A GENERALIZED ACTIVE AGENT SYSTEM FOR EXTENDING THE ACTIVE CAPABILITIES OF A RDBMS

By

YOUNGHUN KIM

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
2000

To my parents and HyeKyung Kim

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Sharma Chakravarthy, for his continuous advice, guidance, support in this research work and for giving me the opportunity to work on this good topic. His help in making me more knowledgable has been priceless. I am grateful to Dr. Sumi Halal and Dr. Joachim Hammer for serving as my committee members.

I would like to thank John Bowers and Ardiniece Caudle for their help and taking good care of graduate students.

I would like to thank Sharon Grant for maintaining a well-administered research environment and being so helpful at times of need.

I really thank SeokWon Yang, Mahesh Dudgikar, Hongen Zhang and Zecong Song for their technical help and fruitful discussions. I will never forget their invaluable help and the life in the Sentinel group. Also, I would like to thank all my friends for their support and encouragement.

This work was supported in part by the Office of Naval Research and the SPAWAR System Center-San Diego, and by the Rome Laboratory, DARPA.

I would like to take this opportunity to thank my family for their endless love. Without their support, this work would not have been possible. I am cordially grateful to HyeKyung Kim for her patience and endless love. Her trustfulness and encouragement made me brave when I was weary.

Finally, and supremely, I would sincerely love to thank God for His love. I could have accomplished absolutely nothing apart from His will.

TABLE OF CONTENTS

LIST OF TABLES

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 GENERALIZED ACTIVE AGENT SYSTEM
FOR EXTENDING THE ACTIVE CAPABILITIES OF A RDBMS

By

YoungHun Kim

December 2000

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Science and Engineering

Database management systems (DBMSs) have evolved remarkably to meet the requirements of emerging applications. One of the requirements is to satisfy the needs of many applications that require a timely response to situations. Active database systems have been proposed as a data management paradigm to represent real-world situations as part of the database, and monitor and react to them automatically without user or application intervention. Event Condition Action (ECA) rules are used to make a database system active. In Relational Database System (RDBMS), ECA rules can be defined by triggers. Events can be modifications of tables. Conditions can be the checking of constraints. Rule part is defined in use of triggers. Even if a number of research prototypes of active database systems have been built, ECA rule capability in RDBMS is very limited. This thesis addresses the problem of turning a traditional database management system into a true active database system without changing the semantics of the existing system and provides a more generalized event definition mechanism and extends the active capabilities of RDBMS to a fully active DBMS. The advantages of this approach are very obvious. First, transparency is guaranteed. That is, we can add the active capability without changing the client programs. Second, every underlying functionality of the RDBMS is maintained. Third, ECA rules can be made persistent using the native database functionality. Our approach is a generalized one because an ECA Agent can connect any RDBMS.

Active database semantics can be supported on an existing SQL Server (we use Oracle SQL Server as the test SQL Server) by the ECA Agent inserted between the SQL Server and its clients. ECA rules are completely supported through the ECA Agent without changing applications in the SQL Server. Both primitive and composite events can be detected in the ECA Agent and actions are invoked in SQL Server. All events are persistent in RDBMS. The Java Local Event Detector (Java LED) is used to notify and detect both primitive events and composite events. The ECA Agent uses Java Database Connectivity (JDBC) to connect to the SQL server. The architecture of the ECA Agent and its implementation are elaborated in this thesis.

CHAPTER 1
INTRODUCTION

Traditional database management systems (DBMSs) are referred to as passive since any situation to be monitored over the state of the database has to be done explicitly by the user or application by executing queries or transactions. This traditional view makes DBMS limited as an information repository of data. During the last decade, the need for making a DBMS capable of reacting to specific situations without user or application intervention has been proposed. As a result, DBMSs have evolved to meet the diverging requirements of several classes of applications. Most new developments in database technology represent real-world situations as a part of the database to monitor and react to them automatically without user or application intervention. An Active DBMS can continuously monitor situations to initiate appropriate actions in response to database updates and the occurrence of particular states automatically.

A frequently mentioned example to distinguish the difference between a passive DBMS and active one is a hospital environment. If an electrocardiogram is being recorded in a database for an intensive care unit patient, a doctor or nurse is responsible for checking the data values over a period to determine an emergency state of the patient in the view of the passive database. However, in an active database, DBMS will alert the doctor when it detects any state of emergency based on the set of rules triggered as a result of the updates. The monitoring of situations can be done by defining ECA rules on events of interest. ECA rules consist of three components: An event, a condition, and an action. An event is an indicator of a happening which can be either simple or complex. Most events are state changes produced by database operations (e.g., a method invocation or a database update). The condition can be a simple or a complex query based on the existing database states and set of data objects, transitions between states of objects, or event trends and historical data. Actions specify the operations to be performed when an event has occurred and the condition evaluates to true. After ECA rules are specified declaratively, it is the responsibility of the DBMS to monitor the situation and trigger the rules when the condition is satisfied. There have been a number of prototypes of active database systems that have been developed, such as HiPAC [CHA89], Sentinel [CHA93], Starburst [WID96], Postgres [STO91], etc. The prototype prototypes listed are representative of a much larger number of active database systems that have been designed using an integrated approach. That is, the production rule components of active database systems have typically been integrated directly into the kernel of the DBMSs. The implementation of an integrated approach requires access to the internals of a DBMS into which the active capability is being integrated. This requirement of access to internal code makes the cost high and integration time long as well. This thesis starts with the fact that no commercial DBMS supports full active capability although a lot of active database technologies are proposed and with the question whether it is possible to turn a commercial DBMS into a true active DBMS without making any changes to the underlying system [LEE 98]. This thesis considers the possibility of adding the active capability to Oracle DBMS. This thesis introduces an approach which adds an agent between the Oracle SQL Server and the client to provide full ECA functionality to the user.

1.1 Advantage of Providing the Active Capability to the RDBMS

There are many reasons why the use of a generalized architecture is appropriate for adding full active capability transparently:

- Maintenance of System functionality: None of the existing RDBMS' functionality would be lost.
- Scalability: The architecture would be more scaleable.
- Portability: Once an ECA Agent has been developed, it may be ported to other DBMSs.
- Transparency: The clients are unaware of the agent introduced between the client and the server.

1.2 Reasons for Choosing Oracle as Test DBMS

Oracle has many features which facilitate the implementation of an active component on top of an existing database. The features that have been exploited in our approach are the following:

- Oracle DBMS provides relatively simple active rules using triggers although it is generally limited compared with the active database prototypes such as Sentinel, Ode, etc.
- Oracle DBMS offers Cursors. A cursor is a handle or name for a private SQL area – an area in memory in which a parsed statement and other information for processing the statement are kept. We use the feature of Oracle to issue recursive SQL statements. Recursive calls are made for those recursive cursors. These recursive cursors use a shared SQL area.

- Oracle is an open architecture and can be connected to any application using many connection tools.

Another advantage of Oracle is that its Open Client database API is widely accepted in industry, and many products and applications support the Oracle database library.

1.3 The Contributions

The contributions of this thesis are the following :

- This thesis presents a generalized approach that significantly extends the Oracle ECA functionality:
    1. A client can create multiple triggers on the same event.
    2. A client can create composite events and triggers on them.
    3. Previously defined events can be reused.
    4. Triggers associated with primitive or composite events can be dropped.
    5. Once events are created, they can be persisted into the database system.
    6. All primitive events and composite events can be detected, and actions are invoked in SQL server.
- This thesis discusses alternative approaches and shows why this generalized approach is appropriate.
- This thesis presents the architecture of a generalized ECA Agent system and how an ECA Agent is implemented.

Related work is discussed in chapter 2. The technology used to design ECA Agent system is described in chapter 3. Chapter 4 presents how the ECA Agent has been implemented. Chapter 5 and chapter 6 respectively explain the implementation of primitive events and implementation of composite events. Chapter 7 includes the conclusion of this thesis and future work.

