A DYNAMIC RULE EDITOR FOR SENTINEL: DESIGN AND IMPLEMENTATION

By

PRAHLAD MADABHUSHI

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

UNIVERSITY OF FLORIDA

1997

Dedicated to Shirdi Sai Baba
and my Parents

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Sharma Chakravarthy for giving me an opportunity to work on this challenging topic and for providing continuous guidance, advice, and support throughout the course of this research work. I thank Dr. Herman Lam and Dr. Haniph Latchman for serving on my supervisory committee and for their careful perusal of this thesis. I would like to thank Sharon Grant for maintaining a well administered research environment.

I am grateful to Shiby Thomas and Hyoungjin Kim for their invaluable help and fruitful discussions during the design and implementation of this work. I also take this oppurtunity to thank Aravind Yalamanchi, Ramana Nyapathy and Mahesh Jagannath for being helpful in times of need.

I also thank my best friends Madhav Durbha, Mahidhar Gundepudi and Sabrina Karkera for making my stay here in Gainesville a memorable one.

On a more personal note, I would like to thank my father, Dr. M.Padmanabha Iyyangar, my mother, Mrs. Srinivasamma, my grandmother P.T Sarala Devi, my great brother, Kalyan Ram, my sister-in-law Vaishali, my wonderful sister, Rekha, my brother-in-law Ramakanth. Without their love, support, and constant encouragement this work would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

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

A DYNAMIC RULE EDITOR FOR SENTINEL:
DESIGN AND IMPLEMENTATION

By

Prahlad Madabhushi

August 1997

Chairman: Dr. Herman Lam
Cochairman: Dr. Sharma Chakravarthy
Major Department: Electrical and Computer Engineering

Over the last few years there has been a surge of interest in active databases. This is mainly because the needs of a variety of nontraditional applications that are not readily met by traditional database management systems (DBMSs). Active data bases have been proposed to meet some of the requirements of nontraditional applications. Active OODBMSs extend the normal functionality of OODBMSs with support for monitoring user-defined situations and reacting to them without user or application intervention. Furthermore, active databases provide an elegant means for supporting integrity constraints, access control, maintenance of derived data, and materialized views and snapshots.

Rules, in the context of an active DBMS, consist primarily of three components: an event, a condition, and an action. An event is an indicator of a happening (either simple or complex). The condition specifies an optional predicate over the database state which is evaluated when its corresponding event occurs. Actions are the operations to be performed when an event occurs and its associated condition evaluates to true.

An active OODBMS application can be written in two types of environments, namely Compilation-based environment and Interpreter-based Environment. In the first approach the user would have to define the rules in the source code and compile to source code, whereas in the second approach, there is an interpreter, which interprets all the code written and there is no need to complie it.

Rules can be classified into two types. They are Application-level rules and External rules. Application level rules are based on events within the application(i.e, rules which are hardcoded in the application). External rules are based on existing/potential events which are added from outside the application.

External Rules can be added to an application in both kinds of environments. In the compilation-based approach, we need to edit the source code to add new rules to the existing application for them to be triggered, whereas in the interpreter-based approach, there will be an interpreter which will interpret the changes made to the application and the rules can be defined on an event externally from outside the application without editing the application source

Sentinel uses compilation-based environment. The problem here in using the compilation-based environment is that Pointers to functions(unlike pointers in data structure) cannot be persisted and used later. In sentinel, conditions and actions are functions. And in a C++ environment, the pointers to these condition and action functions are bound at compilation/Link Time and are used at runtime. Hence, by making the condition and action functions, part of the application code, the above difficulty is avoided. Once, the application has been developed, addition, deletion or modification of rules requires changes to the source code, recompilation and relinking.

The objective of this thesis is to provide user the capability of adding database rules from outside the application in an compilation-based environment and see the behaviour of the rule execution without changing the application source code.

CHAPTER 1
INTRODUCTION

Over the last few years there has been a surge of interest in active databases. This is mainly because the needs of a variety of nontraditional applications that are not readily met by traditional database management systems(DBMSs). These applications–network management, air traffic control, program trading, computer integarted manufacturing(CIM), and engineering design, to name a few - need to, as part of their application semantics, continually monitor changes to the database state and initiate appropriate actions, preferably, without user or application intervention.

Active data bases have been proposed to meet some of the requirements of non-traditional applications. Active OODBMSs extend the normal functionality of OODBMSs with support for monitoring user-defined situations and reacting to them without user or application intervention. These DBMSs continuously monitor situations to initiate appropriate actions in response to data base updates, occurrence of particular states or transition between states, possibly within a response time window. Furthermore, active databases provide an elegant means for supporting integrity constraints, access control, maintenance of derived data, and materialized views and snapshots.

Rules, in the context of an active DBMS, consist primarily of three components: an event, a condition, and an action. An event is an indicator of a happening (either simple or complex). Events are recognized by the system or signalled by the user. For example, database events such as insert, delete, and update are detected by the OODBMS. The condition specifies an optional predicate over the database state which is evaluated when its corresponding event occurs. The conditions to be

monitored may be arbitrarily complex and may be defined not only on single data values or individual database states, but also on sets of data objects, transitions between states of materialized/derived objects, trends and historical data. Actions are the operations to be performed when an event occurs and its associated condition evaluates to true. Actions may be programs which may in turn cause the occurrence of other events. Once rules are specified declaratively to the system, it is the system's responsibility to monitor the situations (event-condition pairs) and execute the corresponding action when the condition is satisfied without any user or application intervention. The advantage of using rules as a means of providing active behavior is the freedom from explicitly hard-wiring code which checks the situations being monitored into each program that updates the database.

1.1 Previous Research On Active OODBMS

1.1.1 Ode

Ode [1] is a database system and environment based on object paradigm. The database is defined, queried and manipulated using the database programming language O++, which is an upward-compatible extension of the object-oriented programming language C++. O++ extends C++ by providing facilities suitable for database applications, such as facilities for creating persistent and versioned objects, defining and manipulating sets, organizing persistent objects into clusters, iterating over clusters of persistent objects, and associating constraints and triggers with objects.

Treatment of rules

Ode provides active behaviour by incorporating rules in the form of constraints and triggers. Both constraints and triggers consist of a condition and a action, and

are defined within a class definition. Events in Ode are implicit and are considered as the disjunction of all nonconstant public methods. Constraints are used to maintain the notion of object consistency and hence are applicable to all instances of the class in which they are declared. Triggers, on the other hand, are used for monitoring database conditions other than those representing consistency violations and are applicable only to those instances specified explicitly by the user at runtime. Triggers in Ode are parameterized. The activation of all types of triggers occurs explicitly by the user. Triggers are checked at the end of each method and are appended to a to-be-executed list, if they evaluate to true. Trigger bodies are executed in seperate transactions after the commit(not necessarily immediately after) of the transaction firing them. More recently Ode has proposed a language for specifying composite events. They specify composite events using a set of operators. Events are declared within a class definition. Basic(primitive) events are defined and composite events are constructed by applying operators to basic events. The basic events supported are object state events, method execution events, timed events and transaction events. The event operators supported are relative, prior, sequence, choose, every, fa, and faAbs. Detection of events is accomplished by using a finite automata.

1.1.2 ADAM

Adam is an active OODB implemented in PROLOG. It focuses on providing a uniform approach to the treatment of rules in an object-oriented environment.

Treatment of rules

Rules are incorporated in ADAM by using an object Based Mechanism. Both the events and rules are treated as first class objects which are created, deleted and modified in the same fashion as other objects. An object's definition is enlarged to indicate which rules to check when the object raises an event. Thus each class structure is

augmented with a class-rules attribute; this attribute has as its value the set of rules that are to be checked when the class raises an event. A Rule-class is defined where each rule is an instance of that class. The structure of the Rule-class consists of the attributes event, active-class, is-it-enabled, disabled-for, condition and action. The event attribute indicates the event which triggers the rule, the active-class attribute indicates the class name on which the rule is applicable, the is-it-enable attribute specifies whether the rule is enabled or not, the disabled-for attribute has as its value the set of instances for which the rule is disabled while the condition and action attributes specify the rule's condition and action, respectively. Rule operations are implemented as class methods. Only primitive events are supported.

The Advantages of using ADAM are as follows:

- Since complex events are not supported in ADAM, attaching the rules as attributes to a class leads to efficient rule detection.
- Rules are treated as objects as they can be created, deleted and modified like any other object.
- Inheritance of rules is supported, but it cannot be adopted to other environments.

1.1.3 SENTINEL

Sentinel is an active object-oriented database management system which support ECA rules in both centralized and now in a distributed environment. An event specification language Snoop has been developed to specify events which include local events (local *primitive* events and local *composite* events) as well as global events (global *primitive* events and global *composite* events). Two event detection mechanisms, namely, a *local event detector* and a *global event detector*, are implemented to monitor the behavior of local events as well as global events across applications.

Events and rules [2] [3] can be defined at the class level (inside a class definition) as well as at the instance level (outside of a class definition). A class level event and rule is applicable to every object of this class, whereas an instance level event and rule is applicable only to the specific object instance.

Treatment of rules

There should be an event, a condition, and an action to specify a rule. Rule can have atmost four options for detecting the event which it subscribes along with a few more parameters such as *coupling mode*, *parameter context event consumption mode* and *priority* of the rule.

Usage:

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

Among the optional parameters of a rule definition, the *parameter_context* should be specified first, and others can be given in any order using the keyword match. The *parameter_context* of the event which a rule subscribes should be placed right after action_function if it exists. If several rules need to be defined on the same event in different parameter contexts, then the rule definition has to be duplicated for each context. Parameter context can be one of the RECENT, CHRONICLE, CONTINUOUS, or CUMULATIVE.

Coupling mode defines the mode of the rule execution. An arbitrary number of totally ordered priority classes can be defined. Rule_Trigger_Mode can be used for specifying the time at which event occurences are to be considered for the rule. two options, NOW and PREVIOUS are supported as rule triggering modes.

The parameters of a *primitive* event are the parameters of the method that this *primitive* is declared on and the time of the event occurrence. The parameter of a *composite* event is the combination of parameters of its constituent events.

In Sentinel, multiple rule executions, nested rule executions and prioritized rule executions are supported. Two coupling modes, immediate and deferred are implemented.

1.1.4 SAMOS

SAMOS [4] is an active OODBMS. It supports two types of rules *class-internal rules* and *class-external rules* class internal rules are part of the class definition. They are encapsulated within instances and are visible to the class-implementor only. Conditions and actions of class-internal rules are allowed to operate directly on values. Class-external rules can be defined by any user or application independently of a class definition.

