

CONSTRUCTION OF
TRIGGER AND DEPENDENCY GRAPHS USING
EVENT AND RULE DECLARATIONS OF AN
ACTIVE OBJECT-ORIENTED
DATABASE MANAGEMENT SYSTEM

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By

Ali Saman Tokun

July, 1997

THESIS
QA
76.9
.D3
767
1997

CONSTRUCTION OF
TRIGGER AND DEPENDENCY GRAPHS USING
EVENT AND RULE DECLARATIONS OF AN
ACTIVE OBJECT-ORIENTED
DATABASE MANAGEMENT SYSTEM

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By

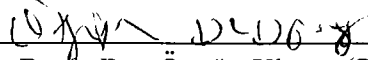
Ali Şaman Tosun

Ali Şaman Tosun
July, 1997

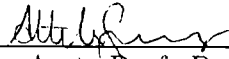
QA
76.9
-D3
T67
1997

BC38372

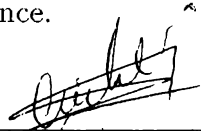
I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Asst. Prof. Dr. Özgür Ulusoy (Principal Advisor)


I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Asst. Prof. Dr. Atilla Gürsoy

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.


Asst. Prof. Dr. İlyas Çiçekli

Approved for the Institute of Engineering and Science:


Prof. Dr. Mehmet Baray Y.
Director of Institute of Engineering and Science

ABSTRACT

CONSTRUCTION OF TRIGGER AND DEPENDENCY GRAPHS USING EVENT AND RULE DECLARATIONS OF AN ACTIVE OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM

Ali Şaman Tosun

M.S. in Computer Engineering and Information Science

Supervisor: Asst. Prof. Dr. Özgür Ulusoy

July, 1997

Traditional database systems are passive, meaning that they only react to explicit requests by users or applications. An active database system on the other hand, executes operations automatically when certain events occur and certain conditions are met. A database management system becomes active through the addition of rules. The main difficulties in the development of rule applications is the lack of design methods and suitable design tools. *Confluence* and *termination* are two important properties to be able to implement rule applications correctly. In this thesis, the construction of trigger and dependency graphs using class and rule declarations of an active object-oriented database system is described. The construction of these graphs provides that termination can be checked and a confluent rule execution can be achieved. Implementation of a preprocessor that constructs trigger and dependency graphs is also provided.

Key words: Active Database Systems, Database Rule Processing, Static Analysis, Confluence, Termination.

ÖZET

AKTİF BİR VERİTABANINDA TETİKLEME VE BAĞLILIK ÇİZGELERİNİN EYLEM VE KURAL TANIMLAMALARI KULLANILARAK OLUŞTURULMASI

Ali Şaman Tosun

Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans

Tez Yöneticisi: Yrd. Doç. Dr. Özgür Ulusoy

Temmuz, 1997

Klasik veri tabanları pasiftir ve sadece kullanıcılar veya uygulamalar tarafından yapılan açık isteklere cevap verebilirler. Aktif veri tabanları ise, belli eylemler gerçekleştiğinde ve belli koşullar sağlandığında işlemleri otomatik olarak işleme koyar. Veri tabanları kuralların eklenmesi yoluyla aktif olur. Kural uygulamaları geliştirilmesindeki en büyük problem tasarım yöntemlerinin ve uygun tasarım araçlarının eksikliğidir. *Birleşme* ve *bitim* uygulamaların doğru geliştirmesi için gerekli iki önemli özelliktir. Bu tezde, tetikleme ve bağlılık çizgelerinin aktif bir veritabanının sınıf ve kural tanımlamalarından oluşturulması tartışılmaktadır. Bu çizgeler *bitim*'i kontrol edebilmemizi ve birleşme kural işlemeyi sağlar. Bu çizgeleri oluşturacak bir önışleyicinin geliştirmesi de tezimizde gerçekleştirilmiştir.

Anahtar kelimeler: Aktif Veri Tabanları, Veri Tabanlarında Kural İşleme, Statik Analiz, Birleşme, Bitim.

To my parents and my sister

ACKNOWLEDGMENTS

I am very grateful to my supervisor, Asst. Prof. Dr. Özgür Ulusoy for his invaluable guidance and motivating support during this study. His instruction will be the closest and most important reference in my future research. I would also like to thank Prof. Sharma Chakravarthy for his guidance, Jennifer Sung for her technical support, my family for their moral support and patience during the stressful moments of my work, and last but not the least, Tahsin Mertefe Kurç, who was always ready for help with his priceless technical knowledge and experience.

Finally, I would like to thank the committee members Atilla Gürsoy and İlyas Çiçekli for their valuable comments, and everybody who has in some way contributed to this study by lending moral and technical support.

Contents

1	Introduction	1
2	Problem, Difficulties and Limits	4
3	Static Analysis of Active Rules	7
3.1	Definitions	7
3.2	Static Analysis of Active Rules	9
3.3	Dependencies	10
3.4	Dependency Graph	11
3.5	Trigger Graph	11
3.6	Related Work	11
4	Trigger and Dependency Graphs	13
4.1	Sentinel	13
4.2	Construction of Trigger and Dependency Graphs	15
4.3	Event Types	19
4.4	Input Processing	20

4.5	Event List Construction	20
4.6	Condition List Construction	22
4.7	Action List Construction	23
4.8	Incremental Methods	23
4.9	Detailed Conflict Detection	23
5	Implementation	27
5.1	Data Structures	27
5.2	Number of Comparisons to detect dependencies	35
5.2.1	Number of Comparisons to Detect Action-Event Dependency	35
5.2.2	Number of Comparisons to Detect Action-Condition Dependency	37
5.2.3	Number of Comparisons to Detect Action-Action Dependency	37
5.2.4	Total Cost of Conflict Detection	37
5.3	User Interface	38
5.4	Usage of the tool	38
6	Conclusions and Future Work	41
6.1	Conclusions	41
6.2	Possible Improvements on Implementation	42
A	Sample Input Files	43

CONTENTS

ix

A.1	Original Program	43
A.2	Sample Class and CLR Declaration	45
A.2.1	Demo_employee.sh	45
A.2.2	Demo_employee.c	46
A.2.3	Demo_company.sh	47
A.2.4	Demo_company.c	48
A.3	Snoop Preprocessed File Demo.c	50

List of Figures

2.1	Syntax of Starburst Rule	5
2.2	Example of a Rule in Starburst	5
2.3	Example of a Rule Specification in Sentinel	6
4.1	BNF for Snoop	14
4.2	BNF for Snoop	15
4.3	Example of Rules	16
4.4	Example Nodes of Rulegraph	17
4.5	Rule Graph	19
4.6	Declaration of Conditions	22
4.7	Declaration of Condition Functions	22
4.8	Format of File for Incremental Methods	23
4.9	Dependency Graph Without Detailed Conflict Detection	24
4.10	Detailed Conflict Detection File Demo_company.dcg	24
4.11	Nodes Of Rulegraph For Rules comprule1, comprule2, comprule3	26
5.1	Algorithm to Construct Event Lists of Nested Events	29

5.2	Sample Run on a Nested Event	30
5.3	Function to Detect Conflicts Between an ILR and a CLR	32
5.4	Function to Detect Conflicts Between two ILR	32
5.5	Algorithm to Detect Action-Event Dependencies	33
5.6	Function to Detect Dependency Conflicts	34
5.7	Algorithm to detect dependencies	34
5.8	Select Rule Set	39
5.9	Trigger Graph	40

List of Tables

4.1 Eventlist Construction 21

5.1 Parameters Used in Determining Number of Comparisons 35

Chapter 1

Introduction

Traditional database systems are passive, meaning that they only react to explicit requests by users or applications. An active database system, on the other hand, executes operations automatically when certain events occur and certain conditions are met. A database management system (DBMS) becomes active through the addition of rules. Event-Condition-Action (ECA) rules can be considered as the most common rule format [Day88]. An ECA rule is composed of three parts: an event, a condition, and an action. An event can be a data manipulation or retrieval operation, a method invocation in an object-oriented database management system (OODBMS), a signal from a timer or a user, or a combination of these. Condition is a test on the database state. When the specified event occurs, the condition part is tested and if the test is successful, the action part is executed.

ECA rules offer a flexible mechanism for common database tasks like constraint enforcement and view maintenance. It is difficult to predict the behavior of a set of active database rules. Three properties of rule behavior: termination, confluence, and observable determinism help the database rule programmer for the prediction of rule behavior. A brief description of each of these properties can be provided as follows [AWH92]:

- **Termination:** A set of rules terminates if rules cannot continue to activate each other indefinitely.

- **Confluence:** A set of rules is confluent if the final database state at termination does not depend on the execution order of non-prioritized rules.
- **Observable Determinism:** A set of rules is observable deterministic if the appearance of actions visible to the environment does not depend on the execution order of non-prioritized rules.

Termination, confluence and observable determinism can be very difficult or impossible to achieve in most cases, therefore some conservative algorithms have been developed for that purpose [AHW95]. These algorithms either guarantee that a set of rules terminates, is confluent, is observable deterministic or say that they may not terminate, may not be confluent, may not be observable deterministic. In static rule analysis, we don't have a priori knowledge about the execution patterns of rules. A rule may or may not trigger another rule depending on a condition on the database state. Future database states that can affect the execution order of rules are not available to us in static analysis.

A new execution model along with priority specifications schemes is proposed in [KC95] to achieve confluent rule execution in active databases. In this work it is assumed that trigger and dependency graphs that are used to check confluence are available. Trigger graph is a directed graph representing trigger relationships between rules. Dependency graph is an undirected graph which represents data and untrigger dependencies [KC95]. Data dependency between two rules indicates that they access the same object in their action parts and at least one of the operations is a write operation. Untrigger dependency between two rules means that the same data object is written to in the action part of one of the rules and read in the condition part of the other rule. To our knowledge, it has not been attempted so far to construct trigger and dependency graphs from class and rule declarations in an OODBMS.

In this thesis, we aim to construct trigger and dependency graphs using the class and rule declarations of an application given as input to the Sentinel Active OODBMS which was developed at the University of Florida. Snoop [CM93] is the event specification language of Sentinel. By using the original input files and the SNOOP preprocessed file, we construct an intermediate data

structure that we call *rulegraph*. Rulegraph is a linked list of rules in which each node has pointers to the action list, condition list, event list of that rule. These lists represent the action, condition and event part of that rule respectively. Trigger and Dependency graphs are constructed by examining the trigger and data dependencies between the rules using event lists, condition lists and action lists. Construction of these lists depends on the event types, condition format and action format of Sentinel.