CHAPTER 2
OVERVIEW OF RELATED WORK

2.1 Sentinel

Sentinel is an Object Oriented Active Database System. It integrates ECA rules capability into the kernel of Open OODB. The goal of integration is the enhancement of Open OODB from a passive OODB to an active OODB by incorporating primitive event detection and support for nested transactions as part of its kernel. In addition, it supports composite event detection, and rule management as separate modules. It uses the Open OODB Toolkit (from Texas Instruments) as the underlying platform. Sentinel has support for events [CHA94a][CHA94b][CHA94c]:

- Primitive and composite event detection: Any method of any object class can be a primitive event. Before and after variants of method invocation are permitted as events. Composite events are formed by applying a set of operators such as AND, OR, SEQ, etc.
- Parameter contexts: The processing of a composite event entails not only its detection, but also the computation of the parameters associated with the composite event. Parameter contexts are motivated by a careful analysis of several classes of applications.
- Online and batch detection of events: The composite event detector needs to support detection of events as they happen (online) when it is coupled to an application (in batch mode).

- Inter-application (global) events: A global event detector (or GED) provides support for composite events whose constituent events come from different applications.

Sentinel has support for rules [CHA94a][CHA94b][CHA94c]:

- Multiple rules: An event can trigger several rules.
- Nested/cascading rules: Rule actions can raise events which trigger other rules.
- Coupling mode: Coupling modes refers to the execution point of condition/action pair relative to the event.
- Rule scheduling: In the view of multiple rules and nested execution, the architecture needs to support prioritized serial execution of rule, concurrent execution of all rules, or a combination of the two.

2.2 Ode

Ode provides active behavior by incorporating rules in the form of constraints and triggers. Both constraints and triggers consist of a condition and an action, and are defined within a class definition. Events in Ode are implicit and are considered as the disjunction of all non-constant 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 [GEH91]. 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 separate transactions after the commit (not necessarily immediately after) of the transaction firing them. More recently Ode had proposed a language for specifying composite events. It specifies composite events using a set of operators; events are declared with in 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.

2.3 ADAM

ADAM [DIA91] is an active OODB implemented in PROLOG. It focuses on providing a uniform approach to the treatment of rules in an object-oriented environment. Rules and events in ADAM cannot be specified as part of an object class specification. Both 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-enabled 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 rules condition and action respectively. Rule operations are implemented as class methods. Only primitive events are supported.

2.4 Starburst

The Starburst system is a prototype extensible relational DBMS developed at IBM’s Almaden Prototype Center [WID96]. Starburst’s extensibility allows the database system to be customized for advanced and non-traditional database applications. One of Starburst’s extension is an integrated active database rule processing facility called the Starburst Rule system.

The Starburst rule language differs from most of the other active database rule languages in that it is based on permitting an execution semantics that is both cleanly defined and flexible. The implementation of the Starburst Rule System was completed rapidly and relies heavily on the extensibility features of Starburst. The Starburst rule processor differs from most other active database rule systems in that it is completely implemented, and it is fully integrated into all aspects of database processing, including query and transaction processing, concurrency control, rollback recovery, error handling, and authorization.

2.5 ACOOD

ACOOD is a prototype based on the ONTOS^{TM} object-oriented DBMS which has C++ as its base language. It was developed by Mikael Berndtsson and his prototype group at the University of Skovde [BER91] and had been addressed in several reports including [BER92] and [BER94].

ACOOD uses a layered approach to develop an active DBMS which means ACOOD is put on the top of ONTOS™ and can use the functionality of this DBMS. ACOOD provides the DBMS with active functionality. An application uses either the ONTOS™ client interface for traditional database operations or the ACOOD interface to provide active functionality. Figure 2.1 shows the architecture of ACOOD and its interface.

ACOOD uses the concept of ECA rules. Events and rules are represented as first class objects in ACOOD so that ACOOD supports runtime management of events and rules.

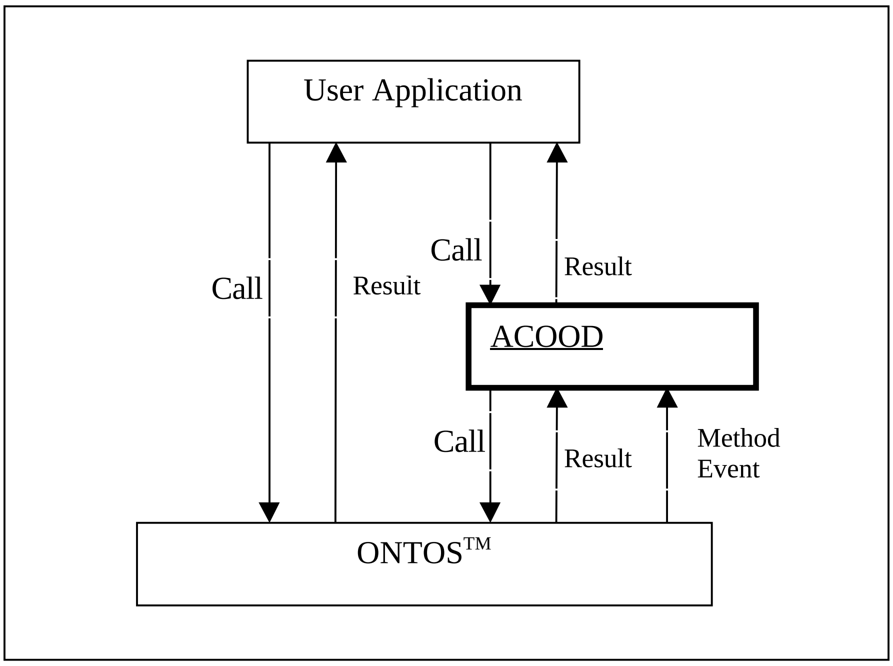


Figure 2.1 ACOOD Architecture and Interactions

2.6 The GATEWAY Approach

The Gateway approach is implemented in [VAN96]. The Gateway architecture is a layered API approach. Figure 2.2 shows the Gateway architecture. The layer of the Gateway approach is transparent to the client. Open Server (ECA Server in the figure) is used for this approach. The ECA Server receives a requirement and connects the SQL Server and sends all information to the SQL Server. The SQL Server authenticates the client and creates the connection. After connection is made, the Gateway takes control of the client process.

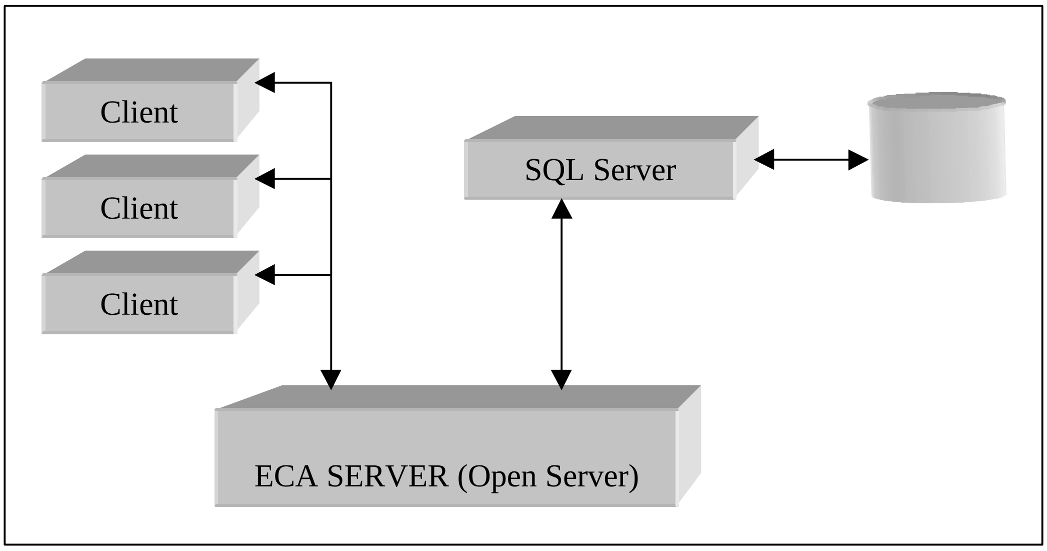


Figure 2.2 The Gateway Architecture

The ECA Server preprocesses the SQL language and procedure requests from the client. The coupling mode of events is processed to determine the execution points of events in the ECA Server. The advantage of this architecture is the following:

- According to the coupling mode, both immediate and deferred trigger semantics are implemented.

- Parallel execution of ECA Server and SQL Server is possible.
- Events are generated in the correct order.
- Events are not lost.
- Performance will be enhanced.

CHAPTER 3
DESIGN ISSUES OF THE ECA AGENT

This chapter discusses the design issues of the ECA Agent. They include the architecture of an ECA Agent, the Java LED, and the SNOOP preprocessor. This chapter shows how they are related to each other and the design an ECA Agent.