Rule execution

The rule definer specifies in SAMOS when a condition has to be evaluated and/or an action has to be executed relative to the triggering event by means of coupling modes (immediate, deferred, decoupled like in HiPAC). Examination of the integration of the execution of triggered operations within a transaction model based on multi-level transactions and semantic concurrency control. In this approach condition eveluation and action execution are implemented as own(sub) transactions. On the basis of semantic concurrency control on the level of methods, the system has to be told about conflict relations over the set of methods of a class.Class-external rules call methods which in turn are synchronized with other methods and rules. Class-internal rules, on the other hand, can manipulate values of objects directly, and thus behave comaprable to methods. Consequently, a class implementor has to provide conflict relations with condition or action parts of class-internal rules. Finally, SAMOS also handles the execution of multiple rules which are triggered by the same event by menas of priorities. However, this is only necessary when condition and action have

the same coupling mode. In this case, the efect of rules depends on the execution order: the action of one rule may invalidate the condition of others.

1.2 Motivation

An active OODBMS application can be written in two types of environments, namely Compilation-based environment and Interpreter-based Environment. In the compilation-based environment, the application is written in languages which have strong-type checking, like C, C++ and ADA. In an interpreter-based environment, the written code is interpreted by an interpreter and is written in languages like in Prolog, SmallTalk and CommonLisp. In the first approach the user would have to define the rules in the source code and compile to source code, whereas in the second approach, there is an interpreter, which interprets all the code written and there is no need to complie it.

Sentinel uses compilation-based environment. In Sentinel, to design an application, we first need to define schema as classes, then define methods, and then compile the application and link the application with sentinel libraries and then finally execute. Change to either schema or methods require recompilation of the changed source code and relinking to produce a new executable.

So the user has to painstakingly edit the source code to add a rule to a particular event and then compile the whole source code to see the changes. This becomes a tedious task especially if you keep adding, one rule at a time. Hence there is a need for the tool through which you can add rules to an existing application without going through all the above mentioned tasks. Dynamic Rule Editor has been implemented for that very pupose. All that a user needs to do to add a rule to an application is to add a rule through the user-friendly interface and then link the corresponding condition/action library to the application and he can see all the changes immedaitely.

The objective of this thesis is to provide the user the capability of adding database rules from outside the application in an compilation-based environment and see the behaviour of the rule execution without changing the application source code.

1.3 Organization of Thesis

This thesis is organized as follows. Chapter 2 explains how rules are supported in different systems by gicing an example. Chapter 3 explains the sentinel architecture in detail. Chapter 4 explains about the Architecture of the Dynamic Rule Editor. Chapter 5 discusses the design while Chapter 6 discusses the implementation issues involved. The final chapter, Chapter 7, presents the conclusion and highlights future areas of work.

CHAPTER 2
RELATED WORK

2.1 How Rules are Supported in Different Systems

2.1.1 ADAM

ADAM is an interpreter-based system. It is implemented in PROLOG. Rules and events in ADAM cannot be specified as a part of an object class specification. Both the events and rules are treated as first class objects which are created, deleted and modified in the same fashion as other objects. An object's definition is enlarged to indicate which rules to check when the object raises an event. Thus each class structure is augmented with a class-rules attribute; this attribute has as its value the set of rules that are to be checked when the class raises an event. A Rule-class is defined where each rule is an instance of that class. The structure of the Rule-class consists of the attributes event, active-class, is-it-enabled, disabled-for, condition and action. The event attribute indicates the event which triggers the rule, the active-class attribute indicates the class name on which the rule is applicable, the is-it-enable attribute specifies whether the rule is enabled or not, the disabled-for attribute has as its value the set of instances for which the rule is disabled while the condition and action attributes specify the rule's condition and action respectively. Rule operations are implemented as class methods.

2.1.2 Ode

Ode is a compiler-based system. It is implemented in O++. It incorporates rules in the form of constraints and triggers. Both constraints and triggers consist of a condition and action, and are defined within a class definition. Constraints are used to maintain the notion of object consistency and hence are applicable to all

instances of the class in which they are declared. Triggers, on the other hand, are used for monitoring database conditions other than those representing consistency violations and are applicable only to those instances specified explicitly by the user at runtime. Triggers in Ode are parameterized. The activation of all types of triggers occurs explicitly by the user. Triggers are checked at the end of each method and are appended to a to-be-executed list, if they evaluate to true. Trigger bodies are executed in seperate transactions after the commit(not necessarily immediately after) of the transaction firing them.

2.1.3 SAMOS

SAMOS is a compiler-based system. The rule definer specifies in SAMOS when a condition has to be evaluated and/or an action has to be executed relative to the triggering event by means of coupling modes. Examination of the integration of the execution of triggered operations within a transaction model based on multi-level transactions and semantic concurrency control. In this approach condition eveluation and action execution are implemented as own(sub) transactions. On the basis of semantic concurrency control on the level of methods, the system has to be told about conflict relations over the set of methods of a class.Class-external rules call methods which in turn are synchronized with other methods and rules. Class-internal rules, on the other hand, can manipulate values of objects directly, and thus behave comparable to methods. Consequently, a class implementor has to provide conflict relations with condition or action parts of class-internal rules. Finally, SAMOS also handles the execution of multiple rules which are triggered by the same event by means of priorities. However, this is only necessary when condition and action have the same coupling mode. In this case, the effect of rules depends on the execution order: the action of one rule may invalidate the condition of others.

CHAPTER 3
OVERVIEW OF SENTINEL AND RULE PROCESSING

3.1 Sentinel

3.1.1 Sentinel Architecture

The Sentinel(an active OODBMS being developed at the University of Florida) extends the passive Open OODB system. The Open OODB Toolkit uses Exodus as the storage manager and supports persistence of C++ objects. Concurrency control and recovery are provided by the Exodus storage manager. A full C++ pre-processor is used for transforming the user class definitions as well as the application code. Extensions incorporated for making the Open OODB active are as follows:

- Specification of ECA rules either as a part of the class definition or as a part of an application. This is preprocessed (by using an enhanced C++ pre-processor) into appropriate code for detection and rule execution.

- Detection of primitive events by using the sentry mechanism of the Open OODB. Sentry mechanism provides a wrapper method that permits us to invoke notification of an event to the composite event detector.

- A composite event detector for detecting composite events in various contexts. There is a composite event detector for each open OODB application or client (each application of Open OODB is a client to the Exodus server).

- A transaction manager for executing rules. Light weight processes are used both for prioritized and concurrent rule execution.

3.1.2 Compilation-Based Approach vs Interpreter-Based Approach

An application can be written in two types of environments, namely compilation-based environment and interpreter-based environment. In compilation-based environment, the application is written in languages which have strong-type checking, like C, C++ and ADA. In an interpreter-based environment, the written code is interpreted by an interpreter and is written in languages like in Prolog, SmallTalk and CommonLisp. In the first approach the user would have to define the rules in the source code and compile to source code, whereas in the second approach, there is an interpreter, which interprets all the code written and there is no need to compile it.

In Sentinel, events and rules can be specified either as part of the schema(class definitions) or as part of the application(main program). Now, Sentinel processes the above and generates C++ code. The generated code is inserted into the application(in main program) and at the execution time, the generated code builds eventgraphs for detection and rules are asssociated with events.

3.1.3 Why Sentinel Uses Compilation-Based Approach?

In order to process ECA rules in Sentinel, it needs pointers to condition-action functions to call the RULE constructor. Pointers to functions(unlike pointers in data structure) cannot be persisted and used later. In sentinel, conditions and actions are functions. And in a C++ environment, the pointers to these condition and action functions are bound at compilation/Link Time and are used at runtime. Hence, by making the condition and action functions part of the application code, the above difficulty is avoided. Once the application has been developed, addition, deletion or modification of rules requires changes to thesource code, recompilation and relinking.

Hence, in Sentinel, to design an application, we first need to define schema as classes, then define methods, and then compile the application and link the application with sentinel libraries and then finally execute. Change to either schema or

methods require recompilation of the changed source code and relinking to produce a new executable.

3.2 Rule Processing in Sentinel

The Rule Processing the sentinel goes through the following steps.

- **Designing and Editing Phase:** The user designs and edits the application code. He specifies the events and rules for the application, along with the Class definitions for

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

- Class-level event specified in a reactive class definition :

  class STOCK: public *REACTIVE*

  {

  ...

  event begin(e1) int buy_stock(int number);

  ...

  }

  The event specification in the above example is transformed as,

  PRIMITIVE *STOCK_e1 = new PRIMITIVE("STOCK_e1", "STOCK",
  "begin", "int buy_stock(int number)");

- Class-level event specified not in a reactive class definition :

  event begin(e2) && end(e3) void STOCK::set_price(float price);

  The above example is transformed into two primitive events as,

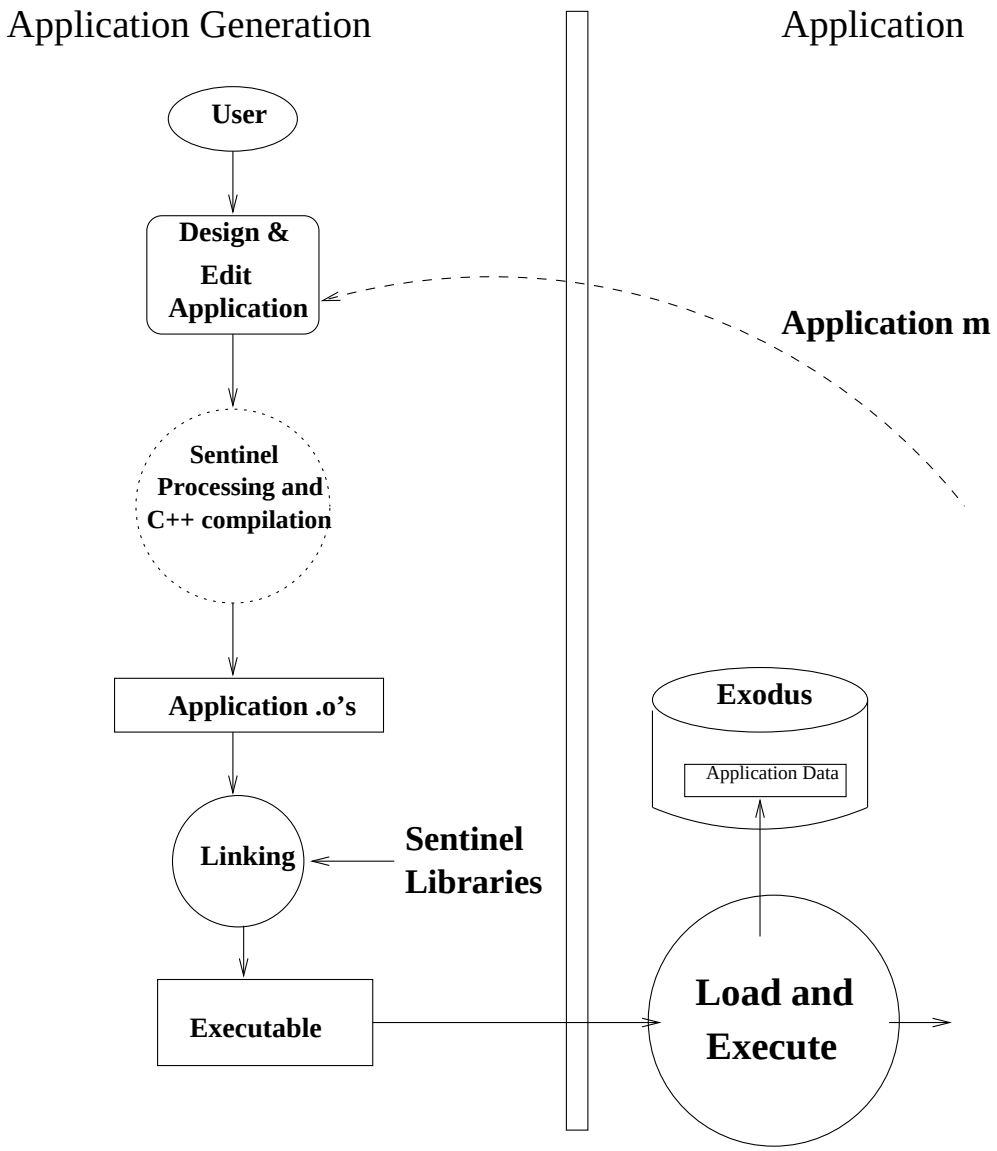


Figure 3.1. Rule Processing in Sentinel

PRIMITIVE *e2 = new PRIMITIVE("e2", "STOCK", "begin",
"void set_price(float price)");
PRIMITIVE *e3 = new PRIMITIVE("e3", "STOCK", "end",
"void set_price(float price)");