Once the graphs are available, we can check for termination by performing cycle detection on the trigger graph. If there is no cycle in the trigger graph then the rules are guaranteed to terminate [AWH92]. We can achieve confluent rule execution by processing the trigger and dependency graphs as described in [KC95]. Our basic work is the construction of these graphs in an active OODBMS. In the thesis, we provide the implementation details and also discuss some sample rule executions. Our work provides the first implementation of a preprocessor to construct these graphs using the class and rule declarations.

A detailed discussion of the issues introduced in this chapter is provided in the following chapters. In Chapter 2, we discuss the problem, difficulties and limits. In Chapter 3, we provide a detailed description of static rule analysis in active DBMSs (ADBMSs). A brief description of Sentinel, SNOOP event specification language, and a description of the preprocessor we have designed is given in Chapter 4. Chapter 5 presents the implementation details. Finally in Chapter 6, conclusions and future work are discussed.

Chapter 2

Problem, Difficulties and Limits

When designing an application using active rules, we must make sure that confluence and termination properties hold. This is an important part of active database application development. The problem is how we can detect these properties using class and rule declarations. There exists some work on doing this when trigger and dependency graphs are available, therefore the missing part is to construct these graphs using class and rule declarations. Constructing these graphs is difficult because we need the syntax and semantics of ECA rules. Each active database has different syntax and semantics, so we need to restrict our work to one of them. We have chosen Sentinel active OODBMS. Relational and object-oriented active DBMSs have substantial differences regarding this matter. Below we provide a brief description of the rule syntax of an active relational DBMS(Starburst) and the Sentinel active OODBMS to indicate the differences between performing static rule analysis in each type of DBMSs. The problem of rule analysis has already been dealt with on relational active DBMSs [AHW95]. In relational active DBMSs there exists a limited number of event types: inserted, deleted, and updated. In active OODBMSs, it is possible to have rich event sets like Snoop that has about 10 event types. In relational active DBMSs there are limited number of action types like insert, delete, and update but in active OODBMSs every method call is a potential event. Another source of difficulties to analyze rules in active OODBMSs is the object-oriented paradigm. We have to keep track of which object is an instance of which class, and also we need additional information from user to provide

```
create rule name on table
when triggering-operations
[if condition]
then action
[precedes rule-list]
follows rule-list]
```

Figure 2.1: Syntax of Starburst Rule

```
emp(id,rank,salary)
sales(emp-id,month,number)

create rule good-sales on sales
when inserted
then update emp
      set salary = salary +10
      where id in (select emp-id from inserted where number >50)
```

Figure 2.2: Example of a Rule in Starburst

a more precise analysis. We face certain limitations while constructing trigger and dependency graphs using class and rule declarations. The information required for the construction of graphs is not available at compile time. The condition of a rule is a test on the database which can dynamically change during execution. When we detect a cycle in the trigger graph, we can only say that the rule set may not terminate because after a number of executions the condition of one of the rules in the cycle can become false and the cycle can be broken.

Starburst is an active relational DBMS. The syntax of a rule in Starburst is provided in Figure 2.1. The triggering-operations are one of *inserted*, *deleted*,

```

newco = new company("newco",40, 200000.00,5000.00,30,5);
event end(company_ce1:newco) void incrementemployee();
event end(company_ce2:newco) void decrementemployee();
event company_ce_or = OR(company_ce1,company_ce2)
rule comprule1[company_ce_or, cond_ce_or, action_ce_or, RECENT];
int cond_ce_or(L_OF_L_LIST *n1_list)
{
    if
        (newco->getsalesperemployee() >200000)
        return 1;
}
void action_ce_or(L_OF_L_LIST *n1_list)
{
    newco->updatesalesperemployee();
}

```

Figure 2.3: Example of a Rule Specification in Sentinel

and *updated(a,b,..)* where *a,b,..* are columns of the rule's *table*. The optional *condition* specifies an SQL predicate. The *action* specifies a sequence of database operations to be executed when the rule is triggered and its condition is true. These operations can be standard SQL data modification operations (insert, delete, update), SQL data retrieval operations (select) and transaction abort (rollback). The optional *precedes* and *follows* clauses are used to induce a partial ordering on the set of defined rules database schema and an example of a rule in Starburst is shown in Figure 2.2. This rule increases an employee's salary by 10 whenever that employee posts sales greater than 50 for a month [AHW95].

Sentinel is an active OODBMS developed using Open OODB. OpenOODB is an open (i.e, extendible) object-oriented database management system [Tex93] developed at Texas Instruments. In Sentinel events and rules are objects. An example of a rule in Sentinel is given in Figure 2.3. In Sentinel condition and action of a rule are represented as functions in which the specified methods can be called. In condition function change to database state is not allowed.

Chapter 3

Static Analysis of Active Rules

3.1 Definitions

A rule execution sequence (RES) is a sequence of rules that the system can execute when a user transaction triggers at least one rule in the sequence [KC95]. In the following definitions, R denotes a system rule set and D denotes the set of all database states. (d_j, R_k) , where $d_j \in D$ and $R_k \subseteq R$, denotes a pair of a database state and a triggered rule set. Set of rules directly triggered by a user transaction is called User-Triggered-Rule-Set (UTRS). UTRS is a multiset since more than one instance of a rule can be in it.

The following definitions are adapted from [KC95].

Partial RES Given R and D , for a nonempty set of triggered rules $R_k \subseteq R$ and a database state $d_j \in D$, a *partial RES*, σ is defined to be a sequence of rules that connects pairs of a database state and a triggered rule set as follows:

$$\sigma = \langle (d_j, R_k) \xrightarrow{r_i} (d_{j+1}, R_{k+1}) \xrightarrow{r_{i+1}} \dots \xrightarrow{r_{i+m-1}} (d_{j+m}, R_{k+m}) \rangle$$

where $d_{j+l} \in D (1 \leq l \leq m)$ is a new database state obtained by the execution of r_{i+l-1} , each rule $r_{i+l} (0 \leq l < m)$ is in a triggered rule set R_{k+l} , and eligible for execution in d_{j+l} ; i.e., d_{j+l} evaluates the rule's condition test to true. Each triggered rule set $R_{k+l} \subseteq R (1 \leq l \leq m)$ is built as $R_{k+l} =$

$((R_{k+l-1} - \{r_{i+l-1}\}) - Ru_{k+l}) \cup Rt_{k+l}$, where Ru_{k+l} is a set of rules untriggered by r_{i+l-1} and Rt_{k+l} is a set of rules triggered by r_{i+l-1} .

We are interested in Complete RES which is a partial RES that satisfies certain conditions.

Complete RES Given R and D , for a nonempty set $R_k \subseteq R$ which is a set of rules triggered by a user transaction and $d_j \in D$ is a database state produced by operations in the user transaction, a *complete RES (or RES)*, σ is defined to be a partial RES:

$$\sigma = \langle (d_j, R_k) \xrightarrow{r_i} (d_{j+1}, R_{k+1}) \xrightarrow{r_{i+1}} \dots \xrightarrow{r_{i+m-1}} (d_{j+m}, R_{k+m} = \emptyset) \rangle$$

where no triggered rules remain after execution of the last rule r_{i+m-1} .

Rule shuffling Given a partial RES σ_1 , two rules r_i and r_j in σ_1 can exchange their positions provided $r_j \in R_y$, yielding a different partial RES σ_2 as below:

$$\sigma_1 = \langle (d_x, R_y) \xrightarrow{r_i} (d_k, R_l) \xrightarrow{r_j} (d_u, R_v) \rangle$$

$$\sigma_2 = \langle (d_x, R_y) \xrightarrow{r_j} (d_m, R_n) \xrightarrow{r_i} (d_s, R_t) \rangle$$

If shuffling two rules gives the result, then these rules are said to be *commutative*. Commutativity is an important property to show the confluence of a rule set.

Rule commutativity Given R and D , two rules $r_i, r_j \in R$ are defined to be *commutative*, if for all $R_y \subseteq R$, where $r_i, r_j \in R_y$, and for all database state $d_x \in D$, the following two partial RESs can be defined:

$$\langle (d_x, R_y) \xrightarrow{r_i} (d_k, R_l) \xrightarrow{r_j} (d_u, R_v) \rangle$$

$$\langle (d_x, R_y) \xrightarrow{r_j} (d_m, R_n) \xrightarrow{r_i} (d_u, R_v) \rangle$$

where $d_x, d_k, d_m, d_u \in D$ need not be distinct and likewise $R_y, R_l, R_n, R_v \subseteq R$ need not be distinct.

Equivalent partial RESs Two partial RESs σ_i and σ_j are defined to be equivalent(\equiv) if:

1. σ_i and σ_j begin with the same pair of database state and triggered rule set, and end with the same pair of database state and triggered rule set, and
2. in σ_i and σ_j the same set of rules is triggered, possibly in different orders.

Equivalence Class of partial RESs For a partial RES, $\sigma \in S$, the *equivalence class* of σ is the set S_σ defined as follows

$$S_\sigma = \{\gamma \in S \mid \gamma \equiv \sigma\}$$

All partial RESs in an equivalence class have the same result.

Confluent Rule Set Given R and D , if there exists only one equivalence class of complete RESs for every nonempty set $R' \subseteq R$ and every $d \in D$, R is defined to be *confluent*.

3.2 Static Analysis of Active Rules

Static analysis is the systematic examination of the rule structure for the purpose of showing that certain properties are satisfied, regardless of the execution path. Static analysis can be performed without the execution of rules. The most notable difference between this technique and dynamic analysis is the presentation of actual rule behavior. Static analysis represents actual behavior with a model based upon the rules semantic features and structure, while dynamic analysis represents actual behavior with actual executions. All models are simplifications built by discarding details. Thus static analysis results are based upon simplifications, and cannot support probings of the rules to arbitrary levels of detail [Ost96].

In active databases, static analysis techniques are used to determine termination and confluence of rule sets. Detection of these properties is important to be able to implement applications correctly.

Let us examine the termination and confluence problems in active rules. Consider two rules R_i and R_j , in which R_i 's action can trigger R_j and R_j 's action can trigger R_i . It is possible that R_i and R_j can keep triggering each other indefinitely. This is the so called termination problem. To describe the confluence problem, consider two rules R_i and R_j triggered and ready for execution. We must select one of the rules to execute. If no priorities on rules are specified, the final database state may depend on the order in which the rules are executed. If R_i 's action changes the database state in a way that R_j 's condition evaluates to false, then first R_j and then R_i can be executed; but when R_i is executed first, then R_j will not be executed. Thus we may come up with different database states. In this case, the rules are not commutative and we have the confluence problem.

3.3 Dependencies

Two kinds of dependencies are defined on active rules [KC95].

