name() ,"John")
AND have_record( patient1 ,"flu")
```

Where the “compare_equal” and “have_record” are user-defined boolean functions.

- Modify doctor John's patients' ID.
  This query demonstrates update clause.

```
UPDATE (patient1.pat_id(999))
FROM Patient patient1 IN patients
WHERE patient1.pat_id()==127
```

- Insert a new patient to "patients".
  This query demonstrates insert clause.

```
INSERT Patient(
PTR Physician(Address("16th Ave.","Gainseville","FL"),"OBE",001,
Person(Address("16th Ave.","Gainseville","FL"),"John",40)),
007,
Set_MedicalRecord(MedicalRecord(9998,"Liver CA","10/25/95"),
MedicalRecord(9999,"TB","05/10/95")),
Person(Address("16th Ave.#140","Gainseville","FL"),"Test",60))
INTO patients
```

- Transfer all patients of Dr. John in "set_patient" to another set named "John_patients".
  This query demonstrates another usage of insert clause.

```
INSERT INTO John_patients
FROM Patient patient1 IN set_patient
WHERE compare_equal(patient1.family_dr().name() , "John");
```

- Delete John's patient whose id is 125.
  This query demonstrates delete clause.

```
DELETE FROM Patient p IN patients
WHERE p.family_dr().Name()== "John"
AND p.pat_id()== 123
```

A.5 Limitations

Current OQL is still not complete. The following is the summary of its limitations:

- Query Processing

  The code is not optimized.

No dynamic queries are supported.

- Coverage of the language

```
SELECT <objects>
FROM <range variable> IN <set>
WHERE <predicate>
```

<objects>

the same type of the range variable (e.g. *)	(YES)
new object type (e.g. project and join[1])	(YES)

<set>

set object name stored in database	(YES)
transient objects	(YES)
user defined set-valued function	(NO)
collection types other than Set	(NO)
heterogeneous collection	(NO)

<predicate>

path expression (chain of member functions)	(YES)
user-defined Boolean-valued function	(YES)
nested query statements	(limited)

The nested query statements should include the clauses IN, CONTAINS, ALL, ANY, and EXISTS. Currently, only IN and EXISTS clauses are supported.

A.6 Demo Programs of OQL

There are three demo programs designed and implemented to facilitate the understanding of OQL. The first one is written for OQL beginners. It is a menu-driven OQL application in which OQL statements with different functionality are hard coded in the program. Users who follow the menu will get a whole idea of OQL. The top most level of the menu includes the following:

```
***  1. CREATE DATABASE        ***
***  2. QUERY DATA             ***
***  3. MODIFY DATA            ***
***  4. DROP DATABASE          ***
***  5. LIMITATIONS            ***
***  6. THINGS TO DO           ***
```

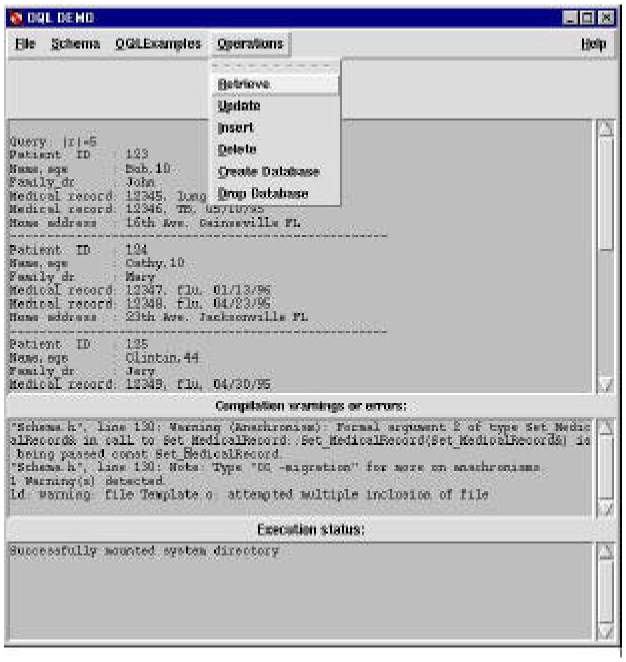
[1] Only two-set joins are supported.

*** 7. OQL REPORT ***
*** 8. INTRODUCTION ***

The second demo program is a schema independent platform with TCL interface using which users can give ad hoc OQL statements. The back-end mechanism is described as follows: The system has two template files with some question marks. One is for the query statement that retrieves something, such as SELECT statements. The other is for the statement that does not retrieve anything, such as INSERT or DELETE statement. Those question marks will be replaced with the information got from the users through the interface. The completed files are then complied and executed. The compilation errors/warnings, run-time error, and results are redirected to and displayed in different windows. If the input statement is a projection, depending on the types (attributes) of the return objects, a program named "MakeProjectClass" is invoked to produce a template class definition and its implementation. The template class is then replaced with one question mark in the template file. The C++ code generated from the given statement will call the constructor of the template class to create resulting objects. This demo program also provides an option to users to see the C++ code that will be generated from the given statements before they are compiled. The transformation from OQL to C++ is done by a program named "oql". This program is the main portion of the OQL preprocessor.

The third demo program is an extension of the second one. The only difference is that it supports active schemas by calling the Sentinel preprocessor and provides an entry to invoke events by executing the method functions of the resulting objects.

The following windows are the GUI for the third demo program.

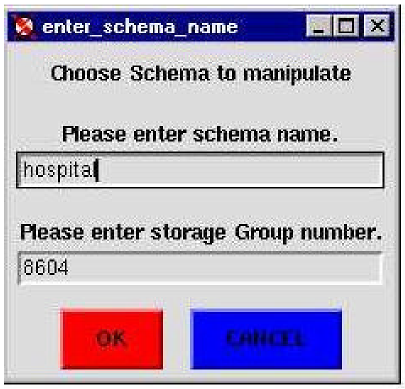