- Instance-level event :

  An instance-level event can be specified only outside of a reactive class definition.

  ...

  STOCK *IBM, *DEC;

  ...

  event end(e4:IBM) int STOCK::sell_stock(int number);

  ...

  The instance name should be specified as well as the class name.

  It is transformed as,

  PRIMITIVE *e4 = new PRIMITIVE("e4", IBM, "end",
  "int sell_stock(int number)");

3.2.1 Composite Event

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

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

event e9 = (e3 | e1) ^ e2 ;

The two examples are transformed each as,

A_star *e8 = new A_star(new NOT(e1,e2,e3), e2, new A(e4, e5, new AND(e6,e7)));

AND *STOCK_e9 = new AND(new OR(STOCK_e3,STOCK_e1), STOCK_e2);

Primitive event

If the above event is specified in the reactive class named STOCK, it is transformed as,

PRIMITIVE *STOCK_rel1 = new PRIMITIVE("STOCK_rel1", "TEMPORAL",
"", "1 hr");

P *STOCK_e10 = new P(STOCK_e1, STOCK_rel1, STOCK_e2);

The name "rel1" is given by the preprocessor. The number (here, it is 1) starts from 1 and i ncreases by 1 in every reactive class. If the above example is specified in an application pro gram, then it is converted as,

PRIMITIVE *rel1 = new PRIMITIVE("rel1", "TEMPORAL", "",
"1 hr");

P *e10 = new P(e1, rel1, e2);

The number in the name "rel1" starts from 1 at the beginning of the application program and increases by 1.

Absolute temporal event

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

If the above absolute temporal event is specified in a reactive class definition, where the cl ass name is "STOCK", it is converted as,

PRIMITIVE *STOCK_e11 = new PRIMITIVE("STOCK_e5", "TEMPORAL",
"", "14:25:00/04/23/96");

If it is specified outside of the class definition, it is transformed as,

PRIMITIVE *e11 = new PRIMITIVE("e5", "TEMPORAL", "",
"14:25:00/04/23/96");

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

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

PRIMITIVE *abs1 = new PRIMITIVE("abs1", "TEMPORAL",
"", "00:00:00/01/01/96");

3.2.2 Rule

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

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

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

RULE *r1 = new RULE("r1", STOCK_e1, check_price, set_price, RECENT);

r1->set_mode(IMMEDIATE);

r1->set_parameter(NOW);

r1->set_priority(10); The regular C++ codes are passed through the Snoop preprocessing without any modifications. The preprocessor also inserts Sentinel-related codes in the application program to make it easy for the user to use the Sentinel local event detector without worrying about details.

3.3 Postprocessing and Integrating into Open OODB Preprocessor

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

3.4 Example of Preprocessing

Original program

```
class STOCK : public REACTIVE
{
    private:
    ......
    public:
    ......
    event end(e1) int sell_stock(int qty);
    event begin(e2) && end(e3) void set_price(float price);
```

```
        int get_price();
        event e4 = e1 ^ e2; /* AND operator */
        /* class-level rules */
        rule R1[e4, cond1, action1, CUMULATIVE, DEFERRED];
};
int STOCK::sell_stock(int qty) { ..... }
void STOCK::set_price(float price) { ..... }
int STOCK::get_price() { ..... }
/* Main program */
STOCK IBM, DEC, Microsoft;
main()
{
        .....
        /* Creating instance-level primitive event */
        event begin(instance_set_price:IBM) void STOCK::set_price(float price);
        /* SEQUENCE operator */
        event seq_event = STOCK_e4 >> instance_set_price;
        /* Creating class-level primitive event */
        event begin(sell_stock) void STOCK::sell_stock(int qty);
        /* Creating class-level P event */
        event p_event = P([00:00:00/01/01/96], [7 days], [00:00:00/12/31/96]);
        /* Creating a rule which contains both class-level
        and instance-level events */
        rule R2[seq_event, cond2, action2,20, PREVIOUS];
        /* Creating a class-level rule */
        rule R3[p_event, cond3, action3, RECENT];
```

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

Preprocessed program

```
class STOCK : public REACTIVE
{
      private:
      .............
      public:
      .............
      int sell_stock(int qty);
      void set_price(float price);
      int get_price();
};
/* Main program */
STOCK IBM, DEC, Microsoft;
LOCAL_EVENT_DETECTOR *Event_detector;
void init_func();
main()
{
```

```
.....
/* Creating the local event detector */
Event_detector = new LOCAL_EVENT_DETECTOR();
init_func();
/* Creating primitive events */
PRIMITIVE *STOCK_e1 = new PRIMITIVE("STOCK_e1", "STOCK"
                                   "end", "int sell_stock(int qty)");
PRIMITIVE *STOCK_e2 = new PRIMITIVE("STOCK_e2"  "STOCK",
                                   "begin", "void set_price(float price)");
PRIMITIVE *STOCK_e3 = new PRIMITIVE("STOCK_e3", "STOCK",
                                   "end", "void set_price(float price)");
/* Creating Rule R1 */
RULE *R1 = new RULE("R1", STOCK_e4, cond1, action1, CUMULATIVE);
R1->set_mode(DEFERRED);
/* Creating instance-level primitive event */
PRIMITIVE *instance_set_price = new PRIMITIVE("instance_set_price",
                                        IBM, "begin", "void set_price(float price)");
/* Creating Rule R2 */
RULE *R2 = new RULE("R2", seq_event, cond2, action2);
R2->set_priority(20);
R2->set_trigger_mode(PREVIOUS);
/* Creating Rule R2 */
RULE *R3 = new RULE("R3", p_event, cond3, action3, RECENT);
Notify(NULL, "OODB", "beginT", "begin", system_list);
OpenOODB->beginTransaction();
Notify(NULL, "OODB", "beginT", "end", system_list);
```

```
        IBM.set_price(115.00);
        DEC.set_price(100.00);
        Microsoft.sell_stock(200);
        DEC.get_price();
        IBM.set_price(75.95);
    Notify(NULL, "OODB", "commitT", "begin", system_list);
    OpenOODB->commit();
    Notify(NULL, "OODB", "commitT", "end", system_list);
}
```

Open OODB preprocessed program

```
class STOCK : public virtual _Wrapper, public REACTIVE
{
    private:
    .....
    public:
    .....
    int sell_stock_OOdbFn(int qty);
    int sell_stock(int __1ooAgr0);
    void set_price_OOdbFn(float price);
    void set_price(float __1ooAgr0);
    int get_price_OOdbFn();
    int get_price();
};
int STOCK::sell_stock_OOdbFn(int qty)
{
    /* original sell_stock method */
```

```
}
int STOCK::sell_stock(int __1ooArg0)
{
    .... Open OODB code ...
    /* Parameters are collected in a linked list */
    PARA_LIST *sell_stock_list = new PARA_LIST();
    sell_stock_list->insert("qty", INT, __1ooArg0);
    if is_begin_of_this_subscribed_
       /* Notify begin of method */
       Notify(this, "STOCK", "int sell_stock(int qty)",
              "begin",sell_stock_list);
    /* The original sell stock method is invoked here */
    int ret_value = sell_stock_OOddbFn(__1ooArg0);
    /* Only if this event is subscribed */
    if is_end_of_this_subscribed
       /* Notify end of method */
       Notify(this, "STOCK", "int sell_stock(int qty)",
                                  "end",sell_stock_list);
    return(ret_value);
}
void STOCK::set_price_OOdbFn(float price)
{
    /* original set_price method */
}
void STOCK::set_price(float __1ooArg0)
{
```

```
.... Open OODB code ...

/* Parameters are collected in a linked list */
PARA_LIST *set_price_list = new PARA_LIST();
set_price_list->insert("price", FLOAT,__1ooArg0);
if is_begin_of_this_subscribed
   /* Notify begin of method */
   Notify(this, "STOCK", "void set_price(float price)",
                                "begin",set_price_list);
/* The original set price method is invoked here */
set_price_OOdbFn(__1ooArg0);
if is_end_of_this_subscribed
   /* Notify end of method */
}
```

This example illustrates the use of class-level and instance-level events and rules and also shows the wrapping of the methods with a collection of parameters, which are done by the Open OODB preprocessor. A class-level composite event e4 is defined which is an AND of e1 and e2. A class-level rule R1 is defined on event e4. Instance-level primitive event *set_IBM_price* is defined for STOCK object IBM. A composite sequence event is defined which is a combination of an instance-level and class-level event and finally rule R2 is defined on the sequence event(seq_event). A periodic event *p_event* is defined with absolute time interval. R3 subscribes p_event. Notice that after preprocessing the user-defined methods 'sell_stock', 'set_price', and 'get_price' are renamed as 'sell_stock_OOdbFn', 'set_price_OOdbFn' and 'get_price_OOdbFn', and wrapper methods 'sell_stock','set_price' and 'get_price' are introduced. Currently

'get_price' is not subscribed by any of event of rule, but the reason why it is also wrapped is that we allow the user to create the r ules subscribing the event later through the rule editor. As seen from the example, appropriate code is introduced in the wrapper methods to notify the events. Also the processing of the application - level rule and event specification procues appropriate code for generation of event and rule objects along with the relevant parameters.

All the proprocessed code is then compiled by the C++ compiler, and is linked to form an executable. We run the executable and it goes and retrieves the application data from Exodus and processes the rules accordingly.