- **Data Dependency:** Two distinct rules R_i and R_j have a data dependency if R_i writes in its action part to an object that R_j reads or writes in its action part or vice versa.
- **Untrigger Dependency:** Two distinct rules R_i and R_j have an untrigger dependency if R_i writes in its action part to a data object that R_j reads in its condition part or vice versa.

If there is a data dependency between two rules, one rule can change what the other rule reads or overwrite the data written by the other. In this case, the final outcome depends on the execution order of the two rules. If there is no data dependency between the rules, two rules are independent and the final outcome of their execution is the same regardless of the execution order.

If there is an untrigger dependency between two rules, one rule's action can change the condition and other rule may or may not execute depending on this change. Therefore, the final database state depends on the execution order of the two rules.

Absence of data and untrigger dependencies is a sufficient condition for two rules to be commutative [KC95]. If there is a dependency between the rules, they are said to be conflicting with each other.

3.4 Dependency Graph

A *dependency graph* [KC95] $DG = (R, E_D)$ is an undirected graph where R is the rule set and E_D is the dependency edge set. For each rule $r_i \in R$ there is a corresponding node i in the graph. There is a dependency edge (u, v) between two nodes u and v if and only if there is at least one of data dependency and untrigger dependency between the rules r_u and r_v .

3.5 Trigger Graph

A *trigger graph* [AWH92] $TG = (R, E_T)$ is an acyclic directed graph where R is the rule set and E_T is the trigger edge set. There is a trigger edge (u, v) between two nodes u and v if and only if the rule r_u denoted by node u can trigger the rule r_v denoted by node v . If there is no cycle in the trigger graph TG then the rules in R are guaranteed to terminate.

3.6 Related Work

In [AWH92] and [AHW95], some static analysis methods are provided to determine whether a given set of rules terminate, confluent and observable deterministic in the context of Starburst rule system. Starburst is a relational database system [Wid96] so the static rule analysis problem is investigated in

the context of relational databases. [WH95] deals with the termination problem in the OSCAR OODBMS model, and describes a set of algorithms that allow efficient analysis of termination of a set of rules. [vdVS93] presents a design theory for the static detection of confluence and termination in OODBMSs and prove that the static detection of termination and confluence is a decidable problem.

[BCW93] uses an extension of relational algebra for description of active database rules, and provides an efficient termination analysis. [BW95] uses a propagation algorithm based on the extended relational algebra to determine when the action of a rule can affect the condition of another, and to determine when rule actions commute. This approach is widely applicable to relational active databases.

Researchers and developers agree that one of the main difficulties in the development of rule applications is the lack of design methods and of suitable design tools [CR96]. Several tools have been developed to help the user understand the behavior of rules. [DJP93] provides a debugger for active rules in object-oriented context. [BGB95] presents a tool which helps the user in defining, tracing, and debugging a set of active rules. [JUD96] presents the prototype of an active rule debugging environment. [BCFP96] describes another tool for active rule generation, analysis, debugging, and browsing. [CTZ95] provides a visualization tool developed for Sentinel Active OODBMS. [BCP96] assuming the relational model introduces a modularization technique for designing active rules.

[KC95] proposes a new execution model along with priority specification schemes to achieve confluent rule execution in active databases. It assumes that trigger and dependency graphs are available to the system. There is no work to our knowledge that constructs the trigger and dependency graphs from the class and rule declarations provided to an active OODBMS. The aim of our work is to provide this construction on the Sentinel Active OODBMS.

Chapter 4

Trigger and Dependency Graphs

4.1 Sentinel

Sentinel is an ADBMS implemented on top of Open OODB [Tex93]. Rules of Sentinel are expressed in the ECA format. Sentinel supports rules in both centralized and distributed environments. Event and rule specifications in Sentinel are incorporated into the C++ language. An event specification language called Snoop was developed to specify events [CM93]. The grammar of Snoop is described in Figure 4.1. In Sentinel, any invocation of a method is a potential primitive event. A set of operators are used to construct composite events using primitive and composite events. Snoop supports both local events and global events. Local event detector and global event detector were implemented to monitor events in centralized and distributed environments [Lia97].

Three types of primitive events are supported by Sentinel [Lia97]:

1. **Database Events**, which correspond to database operations used to manipulate data. Every method of an object is a potential primitive event, and they are transformed into events using two event modifiers: *begin-of* and *end-of*.

```

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

```

Figure 4.1: BNF for Snoop

2. **Temporal Events**[Lee96], which include *absolute* and *relative* temporal events. An *absolute* temporal event is specified as an absolute value of time. A *relative* temporal event is specified by a reference event and a time offset.
3. **External Events**, which denote events defined by users or application programs and are registered with the system. They are also called global events which support ECA rule processing in a distributed system. External events are assumed to be detected outside the system but are signaled to the system along with their parameters.

Events and rules can be defined at either class level or instance level. A class level event/rule is applicable to every object of that class and declared inside a class definition. An instance level event/rule is applicable to specific object instances.

Sentinel supports four parameter contexts: *recent*, *chronicle*, *continuous*,

```

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

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

rule_spec ::= rule rule_name (event_name,
                               condition_function, action_function
                               [,parameter_context],[coupling_mode]
                               ,[priority],[rule_trigger_mode])

parameter_context ::= RECENT | CHRONICLE | CONTINUOUS
                    | CUMULATIVE
coupling_mode ::= IMMEDIATE | DEFERRED | DETACHED
priority ::= positive integer
rule_trigger_mode ::= NOW | PREVIOUS

```

Figure 4.2: BNF for Snoop

and *cumulative* and two coupling modes: *immediate* and *deferred*. Multiple rule executions, nested rule execution, and prioritized rule execution are supported in Sentinel.

The syntax of the Snoop event/rule specification [Lee96] is provided in Figure 4.2.

4.2 Construction of Trigger and Dependency Graphs

An example of class declaration and rule format in an OODBMS is given in Figure 4.3¹. In this example, an employee class is declared. The class has three class level rules (CLR): classrule1, classrule2 and classrule3. Mike, joe

¹This is a very simple rule format of a hypothetical system.

```
class employee;

employee mike;

class employee {
private:
    real salary;
    int rank;

public:
    int getsalary() {return salary; }
    void setsalary(real x) { salary = x;}
    void increasesalary( int percent) { salary = salary*(100+percent)/100;}
    void salarybound() { salary = 100000;}
    void increaserank() {rank++;}
    void setrank( int y) { rank = y;}
    void getrank() { return rank; }

rules:
    classrule1 ( setsalary() or increasesalary() , mike.getsalary() > 100000,
                mike.salarybound() ; );
    classrule2 ( salarybound() , mike.getrank() > 5 ,
                mike.increasesalary(10) ; );
    classrule3 ( salarybound() , mike.getrank() > 7 ,
                mike.increasesalary(20) ; );
};

employee joe ;
rule instancerule1 ( mike.setsalary() or mike.increasesalary() ,
                    mike.getsalary() > joe.getsalary() ,
                    joe.increasesalary(10) ; );

employee jane ;
rule instancerule2 ( jane.setrank() ,
                    jane.getrank() > mike.getrank() ,
                    mike.increaserank() ; );
```

Figure 4.3: Example of Rules

```
Name : classrule1
Type : 1
Base : employee
Eventlist:
(increasesalary ) (setsalary )
Conditionlist:
(mike getsalary )
Actionlist:
(mike salarybound )

Name : instancerule2
Type : 2
Eventlist:
(jane setrank)
Conditionlist:
(jane getrank) (mike getrank)
Actionlist:
(mike increaserank)
```

Figure 4.4: Example Nodes of Rulegraph

and jane are three instances of the employee class object. Two instance level rules (ILR) are declared: instancerule1 and instancerule2. Each rule has event, condition, and action parts separated by commas. The event part is composed of method calls and operators. The condition part has a comparison operator and the action part is composed of sequence of method calls.

In Sentinel, constructing trigger and dependency graphs can be done conservatively. Each method call is a potential event. If we assume that the action part of rules consists of method calls only, constructing a trigger graph is easy. Two nodes of the rulegraph for the example shown in Figure 4.3 is given in Figure 4.4. If a method call is executed in the action part of a rule and an event is defined in another rule on the same method call, then there is a trigger dependency and a directed edge should be added to the trigger graph. To detect these dependencies we can insert entries for the method calls executed in the action part of a rule into a linked list and insert entries for the method calls in the event part of a list into another linked list. If the two lists have

at least one common element, this means that there is a trigger dependency. Constructing the eventlist is not trivial when the rich event set of Snoop is used. This problem is addressed later in this chapter.

Constructing a dependency graph is similar. An edge is added to the dependency graph if there exists either data dependency or untrigger dependency between any two rules. An entry for every method call executed in the action part of a rule is inserted into an action list constructed for that rule. Condition of a rule is actually a test performed on the database state. In an OODBMS these tests will be performed by means of method calls. Each method call accessed in the condition part of a rule is inserted into a condition list associated with that rule. To detect data dependencies between two rules, we find the intersection of the action lists of these rules. If the intersection is not null, then there exists a data dependency and an edge is inserted to the dependency graph. To detect untrigger dependencies between two rules, we find the intersection of the condition list of the first rule and the action list of the second rule. If two lists intersect, then this means an untrigger dependency, and an undirected edge is added from the node of the first rule to the node of the second rule into the dependency graph.

The event list, condition list, and action list of rules, that are used to perform dependency tests, are stored in the linked list of rule nodes as shown in Figure 4.5. Each rule node has a pointer to the linked list of each of the event, condition, and action of the corresponding rule. A rule node also stores the name, baseclass, and type of that rule. Both CLR and ILR are represented by this structure.

The process of detecting dependencies among rules is not trivial when both CLR and ILR are allowed. If both rules that are checked for dependencies are CLR and they access the same object, there may be data or untrigger dependency even if they call different methods. This is because different methods can use the same variable. For this reason, we assume that two CLR that access the same object are conflicting by default if detailed conflict detection, explained later in the chapter, is not provided.