3.1 The Architecture of the ECA Agent

An ECA Agent is a general method in the sense that it can be used with any kind of relational DBMSs such as Oracle, DB2, Sybase, Informix, etc. In this section, we give the general design of an ECA Agent as it is shown in Figure 3.1. An ECA Agent provides the client interface. The ECA Agent (called the ECA Server) works between multiple clients and the SQL server. The message from clients is sent to the ECA Agent and then the ECA Agent sends it to SQL server. The Result from the SQL Server is returned to the client via an ECA Agent.

The ECA Agent is a multithreaded program. It is placed between clients and the SQL Server so that the SQL Server can provide active capabilities with full user transparency.

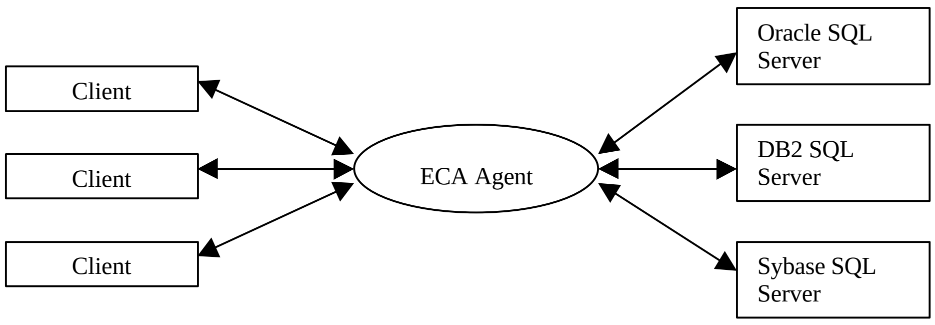


Figure 3.1 General ECA Agent System

From the point of the user, the ECA Agent works as a virtual active SQL Server. Not only can it provide all the native functions of a relational DBMS, but also it provides all the ECA active functions. Figure 3.2 shows the architecture of the ECA Agent system.

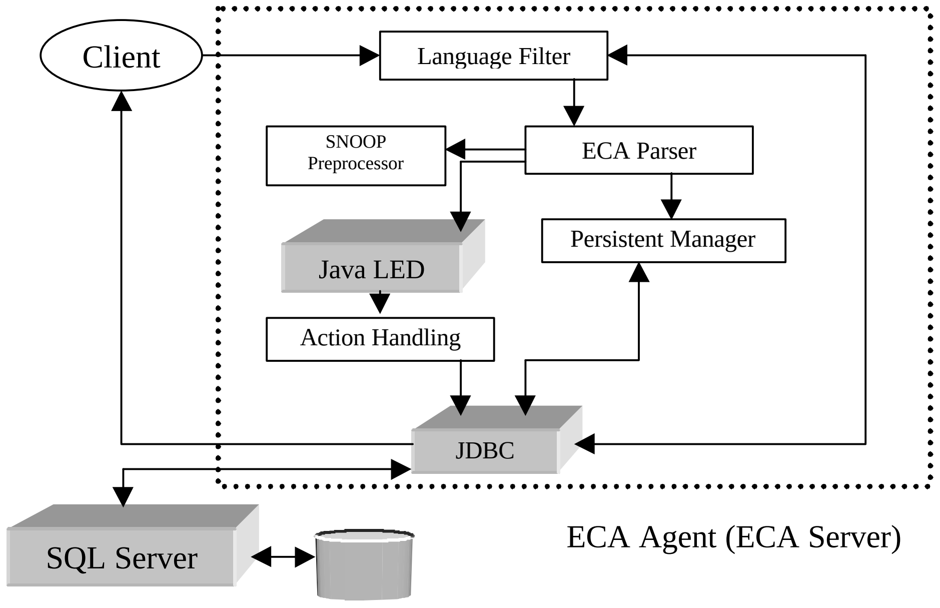


Figure 3.2 The Architecture of ECA Agent System

The following functional modules provide active capabilities:

- Language Filter: First, all client requests are sent to the Language Filter. The Language Filter filters the request to determine if it is an ECA command (extended SQL command) or a pure SQL command. If it is an ECA command, it is sent to the ECA Parser. If it is an SQL command, it is sent to the SQL Server through the Java database connection (JDBC).

- ECA Parser: An ECA command from the Language Filter is scanned and parsed. If there are no syntax errors, the parser generates the corresponding events and rules which can be detected by the Java LED. The events and rules are sent to the Persistent Manager by the Parser.
- Java Local Event Detector: In a relational DBMS, primitive events can only be detected by triggers. We use the Java LED to detect and execute the rules (condition/action pair) associated with composite events.
- Persistent Manager: All events and rules need to be stored into the relational DBMS for subsequent use. The Persistent Manager stores all ECA information into system tables in a relational DBMS. The Persistent Manager restores the needed events and rules from these tables when the ECA Agent starts or recovers.

  The system tables include:

  1) SysPrimitiveEvent: This table stores primitive events.
  2) SysCompositeEvent: This table stores information about composite events.
  3) SysECATrigger: Every trigger in system is stored in this table.

- Java Database Connectivity (JDBC): JDBC is used to connect the ECA Agent to the SQL Server. Clients can request not only relational SQL commands but also user defined commands through JDBC, and JDBC gets values from the SQL server which are sent back to clients.

- Action Handling: When events occur, the related action defined for the event need to be executed. Actions can be the modification of a system table by SQL operation such as insertion, deletion, or update.

3.2 Java Local Event Detector (Java LED)

The Java LED is used to detect composite events. The Java LED detects events in Java applications (user APIs). An event can be a method call from applications. Composite events are detected according to parameter contexts such as RECENT, CHRONICLE, CONTINOUS, and CUMULATIVE. These parameter contexts are motivated by a careful analysis of several classes of applications. The parameter contexts are classified as [KRI94]:

- Recent: In this context, only the most recent occurrence of the initiator for any event that has started the detection of that event is used.
- Chronicle: In this context, for an event occurrence, the initiator, terminator pair is unique. The oldest initiator is paired with the oldest terminator for each event.
- Continuous: In this context, each initiator of an event starts the detection of that event. A terminator event occurrence may detect one or more occurrences of the same event.
- Cumulative: In this context, all occurrences of an event type are accumulated as instances of that event until the event is detected.

Figure 3.3 shows the global event history. In the Recent context, the most recent occurrence of an event is used to detect a composite event. The composite event A will include the event instances $\{e_1^2, e_2^1, e_3^1\}$ (A is detected when e_3^1 occurs) and $\{e_1^2, e_2^2, e_3^2\}$ (A is detected again when e_3^2 occurs). In the Chronicle context, a parameter of event A is computed by using the event instances $\{e_1^1, e_2^1, e_3^1\}$. In the Continuous context, the first occurrence of A has the instances $\{e_1^1, e_2^1, e_3^1\}$. The second occurrence of A consists of the event instances $\{e_1^2, e_2^1, e_3^1\}$. In the Cumulative context, all occurrences of an event are accumulated as instances of the event A when the event is detected.

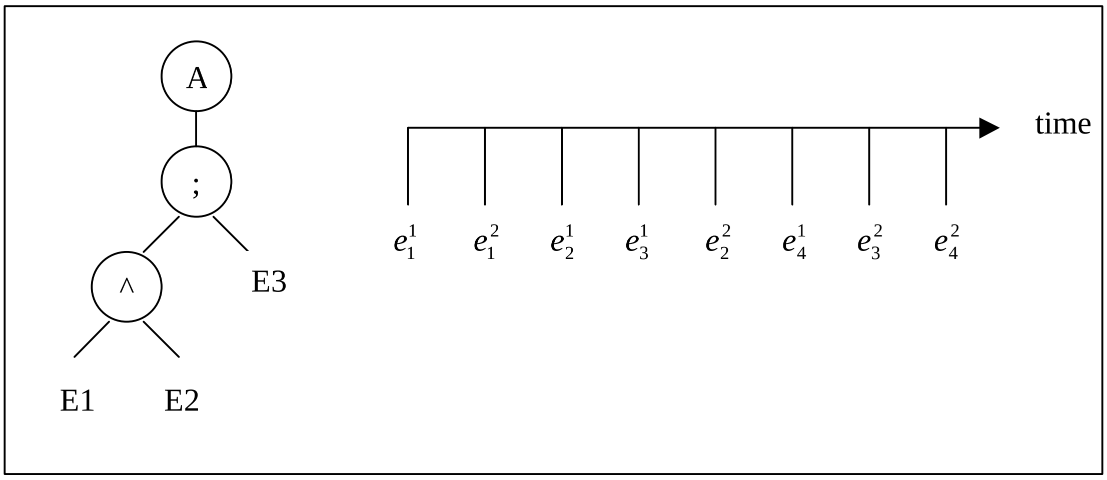


Figure 3.3 Global Event History

The concept of parameter contexts is also used to detect composite events in the ECA Agent.