CHAPTER 4
ARCHITECTURAL DETAILS

4.1 Difference Between Application Rules and External Rules

Rules can be classified into two types. They are Application-level rules and External rules. Application level rules are based on events within the application(i.e, rules which are hardcoded in the application). External rules are based on existing/potential events which are added from outside the application.

The problem here is that sentinel uses an compiation-based approach. In sentinel, to design an application, we first need to define schema as classes, then define methods, and then compile the application and link the application with sentinel libraries and then finally execute. Change to either schema or methods require recompilation of the changed source code and relinking to produce a new executable.

Hence, we thought that DReSS would be an effective way to add rules from outside the application.

4.2 Interface

The interface has been developed keeping the user in mind. The whole demo is divided into five windows.

- Dynamic Rule Editor: This window displays all the existing applications. The user is asked to select an application for which he wants to browse, insert, modify or delete rules.

- Browse Rules: This window displays all the existing events for that particular application. The user is provided with three options. He can either select Insert, Modify or Delete Rules. Also the user can see all the existing primitive

events for that particular application and if he double clicks on that particular event, he can see all the rules associted with that event.

- Insert Rules: This window allows you to enter all the necessary information to create a rule. The user has to give the following information:
  - METHOD/EVENT NAME: The user can choose any of the existing primitive events or methods from the lsitbox provide.
  - RULE NAME: The user has to enter the rule name.
  - CONTEXT: It can be one of these four options: RECENT, CHRONICLE, CONTINOUS, or CUMULATIVE.
  - COUPLING: Coupling mode defines the mode of the rule execution. currently IMMEDIATE and DEFFERED coupling modes are supported.
  - PRIORITY: It can be any of the interger values from 1 to 50
  - TRIGGER MODE: Two options NOW(start detecting all constituent events starting from this time instant.) and PREVOIUS(all constituent events are acceptable) are supported as rule triggering modes with NOW being the default.
  - CONDITION FUNCTIONS: The code for the condition function will be processed and compiled into a .o file
  - ACTION FUNCTIONS: The code for the action function will be processed and compiled into a .o file

  After giving all the information, he can either save the rule or just ignore it.

- Delete Rules: The user is displayed all the rules that he can delete. He has to double click it to get all the details for that particular rule. He has the option to either delete the rule or just ingone it.

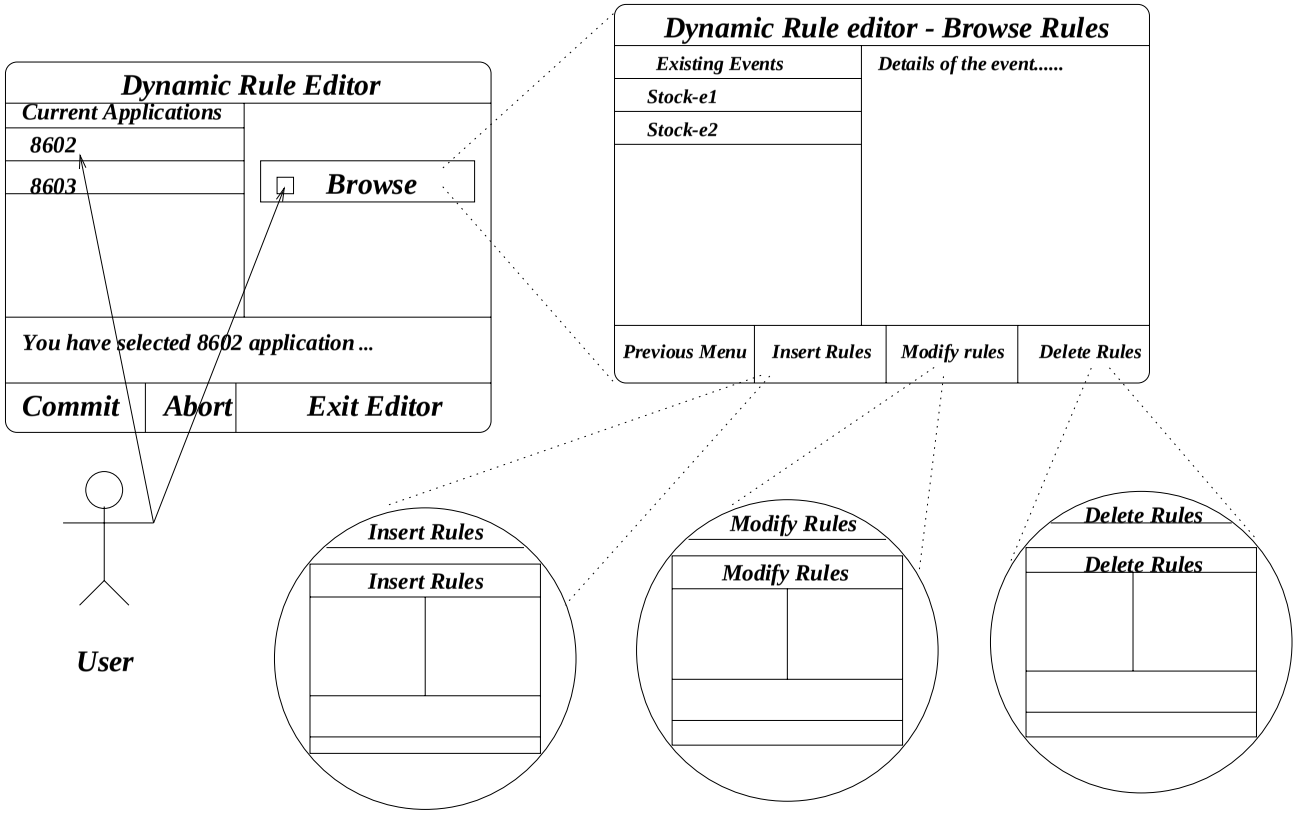


Figure 4.1 RuleEditorDemo

- Modify Rules: The user is displayed all the rules that he can modify for a particular event. He has to double click it to get all the details for that particular rule. He has the option to either modify the rule or just ignore it.

4.3 Exodus

4.3.1 Introduction

The EXODUS Storage Manager is a multi-user object storage system supporting versions, indexes, single-site transactions, distributed transactions, concurrency control and recovery.

4.3.2 Architecture

The EXODUS Storage Manager 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 Manager 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. The server stores all data on volumes which are either Unix files or raw disk partitions.

4.3.3 Facilities

The EXODUS Storage Manager provides objects for storing data, versions of objects, files for grouping related objects, and indexes for supporting efficient object access. The Storage Manager also provides volumes, transactions, concurrency control, receovery and configuration options.

CHAPTER 5
DESIGN OF DYNAMIC RULE EDITOR

5.1 Introduction

Rules in the context of an active DBMS consist primarily of three components: an event, a condition and an action. An event is an indicator of a happening which can be either primitive or composite. The condition can be a simple or a comple query on the current database state or on the previous and the current states, or even on historical data. Actions specify the operations to be performed when an event has occured and the condition evaluates to true.

There are two types of rules: Application level Rules and Rules Added Dynamically. In first, the rules which are based on events within the appliation(rules added along with the application).In second, the rules are added from outside the application without editing the application source code.

Sentinel uses compilation-based environment. In sentinel, to design an application, we first need to define schema as classes, then define methods, and then compile the application and link the application with sentinel libraries and then finally execute. Change to either schema or methods require recompilation of the changed source code and relinking to produce a new executable.

The problem here in using the compilation-based environment is that Pointers to functions(unlike pointers in data structure) cannot be persisted and used later. In sentinel, conditions and actions are functions. And in a C++ environment, the pointers to these condition and action functions are bound at compilation/Link Time and are used at runtime. Hence, by making the condition and action functions, part of the application code, the above difficulty is avoided. Once, the application has

been developed, addition, deletion or modification of rules requires changes to the source code, recompilation and relinking.

In DReSS, function pointers for the condition and action functions were needed. The problem was overcome by maintaining a static file which stores an array of structures

```
-----------------------------------------------------
con-fun.h (It is the header file for con-fun-pointer.c)
#include "Sentinel.h"
#include <stdio.h>

struct cond_key_ptr {
        char cond_key[400];
        int (*cond_ptr) (L_OF_L_LIST *);
};
extern int cond1(L_OF_L_LIST *);


This function stores the condition/action key and its
pointer.
-----------------------------------------------------
con-fun-pointer.c(The file which stores all the keys and its
corresponding condition action pointers.

#include "con-fun.h"
struct cond_key_ptr key_ptr[300] = {
,{"/cis/database15/sentinel/OpenOODB.1.1/Sentinel0.9/
```

```
Rule_Catalog/juice8600/8608/cond_function/condstock.C",
(int (*)(L_OF_L_LIST *))condstock}
};

This is the actual data structure
-----------------------------------------------------------
```

This static file is compiled and its .o is linked with the application. At run-time, it traverses the rule_catlogue linked list that we persisted and gets the key for that particular condition or action. After acquiring the key, it travereses the array of structures and obtains the corresponding condition/action pointer.

5.2 Steps Followed by the Application Level Rules

The application goes through the following procedure:

- The application source code goes though a set of three pre-processors SPP, PPCC, CPP.
- Then the application .o's are made
- Then they are linked to form an executable.

The drawback of the above procedure is that, all the rules are defined inside an application.

5.3 Example of How to Create Rules Through DRed

- Use the user-friendly Dred interface to create the rules..
- Store the rule_catlogue in EXODUS.
- Create .o's for condition/action files by using CPP pre-processor.
- Create condition/action libraries.

- Linking those libraries with the application.
- Create a new executable.

Because of the simplicity of the above design, we thought that DReSS is an effective way of adding rules to an application.

CHAPTER 6
IMPLEMENTATION OF DYNAMIC RULE EDITOR

This chapter discusses about the implementation of Dynamic Rule Editor including the datastructures used to store the information, and insert, modify and delete in the RuleEditor frontend.

The ruleeditor data structure that we maintain in Exodus is a three-level linked list. on the top level, we have all Application name and the storage group. In the next level we have all the events associated with a particular application. In the third level, we have all the rules associated with a particular rule.

6.1 Interface Details

6.1.1 Save/Compile for Insert Rules

The following steps are executed when we use the Save/Compile button in the Insert Rules Window.

- Adds the necessary header files and saves the condition and action-files in /tmp directory as .C files.
- Compiles the condition and action files to obtain their .o files.
- Gives all the details to the Cmd procedure for it to include in the data structure.
- The insertrule procedure checks for that particular event. if it exists , then it adds a rule to that corresponding event. else a new event is created and a rule is appended to the data structure.

6.1.2 Save/Compile for Modify Rules

The following steps are executed when we use the Save/Compile button in the Modify Rules Window.