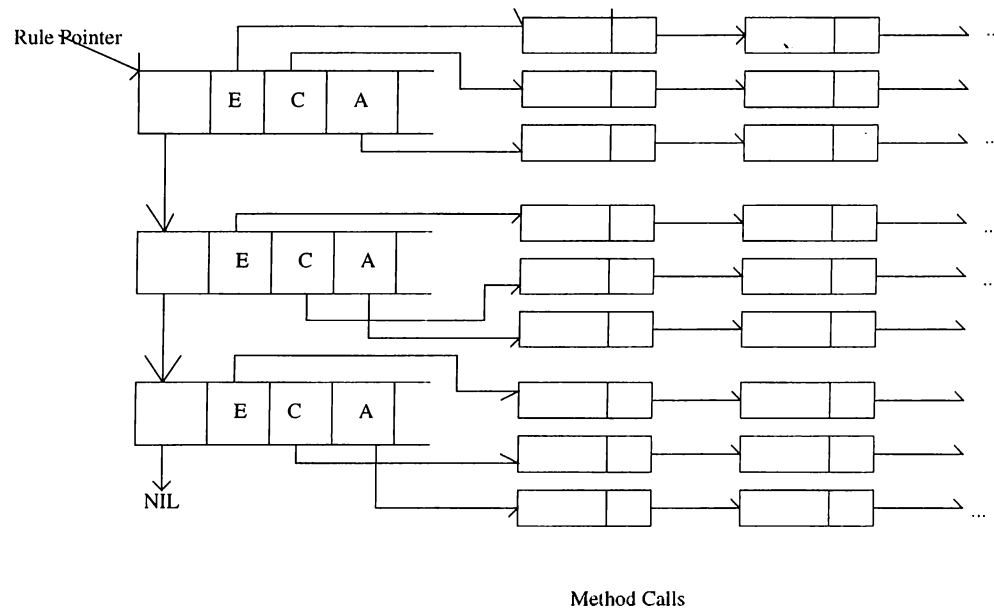


Figure 4.5: Rule Graph

4.3 Event Types

There are 10 types of events specified in Sentinel [Lee96],[CM93]. A brief description of each of these events is given below.

1. **AND(E,F)**: Conjunction of two events E and F is raised when both events are raised irrelevant of the order.
2. **OR(E,F)**: Disjunction of two events E and F is raised when one of the events is raised.
3. **SEQ(E,F)** : Sequence of two events E and F is raised when F is raised provided that E has already been raised.
4. **NOT(E)(F,G)** : The event is raised when E does not occur in the interval formed by the occurrence of events F and G .
5. **ANY(m,E,F,G,...)** : The event is raised when m events out of the specified list of events $E,F,G,..$ occur irrelevant of the order.
6. **A(E,F,G)** : The event is raised each time the event F is raised in the interval formed by the occurrence of events E and G .

7. **A*(E,F,G)** : Cumulative variant of *A*. The event is raised once when the event *G* is raised if the event *F* is raised in the interval formed by the occurrence of events *E* and *G*.
8. **P(E,F,G)** : The event is raised every amount of time specified by the event *F* in the interval formed by the occurrence of events *E* and *G*.
9. **P*(E,F,G)** : Cumulative variant of *P*. The event is raised once when the event *G* is raised and accumulates the time of occurrences of the periodic event whenever event *F* occurs.
10. **PLUS(E,[TI])** : The event is raised after *TI* time units when the event *E* is raised.

4.4 Input Processing

In constructing the rulegraph which contains information about the rules, we process the class and rule declarations and the Snoop preprocessed file. The aim in processing class and rule declarations is to find out which class level rule belongs to which class. This information is not available in Snoop preprocessed file. For detailed conflict detection we process the dcg extension files and insert the information into graphs for each class. The Snoop preprocessed file has all the other necessary information. Events and rules are represented here using Open OODB syntax. Condition and action parts of a rule are represented as functions. When we construct the condition list and action list of a rule we must access the information in these functions. For this purpose we use the following strategy: when we process a rule, we insert the name of the condition and action functions of that rule into a binary tree structure with pointers to associated rule node. When we process the function, by following that link we find the corresponding rulegraph node and construct the condition and action lists.

4.5 Event List Construction

<i>Event Type</i>	<i>Inserted Into Eventlist</i>
AND(E,F)	E,F
OR(E,F)	E,F
SEQ(E,F)	F
NOT(E)(F,G)	G
ANY(m,E,F,G...)	E,F,G,..
A(E,F,G)	F
A*(E,F,G)	G
P(E,F,G)	E
P*(E,F,G)	G
PLUS(E,[TI])	E

Table 4.1: Eventlist Construction

Event list construction is based on the type of the events described in the preceding section. As an example, the event AND(E,F) is raised when both events E and F are raised. Since we don't know the history of events we can assume that the occurrence of any of two events can lead to the raise of AND(E,F) (considering the possibility that the other event might have already been raised). Therefore we insert both E and F into the event list. As another example, A(E,F,G) is raised each time the event F is raised in the interval formed by the events E and G. Only the occurrence of event F can result in the raise of A(E,F,G); therefore we insert F into the event list. Event list construction of all types of events is specified in Table 4.1.

For composite events that can include more than one event type of those described in Section 3.3, the event list can be constructed recursively as follows. Let $E = EVENTTYPE(F,G)$ be an event expression, where F and G are event expressions and $EVENTTYPE$ is the highest precedence event in E . The event lists of F and G are constructed on the basis of the Table 4.1 and then the event list of E is constructed, again referring to Table 4.1 for the event type of $EVENTTYPE$. If an event expression is a primitive event then the event list contains only that primitive event. In construction of event list precedence and associativity rules of SNOOP are also important.

```

< expression > ::= < term > | < expression > or < term >
< term > ::= < item > | < term > and < item >
< item > ::= < methodcall > < comparator > < number >
           | < methodcall > < comparator > < methodcall >
           | < number > < comparator > < methodcall >
< comparator > ::= < = > | < > | < > | < > | < >

```

Figure 4.6: Declaration of Conditions

```

int functionname(...)
{
  if (condition)
    return(1)
}

```

Figure 4.7: Declaration of Condition Functions

4.6 Condition List Construction

We assume that the condition part of a rule is expressed as shown in Figure 4.6. In this declaration, method calls can have other method calls as parameters which might be necessary in some applications. In Sentinel, condition and action parts of a rule are represented as functions.

Each method call executed in the condition part of a rule is inserted into the condition list of that rule. Conditions are functions in Sentinel. We assume that a condition in Sentinel, which is represented as a function, is defined as given in Figure 4.7.

```
delete eventname
delete rulename

insert evendeclaration
insert ruledeclaration

modify eventdeclaration
modify ruledeclaration
```

Figure 4.8: Format of File for Incremental Methods

4.7 Action List Construction

We assume that the action of a rule consists of a sequence of method calls. Each method call executed in the action part of a rule is inserted into the action list of that rule.

4.8 Incremental Methods

In our current implementation, dependency and trigger graphs are constructed after any change to the rule set. In fact, previous graphs are still valid and only an incremental analysis needs to be performed. All our design and implementation were performed such that any change in the rule set can be handled through incremental methods. However, we require that at each update, the deleted, inserted, and modified class and rule declarations should be given in a file. The format of this file can be as given in Figure 4.8

4.9 Detailed Conflict Detection

When two method calls operate on the same data item, they can be assumed to be conflicting but this may not be the case in reality. Two method calls

DEPENDENCY GRAPH

```

comprule1 :  comprule2  comprule3  insrule4
comprule2 :  comprule1  comprule3  insrule4
comprule3 :  comprule1  comprule2  insrule4
insrule1 :  emptemp2   insrule2
insrule2 :  emptemp1   emptemp2   insrule1   insrule3
insrule3 :  emptemp1   insrule2
insrule4 :  comprule1  comprule2  comprule3
insrule5 :
emptemp1 :  insrule2   insrule3
emptemp2 :  insrule1   insrule2

```

Figure 4.9: Dependency Graph Without Detailed Conflict Detection

```

updatesalesperemployee setefficiency
incrementemployee setefficiency
updatesalesperemployee getsalesperemployee

```

Figure 4.10: Detailed Conflict Detection File Demo_company.dcg

can access different variables and they may not conflict. It is not possible to determine the conflicts of such method calls without getting some information from the user. In our implementation we require the user to specify the methods that do not conflict in a .dcg extension file. The name of a file with .dcg extension will be the same as the file in which the class was declared. The more number of nonconflicts provided by the user, the more accurate dependency graph can be generated.

We use a graph structure to keep track of the list of nonconflicts of a class. Each class has an associated graph if its corresponding .dcg file is specified. If all nonconflicts of classes are specified, an accurate dependency graph can be obtained by using this strategy. This will reduce the number of dependencies in the dependency graph.

For the rule set given in Appendix, without detailed conflict detection the dependency graph is as shown in Figure 4.9. If we provide the .dcg file shown in Figure 4.10, we get the dependency graph where the dependencies $(comprule1, comprule2)$, $(comprule2, comprule1)$, $(comprule1, comprule3)$ and $(comprule3, comprule1)$ are removed.

We eliminated the existence of such edges between the nodes of *comprule1* and *comprule2*, and the nodes of *comprule1* and *comprule3* by providing the information in Figure 4.10. The nodes corresponding to rules *comprule3*, *comprule2*, *comprule1* are given in Figure 4.11 to make this clear.

```
Name : comprule3
Type : 1
Base : company
Events:
(updatesalesperemployee )
Conditions:

Actions:
(newco setefficiency)

Name : comprule2
Type : 1
Base : company
Events:
(updatesalesperemployee )
Conditions:
(newco getsalesperemployee)
Actions:
(newco setefficiency)

Name : comprule1
Type : 1
Base : company
Events:
(decrementemployee ) (incrementemployee )
Conditions:

Actions:
(newco updatesalesperemployee)
```

Figure 4.11: Nodes Of Rulegraph For Rules comprule1, comprule2, comprule3

Chapter 5

Implementation

In this chapter, we discuss the implementation details of the construction of the trigger and dependency graphs.

5.1 Data Structures

We construct trigger and dependency graphs using some information about the rules of the system. In Sentinel, this information is provided in the header files which contain class declarations, class level events, rules and the SNOOP preprocessed file. See Appendix A for a sample input file and the SNOOP preprocessed form.

We refer to the SNOOP preprocessed form of the input file as the main file. In processing the main file, we do not use the part preceding *Init.call*. Following this part we have class level event and rule declarations.

Our aim is to construct trigger and dependency graphs. Trigger graph is a directed graph and dependency graph is undirected. We use a directed graph that we call *trigger* for trigger graph and an undirected graph called *dependency* for dependency graph.

It is not easy to find which class level rule operates on which class. We

process the header files and insert the name of a rule and the class it operates on into a binary tree called *ruleclass*. A rule described in the header file operates on the class in which it is declared.