3.3 Rule Scheduler

The rule scheduler is the functional module of the Java LED which controls rule execution. A rule includes a condition and action of an event. In other word, the rule scheduler controls the order of actions. For multiple rule execution, a number of sub-transactions are spawned as a part of the application process. The order of the rule execution is controlled by assigning appropriate priorities to each thread. The rule thread with the highest priority is to be executed first and rules of lower priority are to be executed after the completion of the higher priority rules. A rule is also associated with a coupling mode. Rules are to be executed based on their coupling modes. Rule created with a immediate coupling mode are to be executed at the time the event occurs whereas rules created with a deferred coupling mode are executed at a later time [DAS99].

3.4 Snoop Preprocessor

Snoop is the Event Definition Language (EDL) which the Sentinel group has developed. Snoop provides an easier way for the user to define events. The preprocessor parses user-defined event and rule specifications expressed in Snoop, and inserts appropriate Java code the application program. In other words, the Snoop preprocessor converts Snoop expressions into Java APIs to be inserted into the user program. We discuss how this is done in Section 4.4.

3.5 Java Database Connectivity (JDBC)

The JDBC is the tool to connect the Java applications to the relational DBMS. In our prototype, we use JDBC to send SQL statements from the client's interface to the relational DBMSs. The first thing we need to do is to establish the connection with the DBMS which we want to use. There are three steps to do this:

1) Loading the Drivers: Loading the driver or drivers we want to use is very simple and involves one line of code. If, for example, we want to use the Oralcle driver, the following code loads it:

   Class.forName("oracle.jdbc.driver.OracleDriver");

When we have loaded a driver, it is available for making a connection with a DBMS.

2) Making the Connection: The second step in establishing a connection is to have the appropriate driver connect to the DBMS. The following line of code illustrates the general idea:

   Connection con=DriverManager.getConnection(url,”Login”,”Password”);.

   If we want to use Oracle as a test DBMS, the url should be “jdbc:oracle:thin:@ tokyo.dbcenter.cise.ufl.edu :1521:ORCL.”

3) Creating the JDBC Statements: A Statement object is what sends the SQL statement to the DBMS. We simply create a Statement object and then execute it, supplying the appropriate execute method with the SQL statement we want to send. For a SELECT statement, the method to use is executeQuery. For statements that create or modify tables, the method to use is executeUpdate. The following line of code is used to create the Statement object ”stmt”:

   Statement stmt = con.createStatement();.

   We need to supply this statement to the method we use to execute this statement:

   Stmt.executeQuery(“select * from portfolio”);.

   We can get the result from the RDBMS using the following line of code:

   ResultSet rs= stmt.executeQuery(“select * from portfolio”);.

3.6 Interaction between the ECA Agent Modules

In this section, we discuss how each ECA Agent module works and interacts. If a user sends his request to the Language Filter. The Language Filter filters the user request to determine if the user request is a SQL command (insert, delete, update, select, etc) or it is ECA command (extended create trigger statement which includes event definition). SQL commands are directly sent to SQL Server through JDBC. ECA commands, on the other hand, are sent to the ECA Parser. The ECA Parser scans ECA command and checks if the trigger name and the event name of the event are duplicates. Then the ECA Parser generates a Java source file. For example, if the primitive event name is addStk and the user name is ykim, the Java file name is ykimaddStk.java. After the file, ykimaddStk is generated, it is compiled (we use the method exec() of java.lang.Runtime class to compile a Java source file. Refer to the CompileFile method of ConstructClass.java) and then the method, call_addStk is invoked by dynamic method call using ExecuteMethod() of CallDynamicMethod class (we use the method invoke() of ClassLoader class). The reason why we generate and compile the Java source file and invoke the method is because we have to initialize the Java LED and obtain the event handler of the event so as to detect a composite event related to the primitive event (refer to Figure 6.11). If we have another primitive event, delStk, the processing steps on the primitive event, delStk in the ECA Parser is the same as those of the primitive event, addStk. If we have the composite event, addDel (this event needs both addStk and delStk to be detected by the Java LED), the Java source file, ykimaddDel.java is generated by the

ECA Parser and compiled by the exec() method of java.lang.Runtime class. The call_addDel() method in the ykimaddDel.java file is invoked to initialize the JavaLED and obtain the event handler of the composite event, addDel in the Java LED. The only difference of file generation between primitive events and composite events is the generated file (ykimaddDel.java) for the composite event, addDel has the rule definition which includes a condition function and an action. The Persistent Manager works on system table modifications. It stores trigger information of both primitive and composite events into the system table, SysECATrigger, event information of primitive events into the system table, SysPrimitiveEvent, and event information of composite events into the system table, SysCompositeEvent (refer to chapter 4 on system tables).

CHAPTER 4
IMPLEMENTATION ISSUES OF THE ECA AGENT

This chapter discusses the implementation of the ECA Agent system. In this chapter, we elaborate on the purpose of the system tables and how they are formatted. In this prototype, the system tables store the persistent information on primitive events, composite events, and ECA Actions. Each functional module is also examined in more detail.

4.1 System Tables

4.1.1 "SysPrimitiveEvent"

The system table, SysPrimitiveEvent stores the information on primitive events. The structure of SysPrimitiveEvent is shown in Table 4.1.

Table 4.1 SysPrimitiveEvent

dbname	username	eventname	tablename	Operation	Beafoperation	timestamp	vno

The operation column includes the operation name on a relational table. In this prototype, operation names should be one of Insert, Delete, and Update. The column, beafoperation includes one of Before or After (Oracle has two kinds of operations such as a Before operation and an After operation). The TimeStamp implies the time of the event occurrence. Vno is the number of occurrences of the same event. When a primitive event occurs, this table is populated with the corresponding items of the event. For example, when the following trigger with a primitive event is defined:

[Create trigger t_addStk after insert on stock event addStk ……]

The system table, SysPrimitiveEvent contains the data shown in Table 4.2.

Table 4.2 SysPrimitiveEvent after the event, addStk

dbname	username	Eventname	tablename	Operation	Beafoperation	timestamp	vno
ORCL	ykim	addStk	stock	Insert	After	26-jun-00	0

<u>4.1.2 “SysCompositeEvent”</u>

The system table, SysCompositeEvent is used to store the information on composite events. The structure of this table is shown in Table 4.3:

Table 4.3 SysCompositeEvent

dbname	username	eventname	eventDescribe	Timestamp	Coupling	context	Priority

In Table 4.3, The coupling mode should be one of IMMEDIATE, DEFERED, and DETACHED. The context can RECENT,CHRONICLE, CONTINUOUS, or CUMULATIVE. The priority defines the priority of this composite event. The composite event definion, addDel is shown below. EventDescribe shows which primitive events are included to form this composite event and operator. A trigger with the composite event definition is shown below:

[Create trigger t_addDel event addDel = addStk ^ delStk RECENT IMMEDIATE 1 ….]

After this trigger is defined, the system table, SysCompositeEvent contains the data shown in Table 4.4.

Table 4.4 SysCompositeEvent after the event, addDel

dbname	Username	eventname	eventDescirbe	timestamp	coupling	context	priority
ORCL	Ykim	AddDel	addStk^delStk	26-jun-00	Immediate	Recent	1

4.1.3 "SysEcaTrigger"

This table is used to store the information on triggers created by the user. When users define a trigger, a check for the trigger duplication should be made by searching this table. The structure of the table, SysEcaTrigger is shown in Table 4.5.