```
Query: |z|=5
Patient  ID    : 123
Name, age      : Bob,10
Family_dr      : John
Medical record: 12345, lung
Medical record: 12346, TB, 05/10/95
Home address   : 16th Ave. Gainesville FL
------------------------------------------------------------
Patient  ID    : 124
Name, age      : Cathy,10
Family_dr      : Mary
Medical record: 12347, flu, 01/13/96
Medical record: 12348, flu, 04/23/96
Home address   : 23th Ave. Jacksonville FL
------------------------------------------------------------
Patient  ID    : 125
Name, age      : Clintin,44
Family_dr      : Jery
Medical record: 12349, flu, 04/30/95
```

```
"Schema.h", line 130: Warning (Anachronism): Formal argument 2 of type Set_MedicalRecord& in call to Set_MedicalRecord::Set_MedicalRecord(Set_MedicalRecord&) is being passed const Set_MedicalRecord.
"Schema.h", line 130: Note: Type "CC -migration" for more on anachronisms.
1 Warning(s) detected.
ld: warning: file Template.o: attempted multiple inclusion of file
```

```
Successfully mounted system directory
```

A.2



Choose Schema to manipulate

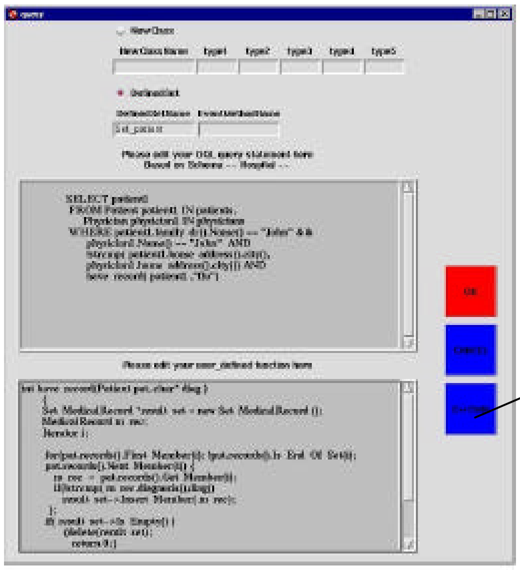
Please enter schema name.

hospital

Please enter storage Group number.

8604

A.3



Set_patient

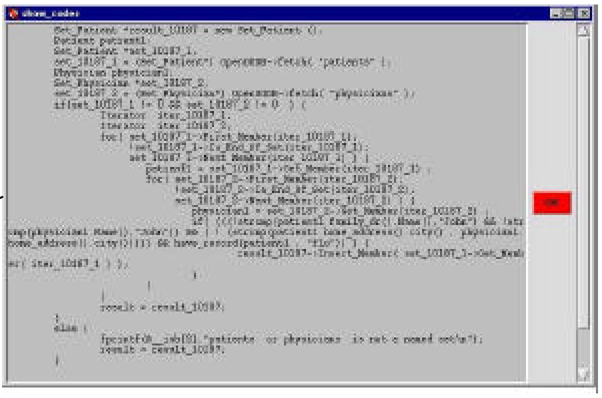
Please edit your OQL query statement here
Based on Schema -- Hospital --

```
SELECT patient1
 FROM Patient patient1 IN patients,
    Physician physician1 IN physicians
 WHERE patient1.family_dr().Name() == "John" &&
      physician1.Name() == "John" AND
      !strcmp( patient1.home_address().city(),
      physician1.home_address().city()) AND
      have_record( patient1 , "flu")
```

Please edit your user_defined function here