We have to construct the event lists of all specified events. We use a graph structure in constructing the event lists. For each primitive event we just store the name of the event in this graph. Composite and nested events might cause some problems. We call an event a nested event if at least one of its arguments¹ is a composite event. In constructing the event list of a composite or nested event it is necessary to find the eventlists of arguments. For this purpose, we keep the eventlist of every event defined in an adjacency list graph structure called *eventgraph*. The nodes of the graph contain the names of events and a pointer to the event list of the associated rule. To construct the event list of a nested event, we use the algorithm given in Figure 5.1. The function *Process_Primitive_Event* inserts the event into the *eventgraph* and the function *Process_Basic_Event* inserts the arguments of the event into the *eventgraph* according to the rules given in Table 4.1. An event which has only one event type is called a basic event. We assign a temporary name to the event whose event list is constructed, and insert that name with the constructed event list in the *eventgraph*. An example of the construction of the *eventgraph* for a nested event is given in Figure 5.2. In this example, we first construct the event lists of the arguments. We have a composite event *AND* as one of the arguments of the nested event. We construct the eventlist of *AND* and insert it into the *eventgraph* together with temporary name *1*.

We assume that class level rule and event specifications appear before *OpenOODB→beginTransaction()*² in the SNOOP preprocessed file. We keep all the information about rules in *rulegraph*. *rulegraph* is constructed from the event lists in the *eventgraph*. The baseclass of a CLR is found using *ruleclass* binary tree structure.

To determine which variable belongs to which class, we look for places where constructors are called. Variables are kept in a binary tree called *vartree*.

¹The arguments of an event *EVENTTYPE(E,F)* are E and F.

²Starts an Open OODB transaction.

```

for each event  $E$ 
  if  $E$  is primitive
     $Process\_Primitive\_Event(E)$ 
  else
    while  $E$  is still a nested composite event do
       $E_i(X, Y..) \leftarrow Find\_Innermost\_Event(E)$ 
       $EL_x \leftarrow$  eventlist of  $X$  in eventgraph
       $EL_y \leftarrow$  eventlist of  $Y$  in eventgraph
       $EF \leftarrow$  construct the eventlist of  $E$  using  $EL_x, EL_y, ..$ 
      insert  $EF$  into eventgraph with name  $id_{E_i}$ 
       $E_{old} \leftarrow E$ 
       $E \leftarrow$  replace  $E_i(X, Y, ..)$  with  $id_{E_i}$  in  $E_{old}$ 
     $Process\_Basic\_Event(E)$ 

```

Figure 5.1: Algorithm to Construct Event Lists of Nested Events

In Sentinel conditions and actions of rules are specified as functions and are declared after *main()*. We must keep track of which function belongs to which rule and whether it is a condition or an action. For this purpose, we use a binary tree called *functiontree*. Each node in that tree has the name of the function, whether it is a condition or an action and a pointer to the rule node that function belongs to. This way we can find the rule that function belongs to in $O(\log N)$ comparisons where N is the number of rules.

In detailed conflict detection, the nonconflicts are specified in .dcg extension files. The .dcg extension file must have the same name with the .sh extension file. The file can have at most one class declaration and event and rule declarations associated with that class. We have to keep track of which class is declared in which file because we label the detailed conflict graphs using class name. We keep this information in a binary tree called *fileclass*.

Declaration of CLR is similar to that of ILR and declaration of class level events is similar to that of instance level events.

Each node in the linked list of rules has the following fields:

```

Example of a Nested Event
("STOCK_e_COMP1",new AND(STOCK_e2, STOCK_e3),STOCK_re15);

After 1 step
("STOCK_e_COMP1",1 ,STOCK_re15);

EVENTGRAPH
1 : STOCK.buy_stock  STOCK.sell_stock
STOCK_e2 : STOCK.sell_stock
STOCK_e3 : STOCK.buy_stock
STOCK_re15 : TEMPORAL.10 sec

```

Figure 5.2: Sample Run on a Nested Event

<i>Field</i>	<i>Type</i>	<i>Meaning</i>
type	boolean	CLR or ILR
name	pointer to string	name of the rule
baseclass	pointer to string	the class for which CLRs are defined
eptr	list object	linked list to store events
cptr	list object	linked list to store conditions
aptr	list object	linked list to store actions
next	pointer to rulenode	pointer to the next rule

Each node of the rule list has the following fields for ILR, in addition to the fields specified above:

<i>Field</i>	<i>Type</i>	<i>Meaning</i>
fcname	pointer to string	name of the object
nname	pointer to string	name of the method
next	pointer to listnode	pointer to the next element

For CLR, a node has the same fields with different contents.

<i>Field</i>	<i>Type</i>	<i>Meaning</i>
fname	pointer to string	name of the method
nname	pointer to string	unused
next	pointer to listnode	pointer to the next element

We also keep a global linked list to store variables defined on classes and their associated classes. This list is sorted on the basis of variable names. The linked list nodes have the following fields:

<i>Field</i>	<i>Type</i>	<i>Meaning</i>
fname	pointer to string	name of the variable
nname	pointer to string	type of the variable
next	pointer to listnode	pointer to the next element

The linked lists for events, conditions, and actions are kept in lexicographical order based on their first field. This makes the process of determining conflicts easy: if both rules are of the same type, it is simply checked whether two lists intersect or not. We assume that an ILR can span several classes, but a CLR spans only one class.

To detect conflicts between an ILR R_i having part list L_i and a CLR R_j having part list L_j , the algorithm in Figure 5.3 is used. Part list refers to either one of an event list, a condition list, or an action list. The algorithm used to detect conflicts between two ILR is presented in Figure 5.4. In the algorithm, $first(L)$ denotes the first element of list L ; $next(x)$ denotes the element that succeeds x in the list. Detecting conflicts between two ILR is achieved as follows: first we find the objects that match, then find out if function names also match. To detect conflicts between two CLR, the algorithm in Figure 5.4 can be used. If the length of the part list of a CLR is n and the length of the

```

Trigger-Instance-Class(  $R_i, L_i R_j, L_j$  )
for each element  $x$  of  $L_j$ 
  if type of  $x$  = baseclass of  $R_i$ 
    for each element  $y$  of  $L_i$ 
      if name of  $y$  = function name of  $L_j$ 
        /* there is a conflict */
        return(1)
return(0)

```

Figure 5.3: Function to Detect Conflicts Between an ILR and a CLR

```

Trigger-Instance-Instance(  $R_i, L_i R_j, L_j$  )
 $x \leftarrow$  first(  $L_i$  )
 $y \leftarrow$  first(  $L_j$  )
while  $x \langle \rangle$  nil and  $y \langle \rangle$  nil
  if object name of  $x$  < object name of  $y$ 
     $x \leftarrow$  next( $x$ )
  else if object name of  $x$  > object name of  $y$ 
     $y \leftarrow$  next( $y$ )
  else while  $x \langle \rangle$  nil and  $y \langle \rangle$  nil and
    object name of  $x$  = object name of  $y$ 
    if function name of  $x$  < function name of  $y$ 
       $x \leftarrow$  next( $x$ )
    else if function name of  $x$  > function name of  $y$ 
       $y \leftarrow$  next( $y$ )
    else /* there is a conflict */
      return(1)
return(0)

```

Figure 5.4: Function to Detect Conflicts Between two ILR

```

Action-Event-Dependencies()
for each node  $i$  of rulegraph
  for each node  $j$  of rulegraph ( $j \neq i$ )
    if both  $i$  and  $j$  are ILR
      if Trigger-Instance-Instance( $i, i \rightarrow \text{aptr}, j, j \rightarrow \text{eptr}$ ) = 1
        Trigger-Insert-Edge( $i, j$ )
      else if  $i$  is an ILR and  $j$  is a CLR
        if Trigger-Instance-Class( $i, i \rightarrow \text{aptr}, j, j \rightarrow \text{eptr}$ ) = 1
          Trigger-Insert-Edge( $i, j$ )
      else if  $i$  is a CLR and  $j$  is an ILR
        if Trigger-Instance-Class( $j, j \rightarrow \text{eptr}, i, i \rightarrow \text{aptr}$ ) = 1
          Trigger-Insert-Edge( $i, j$ )
      else if Trigger-Instance-Instance( $i, i \rightarrow \text{aptr}, j, j \rightarrow \text{eptr}$ ) = 1
        Trigger-Insert-Edge( $i, j$ )

```

Figure 5.5: Algorithm to Detect Action-Event Dependencies

part list of a ILR is m , then detecting conflicts has a time complexity $O(n * m)$. If two rules are of the same type then the complexity is $O(n + m)$.

The trigger graph is a directed graph that keeps information about action-event conflicts. The algorithm in Figure 5.5 is used to detect such conflicts.

Trigger-Insert-Edge(i, j) is a procedure to insert an edge from node i to node j in the trigger graph. The running time of the algorithm in Figure 5.5 is $O(n * n)$ where n is the number of rules in the rule graph.

The dependency graph is an undirected graph that keeps information about action-action and action-condition conflicts. Figure 5.6 shows the algorithm to determine if two part lists operate on the same variable and therefore conflict. The algorithm in Figure 5.7 is used to detect action-condition and action-action conflicts. *Dependency-Insert-Edge(i, j)* is a procedure to insert an edge from node i to node j in the dependency graph.

```

Dependency(  $R_i, L_i, R_j, L_j$  )
 $x \leftarrow \mathbf{first}( L_i )$ 
 $y \leftarrow \mathbf{first}( L_j )$ 
while  $x \langle \rangle \mathbf{nil}$  and  $y \langle \rangle \mathbf{nil}$ 
    if object name of  $x <$  object name of  $y$ 
         $x \leftarrow \mathbf{next}(x)$ 
    else if object name of  $x >$  object name of  $y$ 
         $y \leftarrow \mathbf{next}(y)$ 
    else there is a conflict
        return(1)
return(0)

```

Figure 5.6: Function to Detect Dependency Conflicts

```

Action-Condition-Dependencies()
 $\mathit{temp1} \leftarrow \mathbf{first}(\mathit{rulegraph})$ 
while ( $\mathit{temp1} \langle \rangle \mathbf{nil}$ )
     $\mathit{temp2} \leftarrow \mathbf{next}(\mathit{temp1})$ 
    while ( $\mathit{temp2} \langle \rangle \mathbf{nil}$ )
        if( $\mathbf{Dependency}(i, i \rightarrow \mathit{aptr}, j, j \rightarrow \mathit{cptr}) = 1$  or
            ( $\mathbf{Dependency}(i, i \rightarrow \mathit{cptr}, j, j \rightarrow \mathit{aptr}) = 1$  or
            ( $\mathbf{Dependency}(i, i \rightarrow \mathit{aptr}, j, j \rightarrow \mathit{aptr}) = 1$ 
             $\mathbf{Dependency-Insert-Edge}(i, j)$ 
             $\mathit{temp2} \leftarrow \mathbf{next}(\mathit{temp2})$ 
     $\mathit{temp1} \leftarrow \mathbf{next}(\mathit{temp1})$ 

```

Figure 5.7: Algorithm to detect dependencies

Parameter	Meaning
c	<i>Number of classes</i>
v	<i>Number of variables</i>
i	<i>CLR per class</i>
n	<i>Number of variables per class</i>
j	<i>Number of ILR</i>
ca	<i>Number of method calls in the action part of CLR</i>
cc	<i>Number of method calls in the condition part of CLR</i>
ce	<i>Number of method calls in the event part of CLR</i>
ia	<i>Number of method calls in the action part of ILR</i>
ic	<i>Number of method calls in the condition part of ILR</i>
ie	<i>Number of method calls in the event part of ILR</i>

Table 5.1: Parameters Used in Determining Number of Comparisons

5.2 Number of Comparisons to detect dependencies

We can estimate the number of comparisons required for the implementation of conflict-detection algorithms. This estimation can be helpful to determine the bottlenecks of conflict-detection. Improvements on the parts that are identified as bottlenecks can lead to more efficient static rule analysis. Parameters used in this estimation are given in Table 5.1.