Table 4.5 SysEcaTrigger

dbname	Username	Triggername	TriggerProc	timestamp	eventname

The column, TriggerProc in Table 4.5 contains the name of the procedure defined on a trigger. When the trigger fires, this procedure will be executed. Several different trigger can be defined on the same event as is shown in Table 4.6 below. For example, when we define the primitive event, addStk, the trigger named t_addStk is created automatically inside the ECA Agent system for the insert operation, the delete operation as shown by Table 4.6.

Table 4.6 SysEcaTrigger After the event, addStk

dbname	Username	Triggername	TriggerProc	timestamp	eventname
ORCL	Ykim	t_addStk	t_addStk_proc	26-jun-00	addStk

4.1.4 "SysContext"

The table, SysContext is used to store the occurrence number (Vno) of an event associated with a context. This table is used for keeping track of event occurrences associated with a composite event. The structure of this table is shown in Table 4.7.

Table 4.7 SysContext

EventName	Context	Vno

Suppose the event, addDel associated with the context, RECENT has occurred three times. The following Table 4.8 shows the tuples inserted into SysContext table.

Table 4.8 SysContext after addDel related to RECENT context.

eventName	Context	Vno
addDel	Recent	1
addDel	Recent	2
addDel	Recent	3

4.1.5 "EventContext"

Table EventContext is used to store information on primitive events and contexts where a composite event occurs. The structure of the EventContext table is illustrated in Table 4.9.

Table 4.9 EventContext

EventName	Context
DelStk	Recent

When the composite event, addDel occurs as the following:

[Create trigger t_addDel event addDel = addStk ^ delDtk RECENT ………………]

the table, EventContext is shown in Table 4.10.

Table 4.10 EventContext table after the composite event, addDel

EventName	Context
DelStk	Recent
AddStk	Recent

4.1.6 "Version"

This table is used to store the occurrence number of a primitive event. Initially, this version number is zero. When a primitive event occurs, the version number of the primitive event is increased by 1. The version number is also stored into the SysPrimitive Event table. The structure of the table Version is shown in Table 4.11.

Table 4.11 Version

VNO
1

The Version number is given in the following way:

update SysPrimitiveEvent set VNO = VNO + 1 where EVENTNAME = 'addStk';

delete from Version;

insert into Version select VNO from SysPrimitiveEvent where EventName = 'addStk';

Event context and Version tables are joined into SysContext table for use with parameter context. The join is executed by the trigger action on basic database operations such as insert and delete in the Persistent Manager.

4.1.7 “Stock”

This table is used as a test table on which operations (primitive events) such as insert, delete, and update are executed. The structure of the table is shown in Table 4.12.

Table 4.12 Stock

Symbol	co_name	price	timestamp

When one event (one operation) occurs on this table, One tuple is inserted into this table. An operation might look like the following:

insert into Stock values(‘KIM’,’IBM’,2132,sysdate); (in Oracle, sysdate command stores current time stamp into system table).

Table 4.13 shows the Stock table after one operation, insert.

Table 4.13 Stock table after one operation (insert)

Symbol	co_name	price	timestamp
KIM	IBM	2132	26-jun-00

4.2 The Language Filter and ECA Parser

All client commands go through the Language Filter. The ECA Commands are separated and sent to the ECA Parser. On the other hand, other SQL commands are sent to the SQL Server via the JDBC. It is the responsibility of the Language Filter to distinguish the types of ECA commands such as primitive event command, composite event command, repeated event command, etc. The Language filter filters each event command and then sends it to the corresponding functional modules in the ECA Parser.

In our implementation, functional modules are named CreatePrimitive( ), Createcomposite( ) and RepeatPrimitive( ) in the class ECAparser.

In the ECA Parser, the ECA commands are filtered into the ECA Parser from the Language Filter. The ECA Parser tokenizes and parses the commands. The syntax should be checked first. If there are no syntax errors, the ECA Parser will create corresponding events and rules which depend on the Java Local Event Detector (JavaLED). Figure 4.1 illustrates the Language Filter and the functional modules of the ECA Parser.

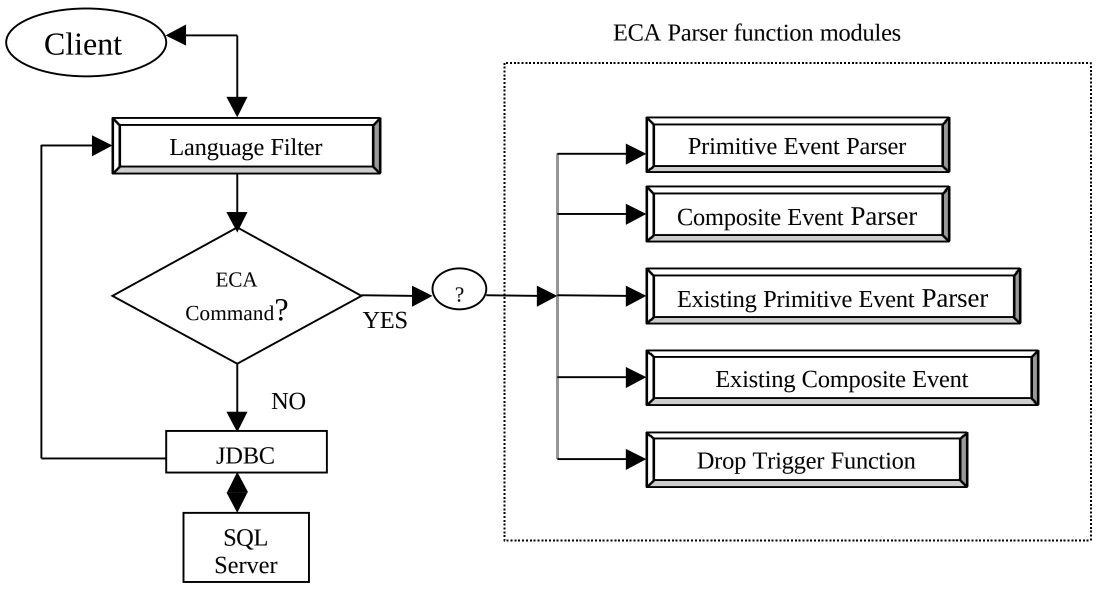


Figure 4.1 Language and Functional Modules in ECA Parser

There are five functional modules:

1) Primitive Event Parser: This module parses a primitive event.
2) Composite Event Parser: This module parses a composite event.

3) Repeated Primitive Event Parser: This module parses the repeated primitive event. A repeated primitive event implies that a trigger has already been created on the existing primitive event.
4) Repeated Composite Event Parser: This module parses the repeated composite event. If a trigger has already been created on the existing composite event, this functional module parses the event.
5) Drop trigger: If a client requests a SQL command such as “drop trigger triggerName”, this module parses the SQL command.

4.3 Persistent Manager

When we run an application program, we often need to store data into memory because we want to use them in the future. However, when the application program terminates, the memory used for that application is recovered by the operating system. Non-DBMS environments use the file system to store data across application runs. In a Relational DBMS, we can use system tables to do this. The Persistent Manager does this as part of the ECA agent. In the ECA Agent System, the only data that we keep in relational DBMS are the ECA rules. The Persistent Manager is in charge of the management of the ECA rules. The list below shows the functions of the Persistent Manager:

- Maintain system triggers: These triggers define the actions which should be executed when either primitive events or composite events occurs.
- Maintain system tables: All related system tables are managed by the Persistent Manager, based on system triggers.

- Persist ECA rules: The information of all the ECA rules is stored into the Relational DBMS.
- Restoration of the ECA rules: When system restarts, all ECA rules can be restored.

Figure 4.2 illustrates the Architecture of Persistent Manager.