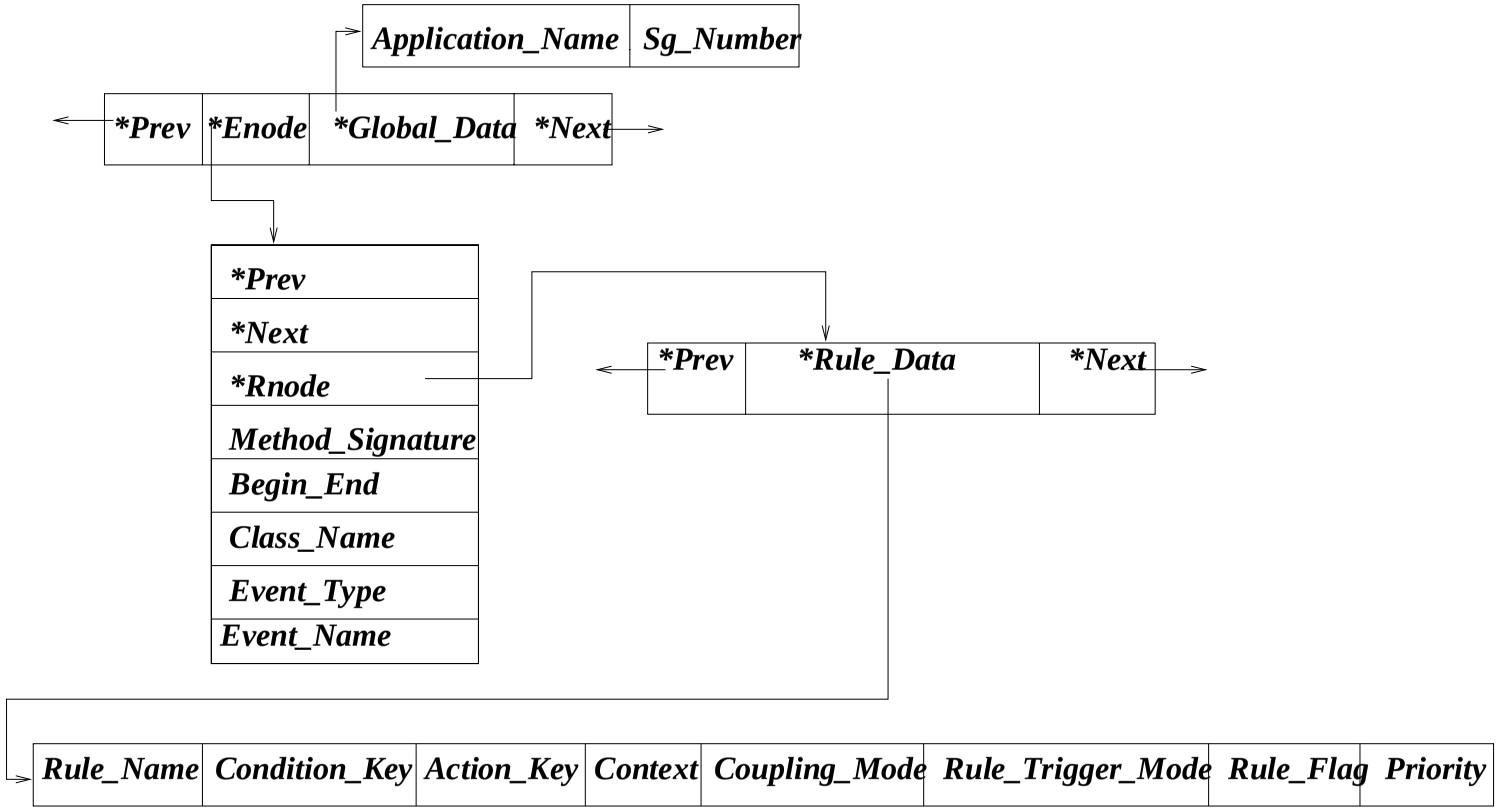


Figure 6.1. RuleEditor Data Structures

- Adds the necessary header files and saves the condition and action-files in /tmp directory as .C files.
- Compiles the condition and action files to obtain their .o files.
- Gives all the details to the Cmd procedure for it to include in the data structure.
- The modifyrules procedures updates the details of the rule in the right place.

6.1.3 Save/Compile for Delete Rules

- It goes and deletes an entry for that particular rule in the data structure.
- It also maintians a list of all the entires that need to be deleted from the con-fun-pointer.c file

6.1.4 Commit/Abort

Commit

It commits all the changes made to the application. It executes the following steps.

- Adds the condition-action pointers to the file . it first checks whether an entry exists for that particular condition/action in that file. if it already exist then it doesnt add it, else it adds it.
- Checks whether all the condition/action files that exist in the directory, has a corresponding condition/action key entry in the data structure. if it doesn't exist then it deletes that file and its corresponding .o from that directory
- Compiles the condition-action function pointers file.
- Creates the condition and action libraries so that they can be linked by the application and the rules can be executed.
- Persists the whole data structure in EXODUS.

Abort

It aborts all the changes made to the application until then and quits the Dynamic Rule Editor.

6.2 Implementation of The back-end for the Interface

6.2.1 Implementation of Insert Rule

This routine first checks whether the rule has been added to an existing event. If yes, then it just adds a rule node to that event node, else, it creates a new event node and adds arule node for that particular event node.

6.2.2 Implementation of Modify Rule

This routine updates the values in the existing rule node data structure.

6.2.3 Implementation of Delete Rule

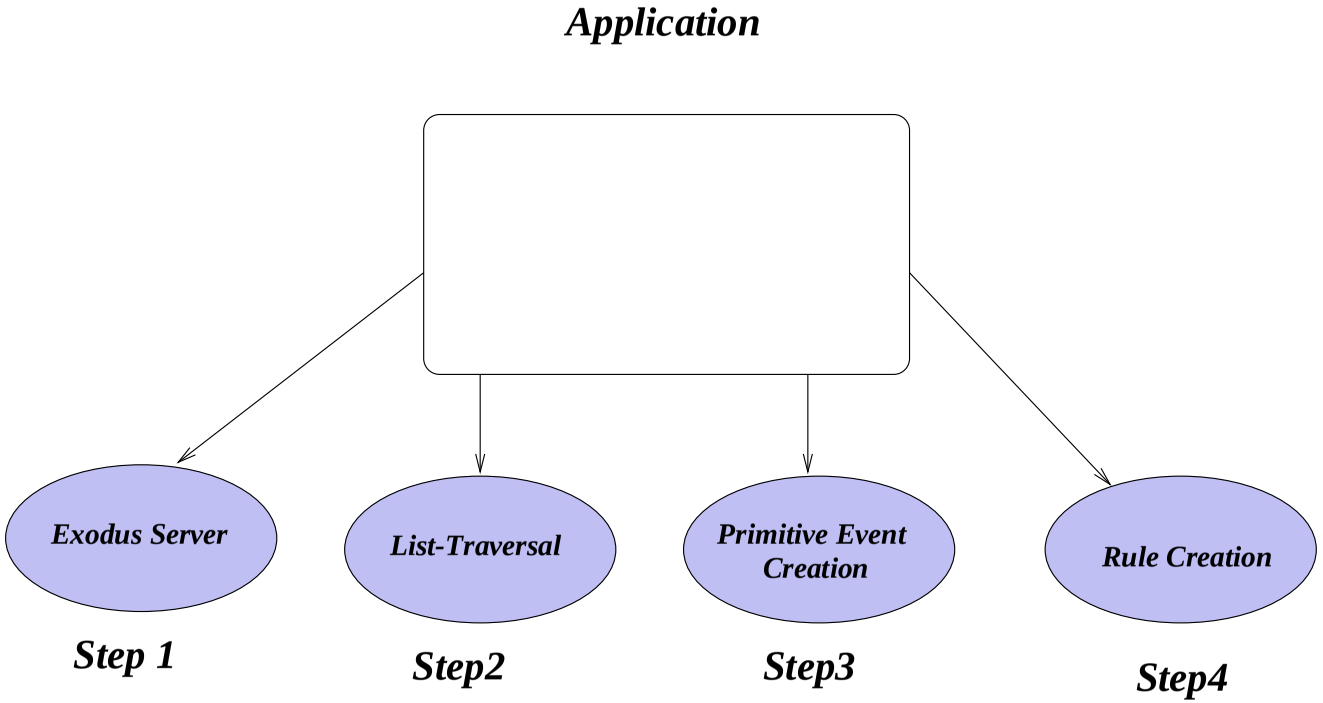
This routine deletes the rule node. If it is the only/last rule for that event, then it deletes the event node along with the rule node.

6.3 Run-Time Details

As soon as the application is started, it first executes the load_dynamic_rules procedure. In that procedure it executes the following steps:

- It first retrieves the Rule_catalogue data structure from EXODUS.
- Traverse the Rule_Catalogue data structure and get the keys for the condition/action pointers. Then with the key it traverses the con-fun-pointer.c file and if the key matches, it retrieves the corresponding condition/action pointer.

```
------------------------------------------------------------
con-fun.h (It is the header file for con-fun-pointer.c)
#include "Sentinel.h"
#include <stdio.h>
```



Step 1 : Retrieves the list from the EXODUS server

Step2 : Traverses the list and acquires the keys for condition-action pointers.

Step3 : Creates a PRIMITIVE event.

Step4 : Creates a RULE.

Figure 6.2. Runtime execution of Program

```
struct cond_key_ptr {
        char cond_key[400];
        int (*cond_ptr) (L_OF_L_LIST *);
};
extern int cond1(L_OF_L_LIST *);
```

{\bf Explanation:}
The structure cond_key_ptr has two members. one
of char type which stores the structure and another
of int type which stores the condition and action
function pointer.

---

con-fun-pointer.c(The file which stores all the keys
and its corresponding condition action pointers.

```
#include "con-fun.h"
struct cond_key_ptr key_ptr[300] = {
,{"/cis/database15/sentinel/OpenOODB.1.1/
Sentinel0.9/Rule_Catalog/uice8600/8608/cond_function
/condstock.C",(int (*)(L_OF_L_LIST *))condstock}
};
```

{\bf Explanation:}
Here we populate the data structure with their keys
and condition/action pointers. The assumption here is that
the right brace should be in the

```
next line so that we can edit the file.
-------------------------------------------------------
```

- An instance of the event is created by calling the EVENT constructor with the following parameters.

```
PRIMITIVE *new_event = new PRIMITIVE("new_event",
tmp_event_ptr->dat->class_name,tmp_evnt_ptr->dat->begin_end,
,char_new_signature);
```

- RULE is created by calling the RULE constructor with the following parameters. The action function method is executed when the condition is met.

```
RULE *new_rule = new RULE("new_rule", new_event,
key_ptr[condition_ptr].cond_ptr,
(void (*) (L_OF_L_LIST *))key_ptr[action_ptr].cond_ptr,
RECENT);
```

CHAPTER 7
CONCLUSION AND FUTURE WORK

7.1 Creation of ECA rules in Conventional way and using RuleEditor

This thesis extends the sentinel system to support dynamic rules. we divided the whole work into two modules . First, Maintaining the Dynamic Rule Editor User Interface, and keeping track of all the the data structures along with the condition and action libraries, Second, the retrieval of the data structure and calling the RULE and EVENT constructor, to add a node in the event graph.