```
int have_record(Patient pat, char* diag )
    {
    Set_MedicalRecord *result_set = new Set_MedicalRecord ();
    MedicalRecord m_rec;
    Iterator i;

     for(pat.records().First_Member(i); !pat.records().Is_End_Of_Set(i);
     pat.records().Next_Member(i)) {
       m_rec = pat.records().Get_Member(i);
       if(!strcmp( m_rec.diagnosis(),diag))
         result_set->Insert_Member( m_rec);
      };
     if( result_set->Is_Empty() )
        {delete(result_set);
         return 0;}
```

A.4



```
        Set_Patient *result_10187 = new Set_Patient ();
        Patient patient1;
        Set_Patient *set_10187_1;
        set_10187_1 = (Set_Patient*) operBMB->fetch( "patients" );
        Physician physician1;
        Set_Physician *set_10187_2;
        set_10187_2 = (Set_Physician*) operBMB->fetch( "physicians" );
        if(set_10187_1 != 0 && set_10187_2 != 0 ) {
                Iterator  iter_10187_1;
                Iterator  iter_10187_2;
                for( set_10187_1->First_Member(iter_10187_1);
                     !set_10187_1->Is_End_Of_Set(iter_10187_1);
                     set_10187_1->Next_Member(iter_10187_1) ) {
                        patient1 = set_10187_1->Get_Member(iter_10187_1) ;
                        for( set_10187_2->First_Member(iter_10187_2);
                             !set_10187_2->Is_End_Of_Set(iter_10187_2);
                             set_10187_2->Next_Member(iter_10187_2) ) {
                                physician1 = set_10187_2->Get_Member(iter_10187_2) ;
                                if( (((!strcmp(patient1.family_dr().Name(),"John") && !strcmp(physician1.Name(),"John")) && ) ) (!strcmp(patient1.home_address().city() , physician1.home_address().city()))) && have_record(patient1 , "flu"))) ) {