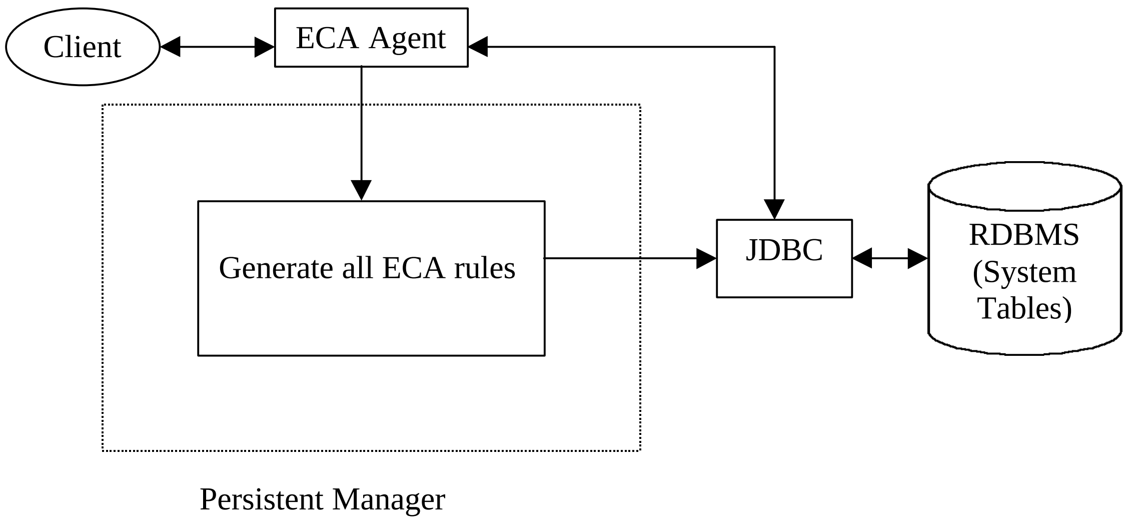


Figure 4.2 The Architecture of Persistent Manager

When a user connects the ECA Agent system, a thread serving one client is created. The persistent Manager runs inside the thread. It generates ECA rules for the triggers stored in the system tables of the RDBMS.

The generated ECA rules for triggers is sent to the RDBMS through JDBC and is used as trigger action parts when a related event occurs .

4.4 Snoop Preprocessor

Snoop, the Event Definition Language has been developed to provide users with an easier syntax for event type definition. There are several event operators such as AND, OR, SEQ, NOT, etc We need to preprocess Snoop expressions to java APIs.

SPP has been developed using the Java Compiler Compiler (JavaCC) by the Sentinel research group at the University of Florida. It converts Snoop, the Event Definition Language, into Java APIs so that users can include Snoop expressions in their APIs and compile them. Snoop expressions are the following:

1) Primitive event expression: *event begin (sellStockBegin : Vktest (class name) : IBM (instance name)) void sellStock(float price);*

2) Composite event expression: *event addDel = delStk ^ addStk ;*

3) Rule expression: *rule r1[e_plus, condition, action, 2, IMMEDIATE, RECENT];*

The following Java APIs are created by the Snoop preprocessor for both a primitive event and a composite event.

1) The API of primitive event: *EventHandle sellStockBegin = myAgent.createPrimitiveEvent("sellBegin", "VkTest",EventModifier.BEGIN, "void sellStock(int)", DetectionMode.SYNCHRONOUS);*

2) The API of composite event: *EventHandle seqEvent = myAgent.createCompositeEvent (EventType.SEQ,"e8", sellSell , buyandGet);*

3) The API of rule: *myAgent.createRule("R1", seqEvent,"VkTest.True", "VkTest.displayPrice", 1 , CouplingMode.DEFAULT, Context.RECENT);*

In ECA Parser, “username+eventname.java” (for example, ykimaddStk.java or ykimdelStk.java) files are generated to create primitive events and to get event handlers in the Java LED. Primitive events are created in the method, CreatePrimitive( ). Those Java files include one of the Java LED methods, createPrimitiveEvent( ), to get the EventHandler for the primitive event. “username+eventname.java” (in this case, ykimaddDel.java) is also used to detect a composite event using the method, Createcomposite( ). Snoop expressions in a composite event definition are parsed into input_composite.txt and converted into Java APIs by Snoop Preprocessor(See the file, Snoop.java). Those Java APIs are stored into “compositeevent.txt” and includes the definition of a composite event and the definition of a rule. The contents of the file, compositeevent.txt are inserted into “ykimaddDel.java”. Whenever each Java file (in this case, ykimaddStk.java, ykimdelStk.java, ykimaddDel.java) is generated, it need to be compiled to get its class file. We use Java Dynamic method call to invoke the method, call_addStk( ) in ykimaddStk.java, the method, call_delStk( ) in ykimdelStk.java and the method, call_addDel( ) in ykimaddDel.java. The Java LED methods, createPrimtiveEvent( ) and createCompositeEvent( ) are executed to get event handlers inside call_eventname methods. Whenever primitive events such as insert and delete occur, those primitive events are raised (registered into event tree) to the Java LED by using “raiseBeginEvent( )” in the file, Led.java. Figure 4.3 shows the generated files which are described in the section.

```
 import Sentinel.*;
 import java.util.Vector;                                    //ykimaddStk.java
 import java.util.Hashtable;       import java.util.Enumeration;
 public class ykimaddStk{      public static EventHandle addStk =null;   public static void call_addStk() {  ECAAgent myAgent
= ECAAgent.initializeECAAgent();
   addStk = myAgent.createPrimitiveEvent("addStk","Led", EventModifier.BEGIN, "void addStk()",
DetectionMode.SYNCHRONOUS);    } }
```

```
import Sentinel.*;
 import java.util.Vector;                                    //ykimdelStk.java
 import java.util.Hashtable;
 import java.util.Enumeration;        public class ykimdelStk{
 public static EventHandle delStk =null;
  public static void call_delStk(){
 ECAAgent myAgent = ECAAgent.initializeECAAgent();
 delStk = myAgent.createPrimitiveEvent("delStk","Led", EventModifier.BEGIN, "void delStk()",
DetectionMode.SYNCHRONOUS); } }
```

```
 import Sentinel.*;
 import java.util.Vector;                                    //ykimaddDel.java
 import java.util.Hashtable;
 import java.util.Enumeration;           public class ykimaddDel{
 static String rdbms = "";      static String url = "";       static String username = "";      static String password = "";
 public static EventHandle ykimaddDel =null;
 public static void call_addDel( String Prdbms, String Purl, String Pusername, String Ppassword) {
 rdbms = Prdbms;
 url = Purl; username = Pusername; password = Ppassword;
 ECAAgent myAgent = ECAAgent.initializeECAAgent();
 ykimaddDel =
myAgent.createCompositeEvent(EventType.AND ,"event_ykimaddDel" ,(EventHandle)ykimdelStk.delStk ,(EventHandle)ykim
addStk.addStk);
 myAgent.createRule("Rule_addDel",ykimaddDel,
"ykimaddDel.True" ,"ykimaddDel.addDelykim" ,1 ,CouplingMode.IMMEDIATE ,Context.RECENT);
}
 public static boolean True(ListOfParameterLists parameterLists)
 System.out.println("***** From Condition ***** ");
 return true;}
 public static void addDelykim(ListOfParameterLists paramLists)
    String paraspc = "delete from parameterContext";
    Jdbc storedPara = new Jdbc(rdbms,url,username,password, paraspc);
    storedPara.ExecuteSqlUpdate("delete from parameterContext ");
    String spc0 = "delete from stock_deleted_tmp";
    Jdbc storedProCom0 = new Jdbc(rdbms,url,username,password, spc0);
    storedProCom0.ExecuteSqlUpdate("delete from stock_deleted_tmp");
    spc0 = "insert into stock_deleted_tmp select * from stock_deleted, sysContext where  sysContext.context='RECENT' and
sysContext.EVENTNAME='delStk' and stock_deleted.vNo=sysContext.vNo";
    storedProCom0 = new Jdbc(rdbms,url,username,password, spc0);
    storedProCom0.ExecuteSqlUpdate("insert into stock_deleted_tmp");
 String Para_spc0= "insert into parameterContext  select * from sysContext  where  vno in ( select max(vno) from sysContext
where eventname='delStk') and eventname='delStk'";
 Jdbc storedPara0= new Jdbc(rdbms,url,username,password, Para_spc0);
 storedPara0.ExecuteSqlUpdate("insert into parameterContext ");
    String spc1 = "delete from stock_inserted_tmp";
   Jdbc storedProCom1 = new Jdbc(rdbms,url,username,password, spc1);
   storedProCom1.ExecuteSqlUpdate("delete from stock_inserted_tmp");
    spc1 = "insert into stock_inserted_tmp select * from stock_inserted, sysContext where sysContext.context='RECENT' and
sysContext.EVENTNAME='addStk' and stock_inserted.vNo=sysContext.vNo";
    storedProCom1 = new Jdbc(rdbms,url,username,password, spc1);
    storedProCom1.ExecuteSqlUpdate("insert into stock_inserted_tmp");
 String Para_spc1= "insert into parameterContext  select * from sysContext  where  vno in ( select max(vno) from sysContext
where eventname='addStk') and eventname='addStk'";
 Jdbc storedPara1= new Jdbc(rdbms,url,username,password, Para_spc1);
 storedPara1.ExecuteSqlUpdate("insert into parameterContext ");}}
```