5.2.1 Number of Comparisons to Detect Action-Event Dependency

Class-class

$c * i * c * i$: Number of list operations

$ca * v * (n * ce/v + (1 - n/v))$: Cost of one list operation

n/v : Probability that a node of CLR conflicts with CLR

v : Finding the type of a variable using variable list
 $c * i * c * i * ca * v * (n * ce/v + (1 - n/v))$: Total cost
 $TC_{cc} = c * i * c * i * ia * (n * ce + v - n)$

Instance-Instance

$j * j$: Number of list operations
 $ia + ie$: Cost of one list operation
 $j * j * (ia + ie)$: Total cost
 $TC_{ii} = j * j * (ia + ie)$

Instance-class

$j * c * i$: Number of list operations
 $ia * v * (n * ce/v + (1 - n/v))$: Cost of one list operation
 n/v : Probability that a node of ILR conflicts with CLR
 v : Finding the type of a variable using variable list
 $j * c * i * ia * v * (n * ce/v + (1 - n/v))$: Total cost
 $TC_{ic} = j * c * i * ia * (n * ce + v - n)$

Class-instance

$c * i * j$: Number of list operations
 $ie + ca$: Cost of one list operation
 $c * i * j * (ie + ca)$: Total cost
 $TC_{ci} = c * i * j * (ie + ca)$

Total Cost

The total cost can then be approximated as $TC_{ae} = TC_{cc} + TC_{ii} + TC_{ic} + TC_{ci}$ for action-event dependencies.

5.2.2 Number of Comparisons to Detect Action-Condition Dependency

In determining action-condition dependency, two CLR are assumed to be conflicting if they have the same baseclass. The total cost of detecting action-condition dependencies can be specified as $TC_{ac} = TC_{cc} + TC_{ii} + TC_{ic} + TC_{ci}$, where:

$$TC_{cc} = c * i * c * i * (ca + ce)$$

$$TC_{ii} = j * j * (ia + ic)$$

$$TC_{ic} = j * c * i * (ia + cc)$$

$$TC_{ci} = j * c * i * (ca + ic)$$

5.2.3 Number of Comparisons to Detect Action-Action Dependency

Again in a similar manner, the cost of detecting action-action dependencies can be given as $TC_{aa} = TC_{cc} + TC_{ii} + TC_{ic} + TC_{ci}$, where:

$$TC_{cc} = c * i * c * i * (ca + ce)$$

$$TC_{ii} = j * j * (ia + ia)$$

$$TC_{ic} = j * c * i * (ia + ca)$$

$$TC_{ci} = j * c * i * (ca + ia)$$

5.2.4 Total Cost of Conflict Detection

The sum of all the above costs is

$$\begin{aligned}
TC &= 2 * TC_{ae} + 2 * TC_{ac} + TC_{aa} \\
&= 2 * c * i * c * i * (ia * (n * ce + v - n) + 2 * ca + 2 * ce) \\
&\quad + 2 * j * j * (4 * ia + ie + ic) \\
&\quad + 2 * j * c * i * (ia * (n * ce + v - n) + 3 * ia + 4 * ca + ic + cc + ie)
\end{aligned}$$

The total cost is mainly determined by the number of rules and the length of event, condition, and action lists.

5.3 User Interface

We implemented a simple user interface for the preprocessor. We assume that the rule set is specified as a file with `.data` extension. Such a file contains many `.sh` files in which class level event and rule declarations are made and a `.c` extension snoop preprocessed file. The user interface has the following top level menus: files, ruleset, and graphs. The files menu has two submenus: edit and quit. Edit calls an editor which can be used to see the contents of the files. Quit quits the program. Ruleset menu has three submenus: select, execute and release. Select allows us to change the current rule set. Execute runs the preprocessor on that rule set and release closes the rule set. In the graphs menu we have trigger and dependency submenus. Trigger displays the trigger graph and a message indicating whether there exists a cycle or not. Dependency displays the dependency graph. The snapshot of the preprocessor is given in Figure 5.8 and a sample trigger graph is given in Figure 5.9.

5.4 Usage of the tool

This tool can be used by anyone who needs to design an active database application. The user writes the class and rule declarations, uses the Snoop preprocessor to obtain the Snoop preprocessed file and forms the file with extension `.data`. In such a file, the last entry is the Snoop preprocessed file. Other entries are the class declarations of user defined classes. Rules and events are defined

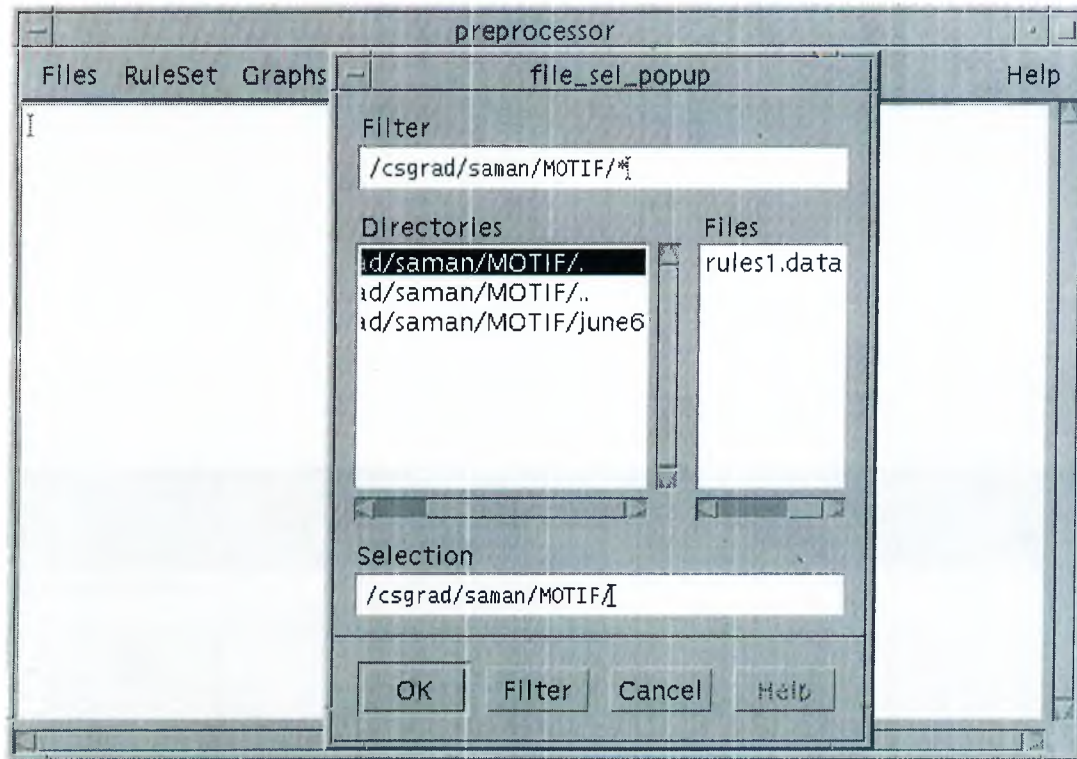
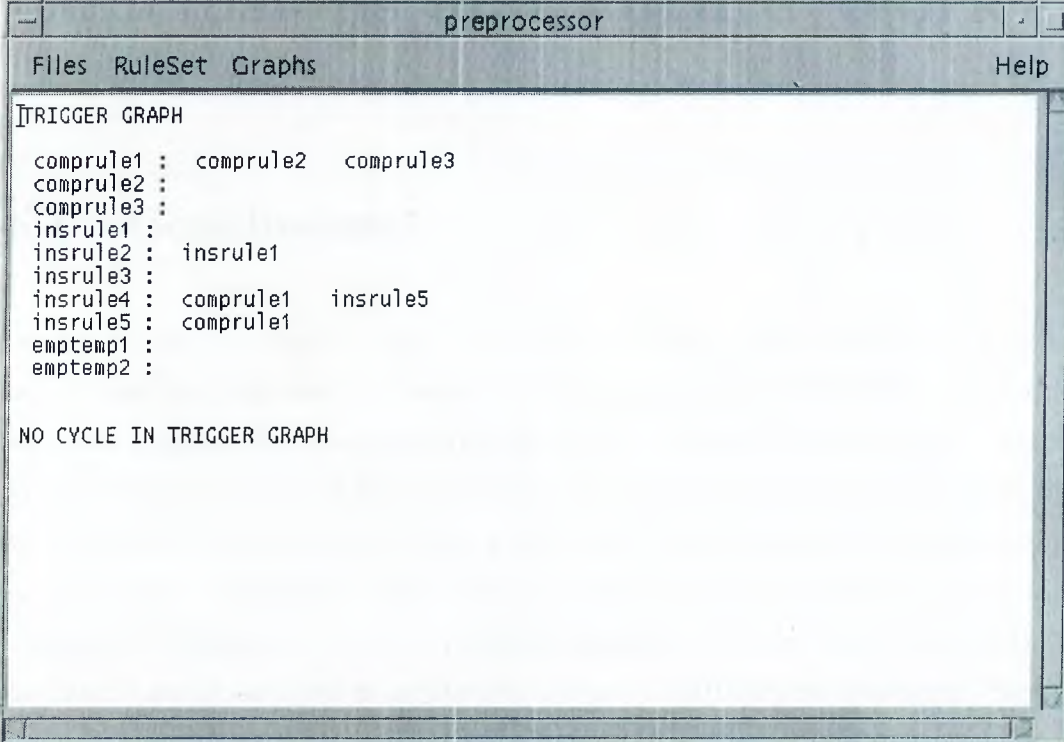