                                        result_10187->Insert_Member( set_10187_1->Get_Member( iter_10187_1 ) );
                                }
                        }
                }
                result = result_10187;
        }
        else {
                fprintf(__iob[2],"patients  or physicians  is not a named set\n");
                result = result_10187;
        }
```

A.5

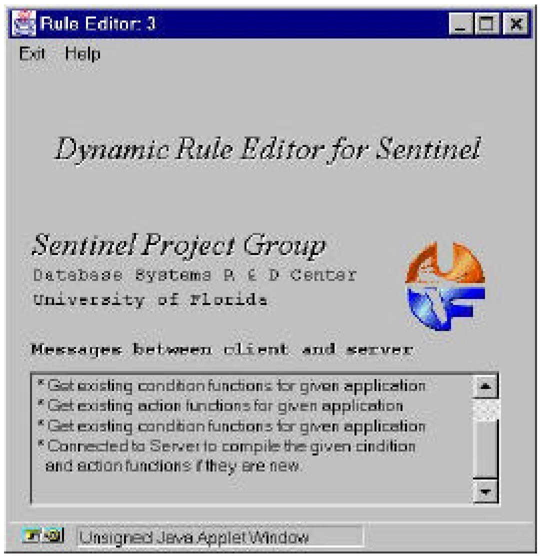
Figure A.2 OQL Demo Platform - Main

Figure A.3 Choosing A Schema

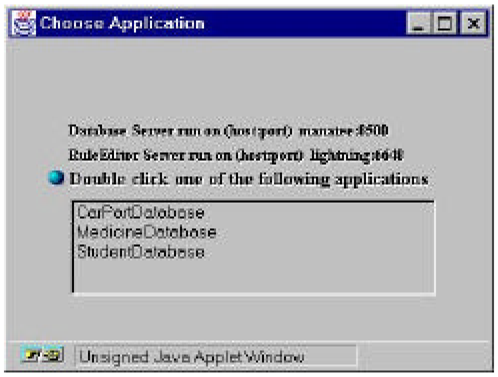
Figure A.4 OQL Demo Platform - Query Editor

Figure A.5 OQL Demo Platform - C++ Code

APPENDIX B
WINDOW SAMPLES OF RULE EDIOR INTERFACE


Rule Editor: 3
Exit
Help
Dynamic Rule Editor for Sentinel
Sentinel Project Group
Database Systems R & D Center
University of Florida
Messages between client and server
Unsigned Java Applet Window

* Get existing condition functions for given application
* Get existing action functions for given application
* Get existing condition functions for given application
* Connected to Server to compile the given cindition and action functions if they are new.



Database Server run on (host:port) manatee:8500
RuleEditor Server run on (host:port) lightning:6640
Double click one of the following applications

CarPortDatabase
MedicineDatabase
StudentDatabase

Figure B.1 Rule Editor - Logo

Figure B.2 Rule Editor -- Applications

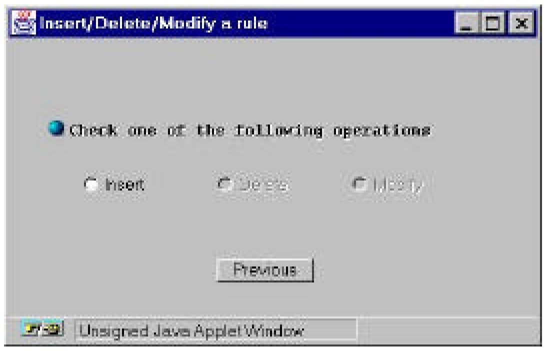

Insert/Delete/Modify a rule
Check one of the following operations
Insert
Delete
Modify
Previous
Unsigned Java Applet Window

Figure B.3 Rule Editor - Operations

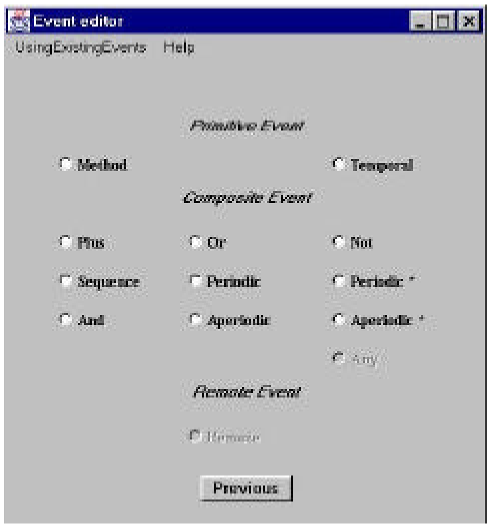
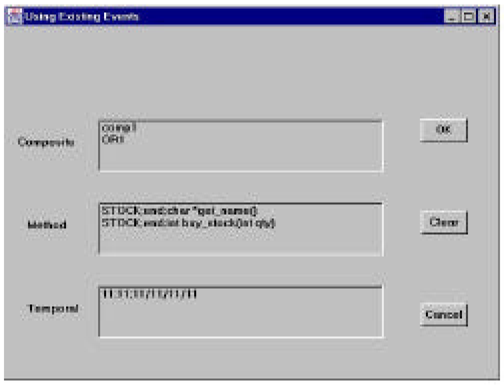


Figure B.4 Rule Editor - Event Operators

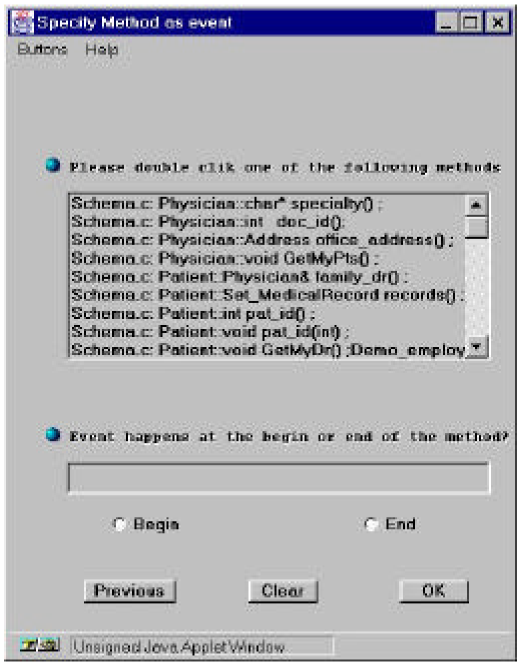