The diagram below, clearly outlines the creation of ECA rules in both the Conventional Way and also using the Dynamic Rule Editor.

Presently, Dynamic rule editor(Sentinel system), supports primitive events, and it can also be extended to support composite events.

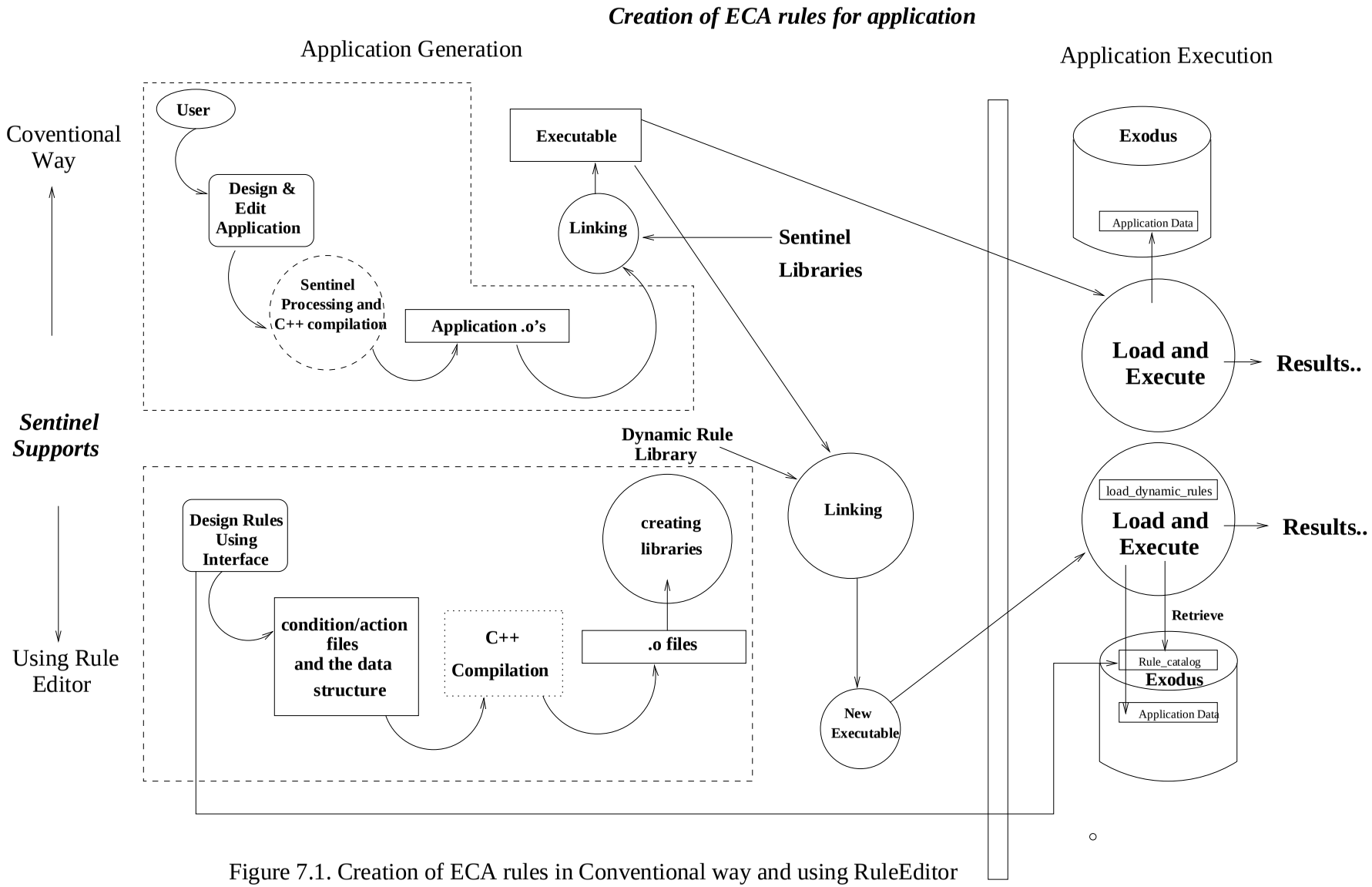


Figure 7.1. Creation of ECA rules in Conventional way and using RuleEditor

The Global_Data data structure stores information about each application existing. It is in the format below and is stored in the EXODUS storage manager.

```
struct Global_Data {
int storage_group_number; // Storage-group number in which
                          // the details of aplication exist.
char  Application_name[VARIABLE_LENGTH]; //Application_name.
};
```

The Event_Data data structure stores information about all the existing primitive events for a particular application.

```
struct Event_Data {
char  method_signature[VARIABLE_LENGTH]; //method_signature.
char  begin_end[VARIABLE_LENGTH]; //begin event or end event.
char  class_name[VARIABLE_LENGTH]; //the event's class name.
char  event_type[VARIABLE_LENGTH]; //PRIMITIVE event
                        // or COMPOSITE event.
char  event_name[VARIABLE_LENGTH]; //The name of the event.
};
```

The Rule_Data data structure stores information about all the rules asscocitaed with a particular event.

```
struct Rule_Data {
```

```
char  rule_name[VARIABLE_LENGTH]; //Name of the rule
char  condition_key[VARIABLE_LENGTH];//condition key string
      //to retrieve its pointer from con-fun-pointer.c
char  action_key[VARIABLE_LENGTH];//action key string to
      //to retrieve its  pointer from con-fun-pointer.c
char  context[VARIABLE_LENGTH];//context of the rule
char  coupling_mode[VARIABLE_LENGTH];//coupling of the rule
char  rule_trigger_mode[VARIABLE_LENGTH];//The mode in which
      // the rule has to get executed
char  rule_flag[VARIABLE_LENGTH];//rule_flag
char  priority[VARIABLE_LENGTH];//the priority in which it
      //should be executed.
};
```

The Rnode struture is a doubly-linked list with pointers to its previous node, its next node and a pointer to Rule_Data data structure.

```
struct Rnode {
int type;
struct Rnode *prev;  // points to previous node
struct Rnode *next;  // points to next node
struct Rule_Data  *dat;    // points to Rule_Data data structure
};
struct Rnode *rule_curr, *rule_tail;//tail, and current
```

The Enode struture is a doubly-linked list with pointers to its previous node, its next node and a pointer to Event_Data data structure.

```
struct Enode {
int type;
struct Enode *prev;  // points to previous node
struct Enode *next;  // points to next node
struct Event_Data *dat;    // points to Event_Data data structure
struct Rnode *rule;
};
struct Enode *evnt_curr, *evnt_tail;// tail and current
```

The Gnode structure is a doubly-linked list with pointers to its previous node, its next node and a pointer to Global_Data data structure.

```
struct Gnode {
int type;     // 1:integer, 2:float, 3:char*, 4:char, 5:class1...
struct Gnode *prev;  // points to previous node
struct Gnode *next;  // points to next node
struct Global_Data  *dat;// points to Global_Data data structure
struct Enode *event;
};
```

The Glist class is used to store and retrieve the applications data structure in Exodus storage manager.

```
class Glist {
public:
 Glist();
~Glist();
```

```
void Insertafter(user_class*, int, Glist*);
void Insertrule(Gnode*, char*, char*, char*, char*,
 char*, char*, char*, char*, char*, char*, char*, char*, char*);
void traverse(); //This is to return the number of applications
                // under the host and port

Gnode* traverse_application(int); //This is to return
// the pointer for a  particular application
void traverse_linked_list(char *);//This is to display
      // rall the events and rules for a particular applcation
void traverse_events(Gnode* );
void traverse_event_rule(Gnode*, char*, char*);
//To get all rules associated with a event

Rnode* traverse_rules(Gnode*, char*, char*);

void Modify_rules(Rnode*, char*, char*, char*, char*, char*, char*);

int First();
int Last();
int Prev();
int Next();
Gptr Retrieve();
void Delete(Rnode *);
void glprint();
void glprint2();
```

```
void glprint3(Gnode *);

void glprint4();

void update_cond_action(int);


//private:


struct Gnode *head, *tail, *curr;


};
```

DYNAMIC RULE EDITOR USER MANUAL

Note: The bracketed notation (e.g <BROWSE>) in this manual represents an option on the screen to be selected. Click<BROWSE> means to click that selection with the LEFT button of the mouse unless otherwise specified.

The Dynamic Rule Editor is Designed to aid a rule designer to graphically Insert, Modify, Delete and Browse rules. In this User Manual we will go through the windows used to insert, modify, delete, and browse rules for an existing set of applications. Invoke a session by typing: **RuleEditorDemo** You are now at the top-level menu of the Dynamic Rule Editor. This gives you a list of all the current applications on which you can insert, modify, delete and update the rules. <Double-click> on the application for which you want to browse. Then Click<BROWSE>

Browse Rules: Now You are in the browsing window of the dynamic rule editor. Double-click on the event to select and view rules for that particular event. You have three options in browse events. 1. To Add rules for that particular event. For this click<Insert Rules>. 2. To Modify the existing rules for a particular event. For this option, you have to make sure that you select an event before you click<Modify Rules>. 3. To Delete the existing rules for a particular event. Again for this option, you have to make sure that you select an event before you click<Delete Rules>.

Insert Rules: Insert Rules allows you to add rules to a particular event. To add rules, first you have to select an event for which you want to add rules. If you donot select an event then the event selected in the browsing window of the dynamic rule editor is taken as default. **Rule Name:** An entry widget is provided to enter the name of the rule. **Context:** Click and HOLD DOWN the LEFT mouse button over the <Context> button and you will get four types of context's to select(CUMULATIVE,

RECENT, CHRONICLE) and release it over the one which you want to select. **Coupling:** Click and HOLD DOWN the LEFT mouse button over the <Coupling> button and you will get three types of coupling modes to select(DEFFERED, DETACHED, IMMEDIATE) and release it over the one which you want to select. **Priority:** An entry widget is provided to enter the priority value. **TriggerMode** Click and HOLD DOWN the LEFT mouse button over the <TriggerMode> button and you will get two types of trigger modes to select(PREVIOUS, NOW) and release it over the one which you want to select. **Condition functions:** Click and HOLD DOWN the LEFT mouse button over the <Condition functions> button and you will get two options to select(WANT TO USE EXISTIING CONDITIONS?.., CREATE A NEW CONDITION) and release it over the one which you want to select. If you select the first option, then it will display the list of all the files that can be selected,.. and release it over the one which you want to select. If you select the second option then it will pop up a window to accept the name of the condition file and after you enter the filename in the entry widget, click<OK>. Once you give the file name it displays the body of the file inside the box beneath it and will allow you to edit the code. **Action functions:** Click and HOLD DOWN the LEFT mouse button over the <Action functions> button and you will get two options to select(WANT TO USE EXISTIING ACTIONS?.., CREATE A NEW ACTION) and release it over the one which you want to select. If you select the first option, then it will display the list of all the files that can be selected,.. and release it over the one which you want to select. If you select the second option then it will pop up a window to accept the name of the action file and after you enter the filename in the entry widget, click<OK>. Once you give the file name it displays the body of the file inside the box beneath it and will allow you to edit the code. After you have finished entering the Rule Details, you have three options. 1. **Save/Compile:** This will save the

rule in the data structure but will not commit the changes onto the Exodus server. click<Save/Compile> to select this. 2. **Previous Menu:** This will take you back to the previously invocated men. click<Previous Menu> to select this 3. **Exit Editor:** This will pop up a window to confirm whether you really want to exit the dynamic rule editor. If Yes, then click<Yes>

Modify Rules: Modify Rules allows you to modify rules to a particular event.The event select ed in the browsing window of the dynamic rule editor is taken as default.