Figure 5.8: Select Rule Set

in this class declarations. Now the tool can be used. We open the ruleset by using *select* on the *ruleset* menu, and then execute the preprocessor by *execute* on *ruleset* menu. Now the trigger and dependency graphs are constructed. We can display them by selecting *trigger* or *dependency* on the graphs menu. We also perform cycle detection on the trigger graph. If there exists a cycle in the trigger graph, the list of nodes involved in the cycle is printed. The user can check if the rule set terminates or not because when there exists a cycle we can only say that the rule set may not terminate. If rules do not terminate user must update the design of rules. To obtain a more accurate dependency graph the user can apply the detailed conflict detection that we described in Section 4.9.



```
preprocessor
Files RuleSet Graphs Help
[TRIGGER GRAPH
comprule1 : comprule2  comprule3
comprule2 :
comprule3 :
insrule1 :
insrule2 : insrule1
insrule3 :
insrule4 : comprule1  insrule5
insrule5 : comprule1
emptemp1 :
emptemp2 :

NO CYCLE IN TRIGGER GRAPH
```

Figure 5.9: Trigger Graph

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we describe how to construct trigger and dependency graphs using class and rule declarations of the Sentinel active OODBMS. The work we have done is the essential part of static analysis of active rules. Once we have trigger graph of an active rule set, we can check for termination by performing cycle detection on that graph. No cycle indicates that termination is guaranteed. Confluent rule execution can also be achieved by processing trigger and dependency graphs. Another important feature of our work is that we handle static analysis of active rules in an OODBMS environment. Static analysis algorithms are conservative, meaning that they either guarantee that a set of rules terminate, is confluent, or say that they may not terminate, or may not be confluent. Therefore, we allow the user to provide additional information to be used in the construction of dependency graphs in order to perform a more precise analysis of rules. We also provide a user interface of the preprocessor that constructs trigger and dependency graphs. Our work is easily adaptable to other systems. By appropriate changes to the processing of declarations and construction of event lists from events, it can be used in other active OODBMSs. Our work can accommodate incremental rule analysis if the modifications are provided in the format specified.

6.2 Possible Improvements on Implementation

An object accessed in the event part of a rule is more likely to be accessed in condition and action parts. Similarly, an object that is accessed in the condition part of a rule is more likely to be accessed in the action part. So we can reduce the cost of determining the type of an object if we use an additional list field in the rule nodes which contains information about the types of objects used in that rule. This list can be searched every time the type of an object is needed. We can make insertions to this list while constructing the trigger graph, and use it while constructing the dependency graph.

In detecting the conflicts, we use linked lists. Instead of this, we can insert the variables into an extensible hash table. Each table entry can point to a linked list where each node contains information about method name, rulename and whether it represents an event, condition, or an action. We can test the conflicts while inserting nodes into that data structure. For conflicts between instance level rules (ILR) and class level rules (CLR), we can use a linked list for each class type keeping information about variables of that class. Since we also need to keep class names, a graph structure will be required to represent the equivalent of variable list in our implementation.

As far as we know, there is no work on analysis of active database rules that can give us such information as the average number of objects an ILR uses, the average number of ILR an object is accessed, and the average number of CLR per class. Once we implement the system, we can make such analysis and improve the performance of the system accordingly.

Appendix A

Sample Input Files

A.1 Original Program

```
#include "Sentinel.h"
#include "Demo_company.h"
#include "Demo_employee.h"
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <time.h>

int cond_emp_or(L_OF_L_LIST *n1_list);
void action_emp_or(L_OF_L_LIST *n1_list);

// Other Function Prototypes

employee *mike, *joe, *jane;
company *newco, *computerco;

main ( int argc, char** argv)
{
    OpenOODB->beginTransaction();
```

```

mike = new employee("mike",40000,4);
joe = new employee("joe",40000,4);
jane = new employee("jane",40000,4);

newco = new company("newco",40, 200000.00,5000.00,30,5);
computerco = new company("computerco",40, 200000.00,5000.00,30,5);

//INSTANCE EVENT definition
event end(e_mike:mike) void setsalary(float x);
event end(e_mike2:mike) void increasesalary(int percent);
event end(e_jane:jane) void setrank(int y);
event compmike = e_mike >> e_mike2; // SEQ operator
event ite = [22:54:00/07/09/96];
event e_plus = e_jane + [1 year]; //PLUS operator
event end(enewco:newco) void seteeficiency(int eff);
event end(enewco2:newco) void settaxrate(int percent);
event end(enewco3:newco) void incrementemployee();
event e_Astar = A*(enewco,enewco3 ,enewco2);
event end(computerco:newco) void seteeficiency(int eff);
event end(computerco2:newco) void settaxrate(int percent);
event end(computerco3:newco) void incrementemployee();
event e_A = A(computerco,computerco3 ,computerco2);

event et1=[09:00:00/01/01/97];
event et2=[09:00:00/01/01/99];
event etemp1 = P(et1,[1 year],et2); //operator P
event etemp2 = !(e_mike2,et1,et2); //operator NOT

//RULE definition
rule insrule1[compmike,inscond1,insact1,RECENT];
rule insrule2[e_jane,inscond2,insact2,RECENT];
rule insrule3[e_plus,inscond3,insact3,RECENT];
rule insrule4[e_Astar,inscond4,insact4,RECENT];
rule insrule5[e_A,inscond5,insact5,RECENT];

```

```
rule emptemp1[etemp1, condtemp1, actemp1, RECENT];
rule emptemp2[etemp2, condtemp2, actemp2, RECENT];

mike->setsalary(50000.00);
newco->settaxrate(35);

OpenOODB->commitTransaction();
delete oodb;
}
int cond_emp_or(L_OF_L_LIST *n1_list)
{
    if
        (mike->getsalary() > 100000)
        return 1;
}
void action_emp_or(L_OF_L_LIST *n1_list)
{
    mike->salarybound();
}

// Other Function Declarations
```

A.2 Sample Class and CLR Declaration

A.2.1 Demo_employee.sh

```
#include "Sentinel.h"
class REACTIVE;

class employee: public REACTIVE
{
private:
    float salary;
```

```

int rank;

public:
    employee(float initsal, int initrank);
    int getsalary();
    void increaserank();
    void setrank( int y);

    //PRIMITIVE EVENT definition
    event end(e1) void setsalary(float x);
    event end(e2) void increasesalary(int percent);
    event end(e3) void salarybound();
    event end(e4) int getrank();

    //COMPOSITE EVENT definition
    event emp_or = e1 | e2;          //operator OR

    //RULE definition
    rule emprule1[emp_or, cond_emp_or, action_emp_or, RECENT];
    rule emprule2[e3, cond_e3, action_e3, RECENT];
    rule emprule3[e3, cond_e3_2, action_e3_2, RECENT];
};

```

A.2.2 Demo_employee.c

```

#include "Demo_employee.h"
#include <string.h>
#include <stdlib.h>

employee::employee(char *n1,float initsal, int initrank):REACTIVE("employee")
{
    name = strdup(n1);
    salary = initsal;
    rank = initrank;
}

```

```
}
int employee::getsalary()
{
    return salary;
}
void employee::setsalary(float x)
{
    salary = x;
}
void employee::increasesalary(int percent)
{
    salary = salary * (100 + percent ) / 100;
}
void employee::salarybound()
{
    salary = 100000.00;
}
void employee::setrank(int y)
{
    rank = y;
}
int employee::getrank()
{
    return rank;
}
```

A.2.3 Demo_company.sh

```
#include "Sentinel.h"
class REACTIVE;

class company: public REACTIVE
{
private:
```

```

    int numberofemployees;
    float sales;
    float salesperemployee;
    int taxrate;
    int efficiency;

public:
    company( int num, float sal, float spe, int tax, int eff);
    float getsales();
    int getemployees();
    float getsalesperemployee();
    void setefficiency(int eff);

//PRIMITIVE EVENT definition
    event end(ce1) void incrementemployee();
    event end(ce2) void decrementemployee();
    event end(ce3) void updatesalesperemployee();
    event end(ce4) void settaxrate(int percent);

//COMPOSITE EVENT definition
    event ce_or = ce1 | ce2;    //operator OR
    event ctemp1 = P*(ce4,[1 month],ce3); //p_star operator

//RULE definition
    rule comrule1[ce_or,cond_ce_or, action_ce_or, RECENT];
    rule comrule2[ce3, cond_ce3, action_ce3, RECENT];
    rule comrule3[ctemp1, cond_ce4, action_ce4, RECENT];
};

```

A.2.4 Demo_company.c

```

#include "Demo_company.h"
#include <string.h>
#include <stdlib.h>

```

```
company::company(char *n1, int num, float sal, float spe, int tax,
                 int eff):REACTIVE("company")
{
    comp = strdup(n1);
    numberofemployees = num;
    sales = sal;
    salesperemployee = spe;
    taxrate = tax;
    efficiency = eff;
}
void company::incrementemployee()
{
    numberofemployees++;
}
void company::decrementemployee()
{
    numberofemployees--;
}
void company::updatesalesperemployee()
{
    salesperemployee = sales/numberofemployees;
}
float company::getsales()
{
    return sales;
}
int company::getemployees()
{
    return numberofemployees;
}
void company::settaxrate(int percent)
{
    taxrate = percent;
}
```

```
float company::getsalesperemployee()
{
    return salesperemployee;
}
void company::setefficiency(int eff)
{
    efficiency = eff;
}
```

A.3 Snoop Preprocessed File Demo.c

```
#include "Sentinel.h"
#include "Demo_company.h"
#include "Demo_employee.h"
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <time.h>

// Function Prototypes

employee *mike, *joe, *jane;
company *newco, *computerco;

#include "Sentinel.fn"
#include <thread.h>
#include <synch.h>
extern char APP_ID[128];
extern void Init_call();
int GFLAG = 0;
int PID;
char *APP_NAME;
char *HOST;
```



```

char *GLOBAL_EVENT_FILE;
void load_dynamic_rules(char *host, char *port, char *sg)
{}
main ( int argc, char** argv)
{load_dynamic_rules("juice", "8600", "8604");
PRIMITIVE *company_ce1 = new PRIMITIVE( "company_ce1", "company",
    "end","void incrementemployee()");
PRIMITIVE *company_ce2 = new PRIMITIVE( "company_ce2", "company",
    "end","void decrementemployee()");
PRIMITIVE *company_ce3 = new PRIMITIVE( "company_ce3", "company",
    "end","void updatesalesperemployee()");
PRIMITIVE *company_ce4 = new PRIMITIVE( "company_ce4", "company",
    "end","void settaxrate(int percent)");
OR *company_ce_or = new OR("company_ce_or",company_ce1, company_ce2);
PRIMITIVE *company_rel1 = new PRIMITIVE( "company_rel1", "TEMPORAL",
    "", "1 month");
P_star *company_ctempe1 = new P_star("company_ctempe1",company_ce4,
    company_rel1,company_ce3);
RULE *comprule1 = new RULE("comprule1", company_ce_or, cond_ce_or,
    action_ce_or, RECENT);
RULE *comprule2 = new RULE("comprule2", company_ce3, cond_ce3,
    action_ce3, RECENT);
RULE *comprule3 = new RULE("comprule3", company_ctempe1, cond_ce4,
    action_ce4, RECENT);

Notify(NULL,"OODB","beginT","begin",system_list);
OpenOODB->beginTransaction();
Notify(NULL,"OODB","beginT","end",system_list);

    mike = new employee("mike",40000,4);
    joe = new employee("joe",40000,4);
    jane = new employee("jane",40000,4);

    newco = new company("newco",40, 200000.00,5000.00,30,5);
    computerco = new company("computerco",40, 200000.00,5000.00,30,5);