```
comp1
OR1
```

```
STOCK:end:char *get_name()
STOCK:end:int buy_stock(int qty)
```

```
11:11:11/11/11/11
```

Figure B.5 Rule Editor - Existing Events

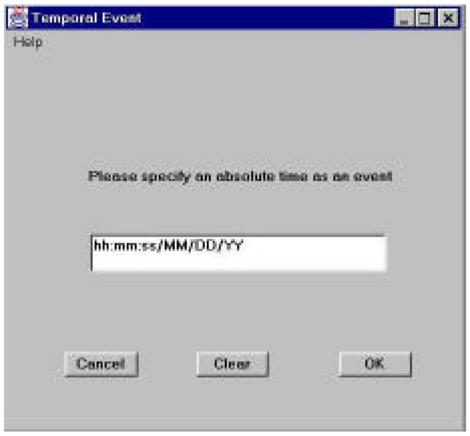


Please double clik one of the following methods

```
Schema.c: Physician::char* specialty() ;
Schema.c: Physician::int  doc_id();
Schema.c: Physician::Address office_address() ;
Schema.c: Physician::void GetMyPts() ;
Schema.c: Patient::Physician& family_dr() ;
Schema.c: Patient::Set_MedicalRecord records() ;
Schema.c: Patient::int pat_id() ;
Schema.c: Patient::void pat_id(int) ;
Schema.c: Patient::void GetMyDr() ;Demo_employ
```

Event happens at the begin or end of the method?

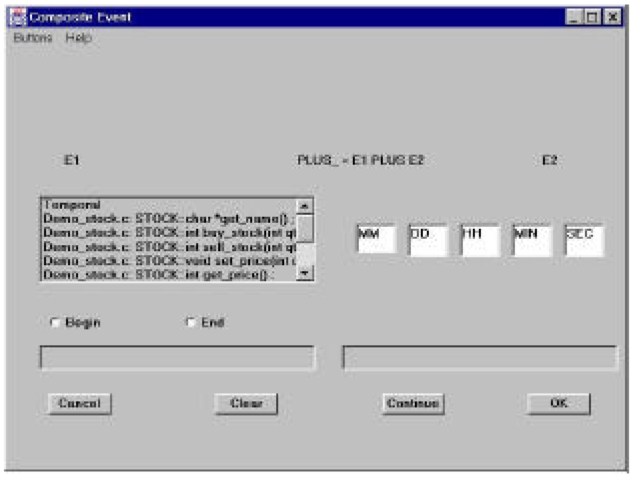
Figure B.6 Rule Editor - Methods



Please specify an absolute time as an event

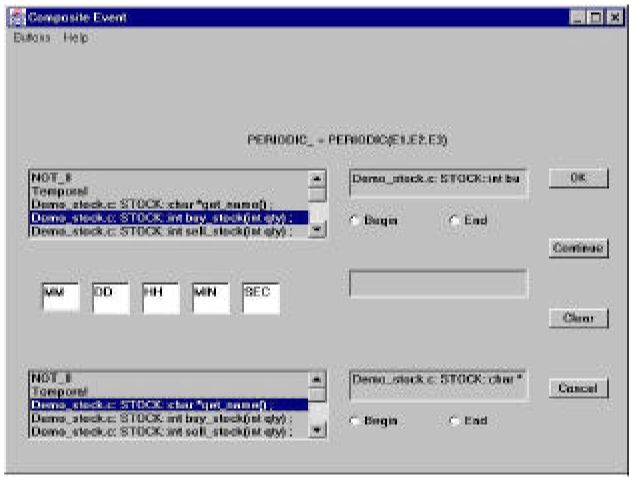
```
hh:mm:ss/MM/DD/YY
```

Figure B.7 Rule Editor - Temporal Events



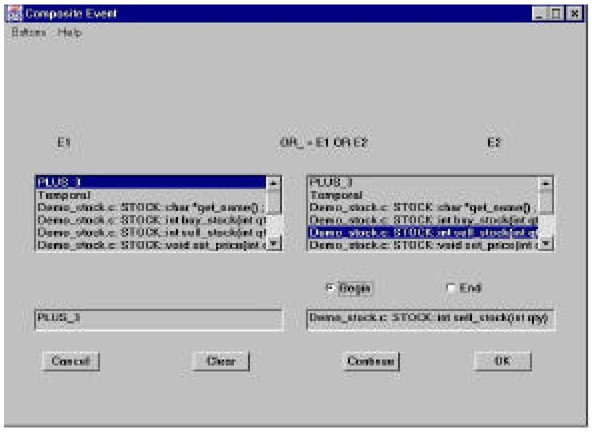
```
Temporal
Demo_stock.c: STOCK::char *get_name() ;
Demo_stock.c: STOCK::int buy_stock(int qt
Demo_stock.c: STOCK::int sell_stock(int qt
Demo_stock.c: STOCK::void set_price(int c
Demo_stock.c: STOCK::int get_price() ;
```

Figure B.8 Rule Editor - Plus Editor



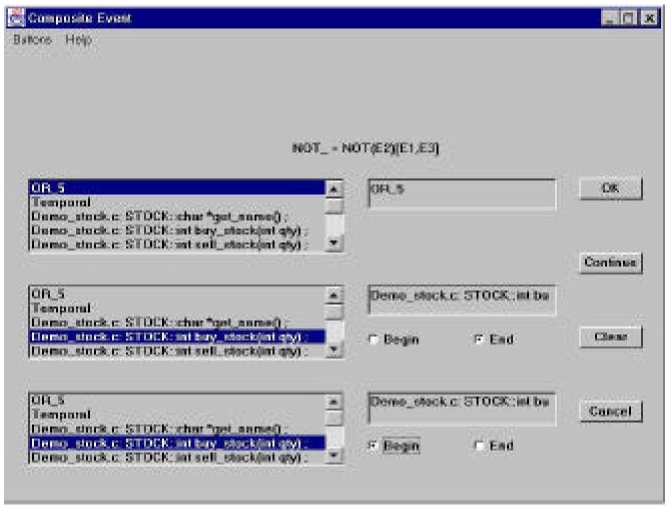
```
PERIODIC_ = PERIODIC(E1,E2,E3)