Figure 4.3 -- continued

```
import Sentinel.*;
import java.util.Vector;                              //Led.java
import java.util.Hashtable;
import java.util.Enumeration;
public class Led {

    //PrimitiveEventMethod
    public void PrimEvent(String eventname,String tablename, int vno) {

     EventHandle[] eventHandler = ECAAgent.getEventHandles(eventname);
     ECAAgent.insert(eventHandler,"eventname",eventname);
     ECAAgent.insert(eventHandler,"tablename",tablename);
     ECAAgent.insert(eventHandler,"vno",vno);
     ECAAgent.raiseBeginEvent(eventHandler, this);
   System.out.println("***** raiseBeginEvent ***** " + eventname);
   }
    //ECA_Condition
   public static boolean True(ListOfParameterLists parameterLists) {
         System.out.println("***** From Condition ***** ");
         return true;
   }
}
```

Figure 4.3 The Generated Java Files

4.5 Action Handling by the Java LED

The Actions executed by the Java LED can be the execution of SQL commands in the Oracle Server, the update of system tables, and the invocation of stored procedures by action method executed when a composite event is detected. Every ECA rule has its event definition as well as rule (only action in case of a trigger) definition. Every event action is processed according to its rule definition. We will discuss the Java LED actions in the section 6.2 and Figure 6.6.

CHAPTER 5
IMPLEMENTATION OF PRIMITIVE EVENTS

This chapter discusses the implementation details of primitive event detection. Although relational DBMS (RDBMS) provides a trigger mechanism, it is very limited in defining events, conditions, and actions. The need of a full-fledged active capability is required by many applications. The ECA Agent System uses the trigger mechanism of RDBMS to support active capability and extends it significantly to turn the RDBMS into a fully active DBMS.

5.1 Trigger Mechanism for Events by the ECA Agent System

Triggers supported by the RDBMS have a number of limitations:

- A trigger cannot be applied to more than one table. (Sybase and DB2 5.0 has this limitation, but Oracle and DB2 6.0 do not have this limitation any more).
- Composite event is not allowed.
- New event on a table for the same operation(insert, delete or update) overwrites the previous one without warning message.
- Only atomic values (not tables) can be passed as parameters to stored procedures.

The ECA Agent overcomes those limitations by providing a complete active database capability. The functionality extended by the ECA Agent is the following:

- We can specify any number of triggers on the same event. Suppose that we have already defined one primitive event as follows:

Create trigger t_addStk on Stock after insert event addStk………

Once one trigger is defined, we can define another trigger on the same event as follows:

Create trigger t_addStk1 on Stock after insert event addStk………

We can add different trigger actions on the same event to the original trigger actions.

- Composite events: A composite event is constructed by using primitive events, event operators and previously defined composite events.
- Drop trigger: Triggers on either primitive events or composite events can be dropped (we discuss the details of dropping triggers in the Section 5.5)

5.2 The Syntax of Primitive Events

In this section, we describe how we define a primitive event using SQL statement. In defining a primitive event, the only difference between Oracle SQL statement and this ECA Agent statement for primitive event is the event declaration using the keyword, *event*. That is, we extend the trigger definition to provide ECA active capability. Figure 5.1 shows Oracle SQL statement. Figure 5.2 shows syntax of primitive event definition of the ECA Agent using Oracle SQL grammar.

```
create trigger t_addStk after insert on stock
for each row
begin
dbms_output.put_line('trigger t_addStk occurs on primitive event, addStk');
end;
```

Figure 5.1 Oracle SQL Statement

```
create trigger t_addStk after insert on stock event addStk
for each row
begin
dbms_output.put_line('trigger t_addStk occurs on primitive event, addStk');
end;
```

Figure 5.2 Primitive Event Definition using Oracle SQL Statement

A primitive event is defined on the table Stock, for the operation insert, in the above example. The trigger definition for the primitive event is written by SQL statement between *begin* and *end* and executed in the SQL server.

5.3 Processing of a Primitive Event

The processing of a primitive event involves parsing and generation of the primitive event node in the event graph. Figure 5.3 shows the processing steps of the primitive event. The processing of the primitive event includes the following steps:

1) Syntax check: The command is checked. If a syntax error is found, an error message is returned to the client.
2) Duplication check on trigger: The duplication of trigger name is not allowed in the RDBMS. Trigger name of an event is checked. If trigger name is duplicated, an error message is returned to client.
3) Persistent code generation: SQL code is generated and persisted. The code includes the following:
    - Insert into SysEcaTrigger values (‘ykim’,’t_addStk’,’t_addStk_proc’, sysdate, ’addStk’);

- Insert into SysPrimitiveEvent values ('ykim' , 'addStk', 'Stock', 'insert', 'after', sysdate, 0);

4) Java LED code generation: If there is no error in defining a primitive event, The ECA code is generated to create a primitive event node in the Java LED. We create the primitive event, *addStk* using the Java LED as follows:
   ECAAgent myAgent = ECAAgent.initializeECAAgent( );
   EventHandle addStk = myAgent.createPrimitiveEvent("addStk", "Led", EventModifier.BEGIN, "void addStk( )", DetectionMode.SYNCHRONOUS);
   (This contents will be inserted into the file, *ykimaddStk.java* in our implementation).
5) File generation: We create the new file to get an event handler for the primitive event. (For example, in our implementation, *ykimaddStk.java* is created. This file is compiled automatically and implicitly using Java Runtime class). We use dynamic method call using Java Reflection (In our implementation, the called method is *call_addStk* inside the file, *ykimaddStk.java*).
6) Create related tables: The tables, Stock_inserted and Stock_deleted are created. The Stock_inserted table is used to store the inserted tuples of a table. It is very similar to table Stock except that it has VNO column. VNO is used to store the occurrence number of the insert operation. For example, the event, addStk is detected whenever a tuple is inserted into Stock table. VNO is increased by 1 for each insertion. (The value of VNO is used to collect

parameter for composite event). Table 5.1 shows Stock_inserted and Stock_deleted tables.