Once you select a rule, it will display the existing details of that particular rule. You also have the option of changing the rule details by doing the following: **Rule Name:** An entry widget is provided to enter the name of the rule. **Context:** Click and HOLD DOWN the LEFT mouse button over the <Context> button and you will get four types of context's to select(CUMULATIVE, RECENT, CHRONICLE) and release it over the one which you want to select. **Coupling:** Click and HOLD DOWN the LEFT mouse button over the <Coupling> button and you will get three types of coupling modes to select(DEFFERED, DETACHED, IMMEDIATE) and release it over the one which you want to select. **Priority:** An entry widget is provided to enter the priority value. **TriggerMode** Click and HOLD DOWN the LEFT mouse button over the <TriggerMode> button and you will get two types of trigger modes to select(PREVIOUS, NOW) and release it over the one which you want to select.

Condition functions: Click and HOLD DOWN the LEFT mouse button over the <Condition functions> button and you will get two options to select(WANT TO USE EXISTIING CONDITIONS?.., CREATE A NEW CONDITION) and release it over the one which you want to select. If you select the firs option, then it will display the list of all the files that can be selected,.. and release it over the one which you want to select. If you select the second option then it will pop up a window to accept the name of the condition file and after you enter the filename in the entry

widget, click<OK>. Once you give the file name it displays the body of the file inside the box beneat h it and will allow you to edit the code. **Action functions:** Click and HOLD DOWN the LEFT mouse button over the <Action functions > button and you will get two options to select(WANT TO USE EXISTIING ACTIONS?.., CREATE A NEW ACTION) and release it over the one which you want to select. If you select the first option, the n it will display the list of all the files that can be selected,.. and release it over the one w hich you want to select. If you select the second option then it will pop up a window to accept t he name of the action file and after you enter the filename in the entry widget, click<OK>. O nce you give the file name it displays the body of the file inside the box beneath it and will al low you to edit the code.

After you have finished changing the Rule Details, you have three options. 1. **Save/Compile:** This will save the rule in the data structure. click<Save/Compile> to select this 2. **Previous Menu:** This will take you back to the previously invocated men. click<Previ ous Menu> to select this 3. **Exit Editor:** This will pop up a window to confirm whether you really want to exit the dynamic rule editor. If Yes, then click<Yes>

Delete Rules: Delete Rules allows you to delete rules of a particular event. The event in the browsing window of the dynamic rule editor is taken as default and the rules are displayed for that particular event

Once you select a rule, it will display the existing details of that particular rule. click<Delete Rules> to delete that particular rule.