NOT_8
Temporal
Demo_stock.c: STOCK::char *get_name() ;
Demo_stock.c: STOCK::int buy_stock(int qty) ;
Demo_stock.c: STOCK::int sell_stock(int qty) ;

Demo_stock.c: STOCK::int bu
( ) Begin    ( ) End

NOT_8
Temporal
Demo_stock.c: STOCK::char *get_name() ;
Demo_stock.c: STOCK::int buy_stock(int qty) ;
Demo_stock.c: STOCK::int sell_stock(int qty) ;

Demo_stock.c: STOCK::char *
( ) Begin    ( ) End
```

Figure B.9 Rule Editor - Periodic Editor



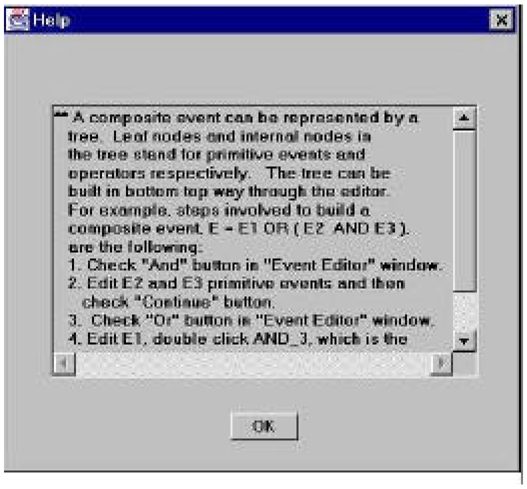
```
OR_ = E1 OR E2

PLUS_3
Temporal
Demo_stock.c: STOCK::char *get_name() ;
Demo_stock.c: STOCK::int buy_stock(int qt
Demo_stock.c: STOCK::int sell_stock(int qt
Demo_stock.c: STOCK::void set_price(int

PLUS_3
Temporal
Demo_stock.c: STOCK::char *get_name() ;
Demo_stock.c: STOCK::int buy_stock(int qt
Demo_stock.c: STOCK::int sell_stock(int q
Demo_stock.c: STOCK::void set_price(int

[✓ Begin]    ( ) End

PLUS_3
Demo_stock.c: STOCK::int sell_stock(int qty)
```

Figure B.10 Rule Editor - Or Editor



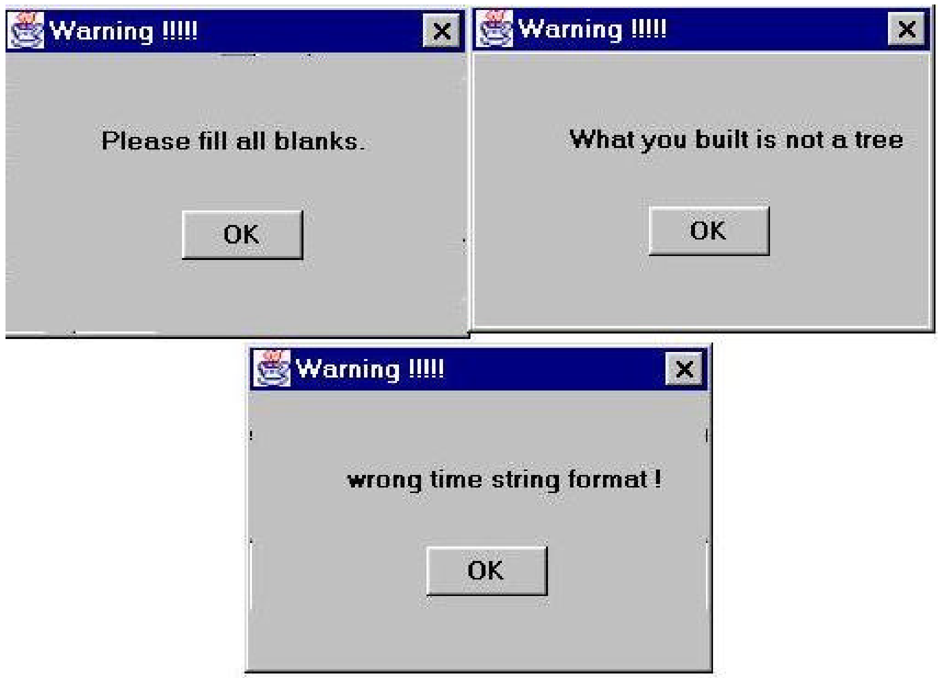
```
NOT_ = NOT(E2)[E1,E3]