```

```

PRIMITIVE *e_mike = new PRIMITIVE( "e_mike", mike, "end",
    "void setsalary(float x)");
PRIMITIVE *e_mike2 = new PRIMITIVE( "e_mike2", mike, "end",
    "void increasesalary(int percent)");
PRIMITIVE *e_jane = new PRIMITIVE( "e_jane", jane, "end",
    "void setrank(int y)");
SEQ *compmike = new SEQ("compmike",e_mike, e_mike2);
PRIMITIVE *ite = new PRIMITIVE("ite", "TEMPORAL", "", "22:54:00/07/09/96");
PRIMITIVE *rel1 = new PRIMITIVE( "rel1", "TEMPORAL", "", "1 year");
PLUS *e_plus = new PLUS("e_plus",e_jane,rel1);
PRIMITIVE *enewco = new PRIMITIVE( "enewco", newco, "end",
    "void seteeficiency(int eff)");
PRIMITIVE *enewco2 = new PRIMITIVE( "enewco2", newco, "end",
    "void settaxrate(int percent)");
PRIMITIVE *enewco3 = new PRIMITIVE( "enewco3", newco, "end",
    "void incrementemployee()");
A_star *e_Astar = new A_star("e_Astar",enewco,enewco3,enewco2);
PRIMITIVE *computerco = new PRIMITIVE( "computerco", newco, "end",
    "void seteeficiency(int eff)");
PRIMITIVE *computerco2 = new PRIMITIVE( "computerco2", newco, "end",
    "void settaxrate(int percent)");
PRIMITIVE *computerco3 = new PRIMITIVE( "computerco3", newco, "end",
    "void incrementemployee()");
A *e_A = new A("e_A",computerco,computerco3,computerco2);
PRIMITIVE *et1 = new PRIMITIVE("et1", "TEMPORAL", "", "09:00:00/01/01/97");
PRIMITIVE *et2 = new PRIMITIVE("et2", "TEMPORAL", "", "09:00:00/01/01/99");
PRIMITIVE *rel2 = new PRIMITIVE( "rel2", "TEMPORAL", "", "1 year");
P *etemp1 = new P("etemp1",et1,rel2,et2);
NOT *etemp2 = new NOT("etemp2",e_mike2,et1,et2);
RULE *insrule1 = new RULE("insrule1", compmike, inscond1, insact1, RECENT);
RULE *insrule2 = new RULE("insrule2", e_jane, inscond2, insact2, RECENT);
RULE *insrule3 = new RULE("insrule3", e_plus, inscond3, insact3, RECENT);
RULE *insrule4 = new RULE("insrule4", e_Astar, inscond4, insact4, RECENT);
RULE *insrule5 = new RULE("insrule5", e_A, inscond5, insact5, RECENT);

```

```
RULE *emptemp1 = new RULE("emptemp1", etemp1, condtemp1, actemp1, RECENT);
RULE *emptemp2 = new RULE("emptemp2", etemp2, condtemp2, actemp2, RECENT);
```

```
    mike->setsalary(50000.00);
    newco->settaxrate(35);
```

```
Notify(NULL,"OODB","commitT","begin",system_list);
OpenOODB->commitTransaction();
Notify(NULL,"OODB","commitT","end",system_list);
```

```
    delete oodb;
}
int cond_emp_or(L_OF_L_LIST *n1_list)
{
    if
        (mike->getsalary() > 100000)
        return 1;
}
void action_emp_or(L_OF_L_LIST *n1_list)
{
    mike->salarybound();
}
int cond_e3(L_OF_L_LIST *n1_list)
{
    if
        (mike->getrank() >5)
        return 1;
}
void action_e3(L_OF_L_LIST *n1_list)
{
    mike->increasesalary(10);
}
int cond_e3_2(L_OF_L_LIST *n1_list)
{
    if
```

```
(joe->getrank() >7)
return 1;
}
void action_e3_2(L_OF_L_LIST *n1_list)
{
    joe->increasesalary(20);
}
int cond_ce_or(L_OF_L_LIST *n1_list)
{
    return(1);
}
void action_ce_or(L_OF_L_LIST *n1_list)
{
    newco->updatesalesperemployee();
}
int cond_ce3(L_OF_L_LIST *n1_list)
{
    if
        (newco->getsalesperemployee() >200000)
        return 1;
}
void action_ce3(L_OF_L_LIST *n1_list)
{
    newco->setefficiency(10) ;
}
int inscond1(L_OF_L_LIST *n1_list)
{
    if
        (mike->getsalary() > joe->getsalary())
        return 1;
}
void insact1(L_OF_L_LIST *n1_list)
{
    joe->increasesalary(10);
}
```

```
int inscond2(L_OF_L_LIST *n1_list)
{
    if
        (jane->getrank() > mike->getrank())
        return 1;
}
void insact2(L_OF_L_LIST *n1_list)
{
    mike->increasesalary(1);
}
int condtemp1(L_OF_L_LIST *n1_list)
{
    if
        (jane->getsalary() <200000)
        return 1;
}
void actemp1(L_OF_L_LIST *n1_list)
{
    jane->increasesalary(10);
}
int condtemp2(L_OF_L_LIST *n1_list)
{
    if
        (mike->getsalary() <100000)
        return 1;
}
void actemp2(L_OF_L_LIST *n1_list)
{
    mike->setsalary(100000);
}
int inscond3(L_OF_L_LIST *n1_list)
{
    if (jane->getsalary()<100000)
        return 1;
}
```

```
void insact3(L_OF_L_LIST *n1_list)
{
    jane->setsalary(100000);
}
int cond_ce4(L_OF_L_LIST *n1_list)
{
    return 1;
}
void action_ce4(L_OF_L_LIST *n1_list)
{
    newco->setefficiency(8);
}
int inscond4(L_OF_L_LIST *n1_list)
{
    if (newco->getsalesperemployee()>100000)
        return 1;
}
void insact4(L_OF_L_LIST *n1_list)
{
    newco->incrementemployee();
}
int inscond5(L_OF_L_LIST *n1_list)
{
    if (computerco->getsalesperemployee()>100000)
        return 1;
}
void insact5(L_OF_L_LIST *n1_list)
{
    computerco->incrementemployee();
}
```

Bibliography

- [AHW95] Alexander Aiken, Joseph M. Hellerstein, and Jennifer Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM Transactions on Database Systems*, 20(1):3–41, March 1995.
- [AWH92] Alexander Aiken, Jennifer Widom, and Joseph M. Hellerstein. Behavior of database production rules: Termination, confluence, and observable determinism. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, pages 59–68, San Diego, California, June 1992.
- [BCFP96] E. Baralis, S. Ceri, P. Fraternali, and S. Paraboschi. A support environment for active rule design. *Journal of Intelligent Information Systems*, 7(2):129–150, October 1996.
- [BCP96] Elene Baralis, Stefano Ceri, and Stefano Paraboschi. Modularization techniques for active rules design. *ACM Transactions on Database Systems*, 21(1):1–29, March 1996.
- [BCW93] Elena Baralis, Stefano Ceri, and Jennifer Widom. Better termination analysis for active databases. In *1st International Workshop on Rules in Database Systems*, pages 163–179, September 1993.
- [BGB95] Emmanuel Benazet, Herve Guehl, and Mokrane Bouzeghaub. Vital: A visual tool for analysis of rule behavior in active databases. In *2nd International Workshop on Rules in Database Systems*, pages 182–196, September 1995.

- [BW95] Elena Baralis and Jennifer Widom. Better static rule analysis for active database systems. Technical report, Stanford Univeristy, December 1995.
- [CM93] Sharma Chakravarthy and Deepak Mishra. Snoop: An expressive event specification language for active databases. Technical Report UF-CIS-TR-93-007, University of Florida, 1993.
- [CR96] Stefano Ceri and Raghu Ramakrishnan. Rules in database systems. *ACM Computing Surveys*, 28(1):109–111, March 1996.
- [CTZ95] S. Chakravarthy, Z. Tamizuddin, and J. Zhou. A visualization and explanation tool for debugging eca rules in active database. In *2nd International Workshop on Rules in Database Systems*, pages 197–212, September 1995.
- [Day88] Umeshwar Dayal. Active database management systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pages 150–169, Jerusalem, June 1988.
- [DJP93] Oscar Diaz, Arturo Jaime, and Norman W. Paton. Dear: a debugger for active rules in an object-oriented context. In *1st International Workshop on Rules in Database Systems*, pages 180–193, September 1993.
- [JUD96] Alexander Jahne, Susan D. Urban, and Susanne W. Dietrich. Peard: A prototype environment for active rule debugging. *Journal of Intelligent Information Systems*, 7(2):111–128, October 1996.
- [KC95] S-K. Kim and S. Chakravarthy. A confluent rule execution model for active databases. Technical Report UF-CIS-TR-95-032, University of Florida, 1995.
- [Lee96] Hyesun Lee. Support for temporal events in sentinel: Design, implementation and preprocessing. Master's thesis, University of Florida, 1996.
- [Lia97] Hui Liao. Global events in sentinel: Design and implementation of a global event detector. Master's thesis, University of Florida, 1997.

- [Ost96] Leon Osterweil. Strategic directions in software quality. *ACM Computing Surveys*, 28(4):738–750, December 1996.
- [Tex93] Texas Instruments. *Open OODB C++ API User Manual*, 1993.
- [vdVS93] Leonie van der Voort and Arno Siebes. Termination and confluence of rule execution. Technical Report CS-R9309, Centrum voor Wiskunde en Informatica, Netherlands, 1993.
- [WH95] Thomas Weik and Andreas Heuer. An algorithm for the analysis of termination of large trigger sets in an oodbms. In *Proceedings of the First International Workshop on Active and Real-Time Database Systems*, pages 170–189, Sweden, June 1995.
- [Wid96] Jennifer Widom. The starburst active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):583–595, August 1996.