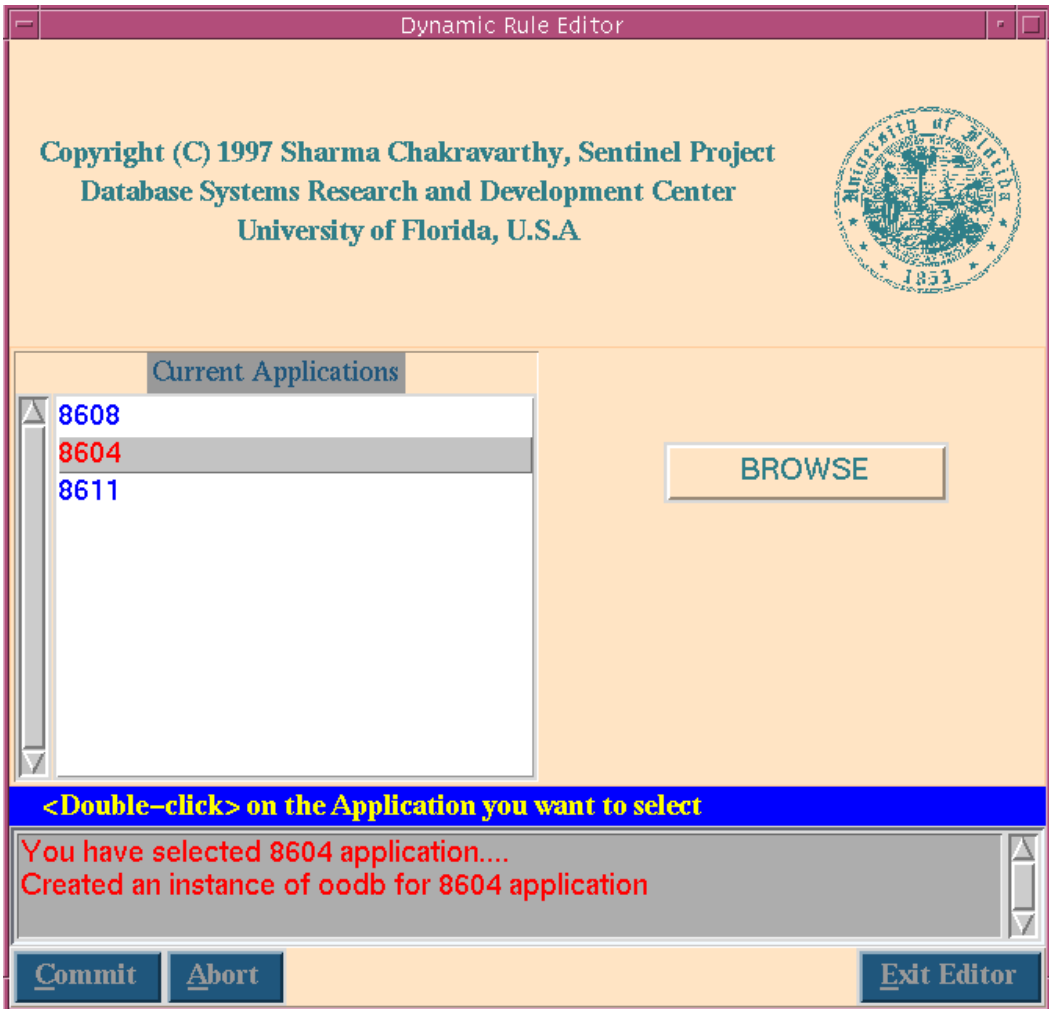


Figure 7.1. Dynamic Rule Editor

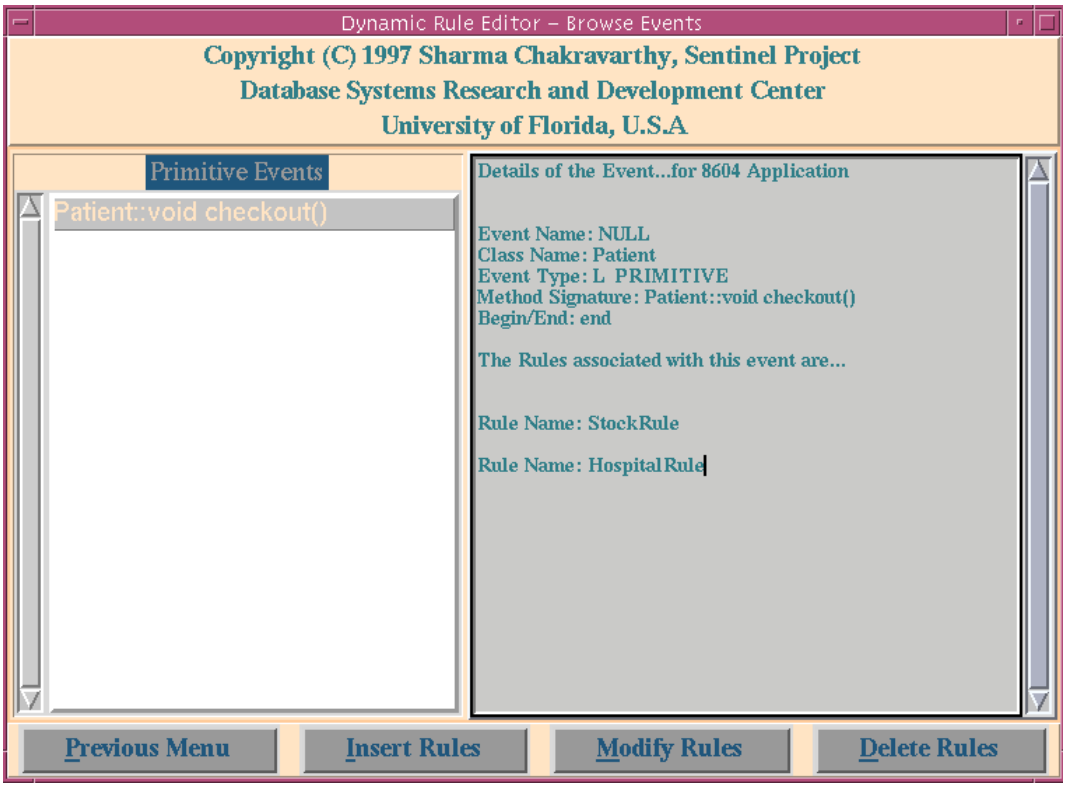


Figure 7.2. Dynamic Rule Editor - Browse Rules

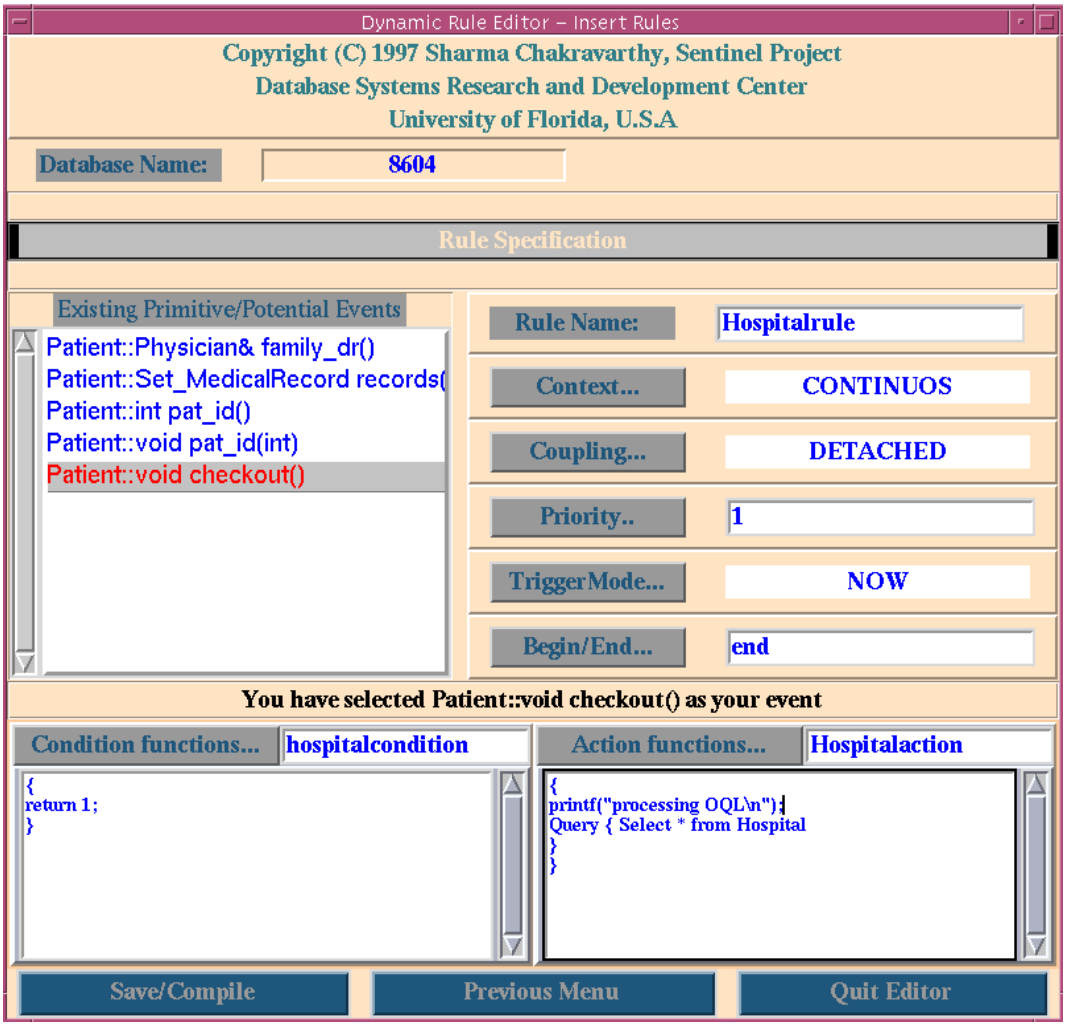


Figure 7.3. Dynamic Rule Editor - Insert Rules

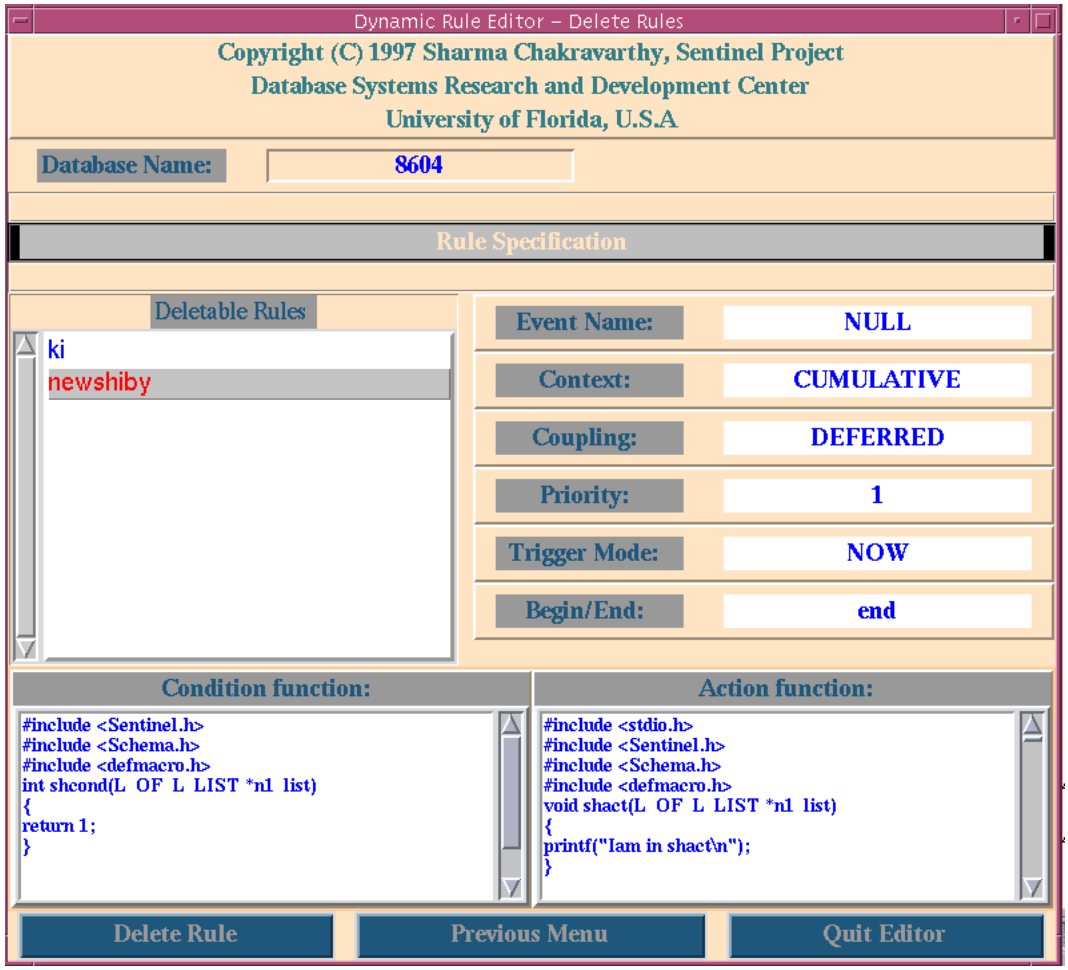


Figure 7.4. Dynamic Rule Editor - Modify Rules

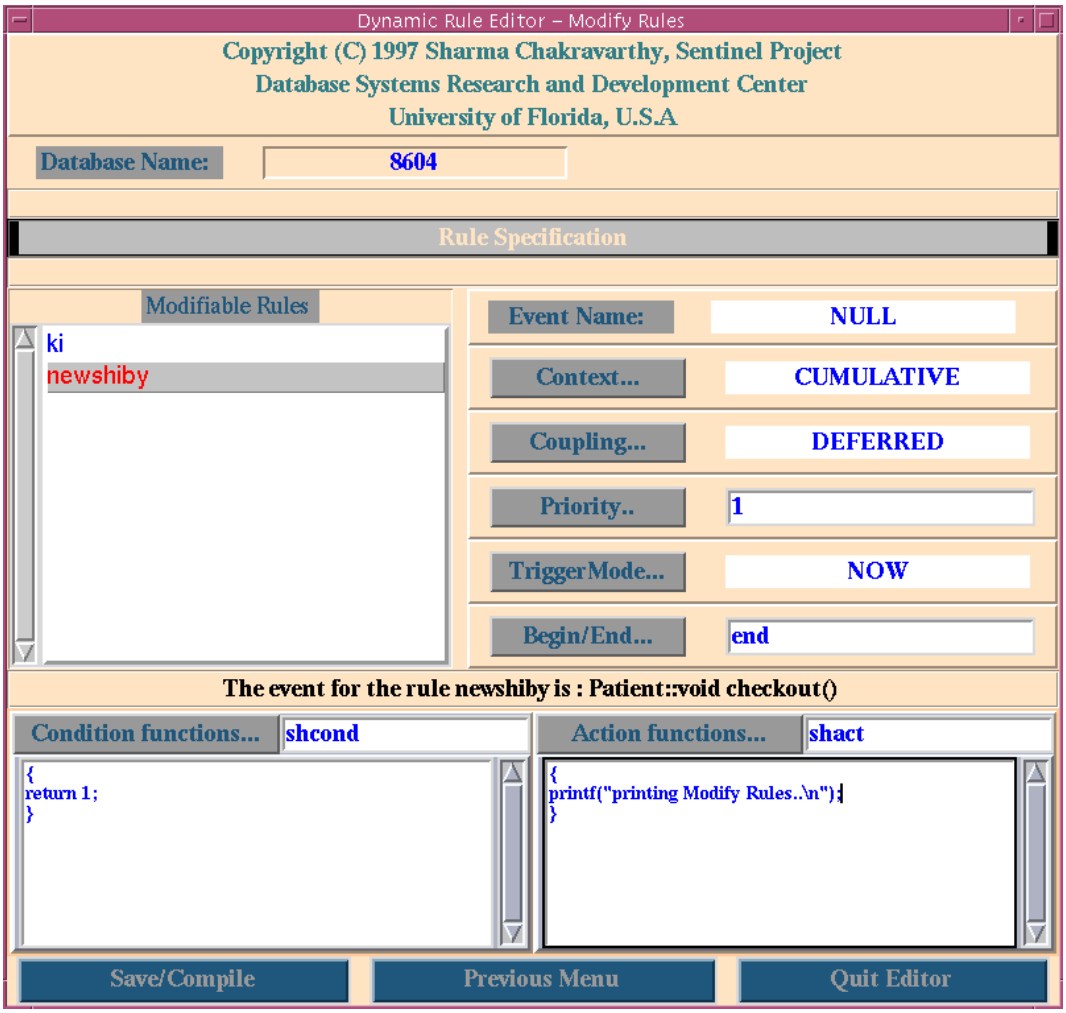


Figure 7.5. Dynamic Rule Editor - Delete Rules

REFERENCES

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

[2] V. Hyesun Lee. Support for temporal events in Sentinel: Design, implementation, and preprocessing. Master's thesis, University of Florida, Gainesville, 1996.

[3] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Anatomy of a Composite Event Detector. Technical Report UF-CIS-TR-93-039, University of Florida, Gainesville, December 1993..

[4] S. Gatziu, K.R. Dittrich. Samos: An active object-oriented database system.. *IEEE Quarterly Bulletin on Data Engineering*, March 1993.

BIOGRAPHICAL SKETCH

Prahlad Madabhushi was born on January 28, 1973, in Hyderabad, India. He received a Bachelor of Science degree from Osmania University, India in June 1993.

He joined the Department of Electrical and Computer Engineering at the University of Florida in August 1994 to pursue a master's degree.

He has worked as a research assistant in the Department of Anesthesiology, Shands Hospital, at the University as a Paradox Programmer from 1994 Fall to 1996 Spring and since then has worked in the Database Systems Research and Development Center of the Computer Information Sciences Department.

His research interests include computer networks and active database systems.