OR_5
Temporal
Demo_stock.c: STOCK::char *get_name() ;
Demo_stock.c: STOCK::int buy_stock(int qty) ;
Demo_stock.c: STOCK::int sell_stock(int qty) ;

OR_5

OR_5
Temporal
Demo_stock.c: STOCK::char *get_name() ;
Demo_stock.c: STOCK::int buy_stock(int qty) ;
Demo_stock.c: STOCK::int sell_stock(int qty) ;

Demo_stock.c: STOCK::int bu
( ) Begin    [✓ End]

OR_5
Temporal
Demo_stock.c: STOCK::char *get_name() ;
Demo_stock.c: STOCK::int buy_stock(int qty) ;
Demo_stock.c: STOCK::int sell_stock(int qty) ;

Demo_stock.c: STOCK::int bu
[✓ Begin]    ( ) End
```

Figure B.11 Rule Editor - Not Editor



```
** A composite event can be represented by a
tree.  Leaf nodes and internal nodes in
the tree stand for primitive events and
operators respectively.   The tree can be
built in bottom top way through the editor.
For example, steps involved to build a
composite event, E = E1 OR ( E2  AND E3 ),
are the following:
1. Check "And" button in "Event Editor" window.
2. Edit E2 and E3 primitive events and then
   check "Continue" button.
3.  Check "Or" button in "Event Editor" window.
4. Edit E1, double click AND_3, which is the
```

Figure B.12 Rule Editor - Help

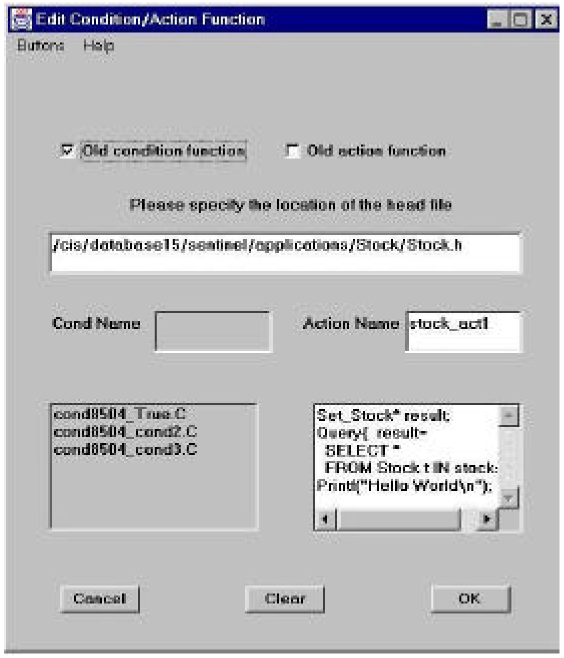


Please fill all blanks.

What you built is not a tree

wrong time string format !

Figure B.13 Rule Editor – Warnings



[✓ Old condition function] Old action function

Please specify the location of the head file

/cis/database15/sentinel/applications/Stock/Stock.h

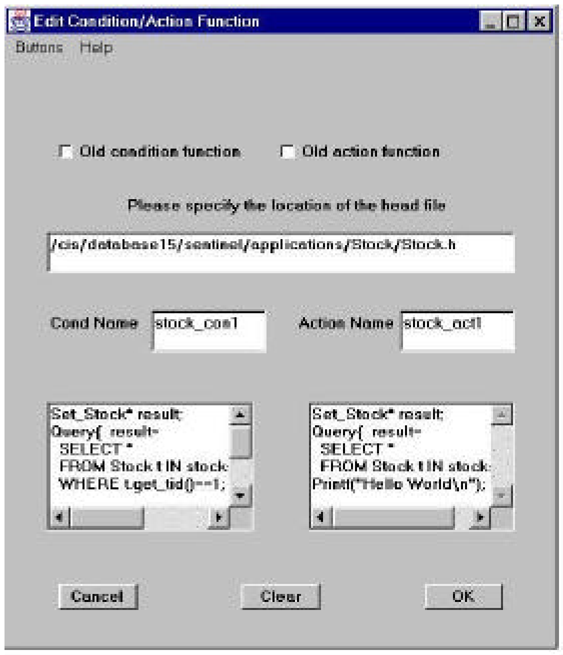
Cond Name:

Action Name: stock_act1

cond8504_True.C
cond8504_cond2.C
cond8504_cond3.C

```
Set_Stock* result;
Query{  result=
  SELECT *
  FROM Stock t IN stock;
Printf("Hello World\n");
```

Figure B.14 Rule Editor - Function Editor 1



Old condition function Old action function

Please specify the location of the head file

/cis/database15/sentinel/applications/Stock/Stock.h

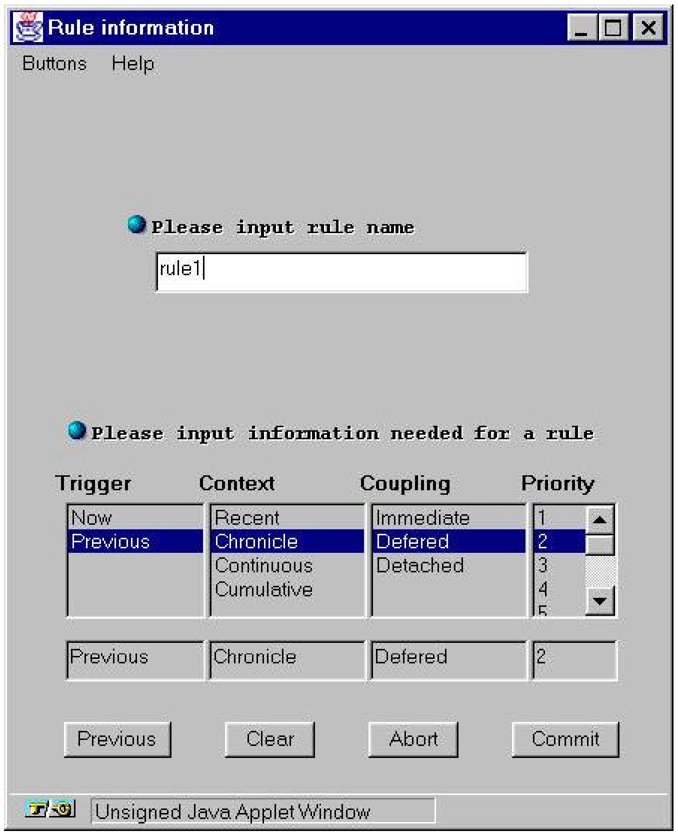
Cond Name: stock_con1

Action Name: stock_act1

```
Set_Stock* result;
Query{  result=
  SELECT *
  FROM Stock t IN stock;
  WHERE t.get_tid()==1;
```

```
Set_Stock* result;
Query{  result=
  SELECT *
  FROM Stock t IN stock;
Printf("Hello World\n");
```

Figure B.15 Rule Editor - Function Editor 2



Please input rule name

rule1

Please input information needed for a rule

Trigger	Context	Coupling	Priority
Now	Recent	Immediate	1
Previous	Chronicle	Defered	2
	Continuous	Detached	3
	Cumulative		4
			5
Previous	Chronicle	Defered	2

Figure B.16 Rule Editor - Rule Attribute Editor

REFERENCES

[1] M. Prahlad. A Dynamic Rule Editor for Sentinel: Design and Implementation. Master's thesis, University of Florida, Gainesville, 1997.

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

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

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

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

[6] H. Liao. Global Events In Sentinel: Design and Implementaion of A Global Event Detector, University of Florida, Gainesville, 1997

[7] H. Lee. Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing. Master's thesis, University of Florida, Gainesville, 1996

[8] 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.

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

[10] S. Han. Three-Tire Architecture for Sentinel Applications and Tools: Separating Presentation from Functionality, University of Florida, Gainesville, 1997

[11] OpenOODB 1.0 C++ API User Manual. Texas Instruments, Dallas, September 1995.

[12] OpenOODB 1.0 Query Language User Manual. Texas Instruments, Dallas, September 1995.

[13] Jose A. Blakeley. ZQL[C++]: Extending the C++ Language with an Object Query Capability, Texas Instruments, Dallas, May 1993.

[14] Jose A. Blakeley and Kiran SreeWastav. Implementation of the Object Query Language Module for Zeitgeist: Version 0.1.0. Database System Branch Technical Report, Information Technology Laboratory, Texas Instruments, Dallas, May 1991.

BIOGRAPHICAL SKETCH

Hung-ju Chu was born on June 9, 1966, at Taichung, Taiwan, R.O.C. He received the Bachelor of Science degree in Electrical Engineering from the National Cheng-Kung University, Tainan, Taiwan, in June 1994. In the fall of 1995, he started his graduate studies with a major in computer and information science and engineering at the University of Florida. He will receive his Master of Engineering degree in computer and information science and engineering from the University of Florida, Gainesville, in May 1998. His research interests include network programming, user interface and active object-oriented databases.