Table 5.1 Stock_inserted and Stock_deleted

symbol	co_name	Price	timestamp	Vno
ykim	IBM	2132	26-jun-00	1

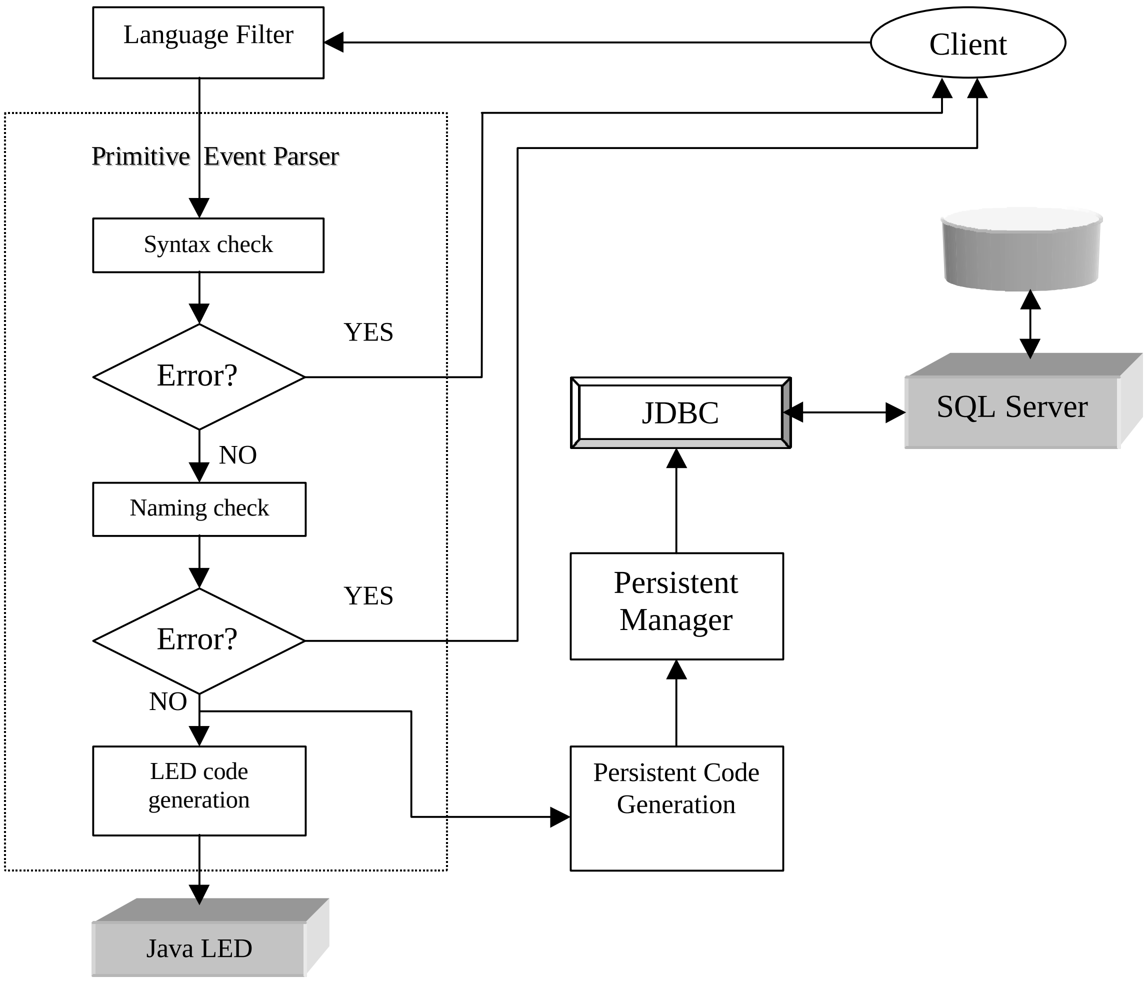


Figure 5.3 Parsing and Generating of Primitive Event

7) Trigger creation for primitive event: After an operation (insert or delete) is executed on ‘Stock’, the trigger will be fired for the operation. The trigger for the operation includes the following steps:

- Set and get event occurrence number:

  Update SysPrimitiveEvent set VNO=VNO+1 where eventName = ’addStk’;

  Delete from Version;

  Insert into Version select VNO from SysPrimitiveEvent where eventName = ‘addStk’;

- Insert tuples into table, SysContext (to be used for composite event).

  Delete from SysContext where eventName=’addStk’;

  *Insert into SysContext select * from eventContext, version where eventContext.eventName =’addStk’;*

  Table, eventContext is made when we define a composite event.

- Insert an inserted tuple into table, Stock_inserted(in the table, Stock_deleted, *new* is replaced with *old*).

  Declare cursor c1 is select VNO from version;

  begin

  for VNO_rec in c1 loop

  Insert into stock_inserted values (:new.symbol ,:new.co_name, :new.price, :new.time, VNO_rec.VNO);

  end loop;

  end;

- Primitive Event Notification:

  insert into notify select EVENTNAME, TABLENAME, VERSION.VNO from SYSPRIMITIVEEVENT, VERSION where EVENTNAME = 'addStk';

  When a primitive event occurs (in this case, insert operation), event name and table name and version number are inserted into notify table. Then, all the contents of notify table on the primitive event are inserted to the Java LED using primEvent method of Led.java when a primitive event occurs. The raiseBeginEvent method inside primEvent raise the primitive event (refer to LED.java file). For example, when insert operation (primitive event: "insert into stock values(…….)") occurs, the SQL statement goes to the Language Filter as follows:

  //in serveOneClient.java

```
String qs1 = "delete from notify";
Jdbc createTableSql = new Jdbc(rdbms,url, username,password, qs1);
createTableSql.ExecuteSqlUpdate("initialize table notify ");
// before client send SQL statement  to Language Filter, ECA Agent delete all contents of notify table. ………………………….
LanguageFilter lf=new LanguageFilter();
result=lf.filter(sqlhost,rdbms,database,url,username,password,sqlstatement); //at this point, ECA Agent knows client sent the server his request. Before the result is returned back from Language Filter, The server will make the contents of notify table if the SQL statement is primitive event operation. (notify table content is changed only by the trigger action on a primtive event (insert or delete)). Here, at this point,  we check if notify table contains a tuple on the event(this one must be guaranteed because notify table content is
```

created only when insert or delete opreation occurs and its trigger is activated). Notity table check is shown in Figure 5.4.

```
 String qs2 = "select eventname from notify";
 Jdbc selectSql = new Jdbc(rdbms,url,username,password,qs2);
 qs2 = selectSql.GetFromNotify().trim ();
 String en = selectSql.GetFromNotify().trim ();

 if (!en.equals("F") && !en.equals("Empty"))    // if trigger has been fired
   {  qs2 = "select TABLENAME from notify";
      selectSql = new Jdbc(rdbms,url,username,password,qs2);
      String tn = selectSql.GetFromNotify().trim ();

      qs2 = "select VNO from notify";
      selectSql = new Jdbc(rdbms,url,username,password,qs2);
      String vnos = selectSql.GetFromNotify().trim ();
      Integer vnoi = new Integer(vnos);
      int vno = vnoi.intValue();

      if(test1 == null) test1 = new Led();  //  new Led class instance because we
declare
                                         // Led test1 = null;     above
           test1.PrimEvent(en,tn,vno);    //to raise primitive event. This one raise
an event in
                                           //  the Java LED
   }
```

Figure 5.4 Notification Check

// Now, the Event can be notified to the Java LED.

……………….

out.println(result); //the result returns back to client.

5.4 Syntax for an Existing Primitive Event

Once an event is defined, the user can define additional trigger on the event. Figure 5.5 illustrates the syntax of defining trigger an a previously defined event.

```
Create trigger t1_addStk event addStk
for each row
begin
dbms_output.put_line('trigger t1_addStk occurs on existing event, addStk');
end;
```

Figure 5.5 Syntax of Defining a Trigger on Existing Event

In this example, we do not have to declare *after [operation]* and *on [table-name]*. In our implementation, if user creates another trigger on an existing event, Repeated Primitive Event Parser (the method, RepeatPrimitive(……..) in ECAparser class) is called. Then, syntax is changed to original Oracle SQL syntax shown in Figure 5.6. This functionality is used to add an additional trigger action to an existing event.

```
Create trigger t1_addStk  after insert on Stock
for each row
begin
dbms_output.put_line('trigger t1_addStk occurs on existing event, addStk');
end;
```

Figure 5.6 The Changed Event Definition on Existing Event

After this trigger definition, only SysEcaTrigger table is updated adding the tuple (as shown in shadow in Table 5.2):

Table 5.2 SysEcaTrigger after the existing event (addStk) definition

Dbname	Username	Triggername	Triggerproc	timestamp	eventname
ORCL	Ykim	t_addStk	t_addStk_proc	26-jun-00	addStk
ORCL	Ykim	T1_addStk	t1_addStk_proc	26-jun-00	addStk

5.5 Syntax of Dropping a Trigger

The Syntax of dropping a trigger is the following in the Oracle DBMS:

Drop trigger trigger_name;

Our implementation also uses the same syntax for user transparency. This drop command is requested by user, our implementation checks if this trigger is defined on a primitive event or a composite event. If it is defined on a primitive event, it invokes the “Drop Trigger on Primitive” method. If the trigger is defined on a composite event, “Drop Trigger on Composite” method is invoked. If the trigger did not be defined on a primitive event or a composite event, the trigger is considered as original RDBMS trigger.

When the client sends a drop trigger request to the ECA Agent, the ECA Agent takes the following steps:

1) Delete the tuple related to the trigger from the table, SysEcaTrigger.
2) Drop the trigger in the Oracle RDBMS.
3) Before a primitive event is removed from SysPrimitiveEvent table, we first need to check if another trigger has been created for the event. If there are no other triggers defined on the primitive event, remove that event tuple from the table, SysPrimitiveEvent. If there is a trigger defined on the primitive event, the event cannot be removed from SysPrimitiveEvent.

We do not need to remove the primitive event from the Java LED because we can use the primitive events in the next use for another composite event (remember that predefined primitive events can be used to define new composite event in the Java LED as shown in Figure 5.7). The next time the user logs in, the event tree created is only for the events that exist in the table. At that time, the node will not be created.

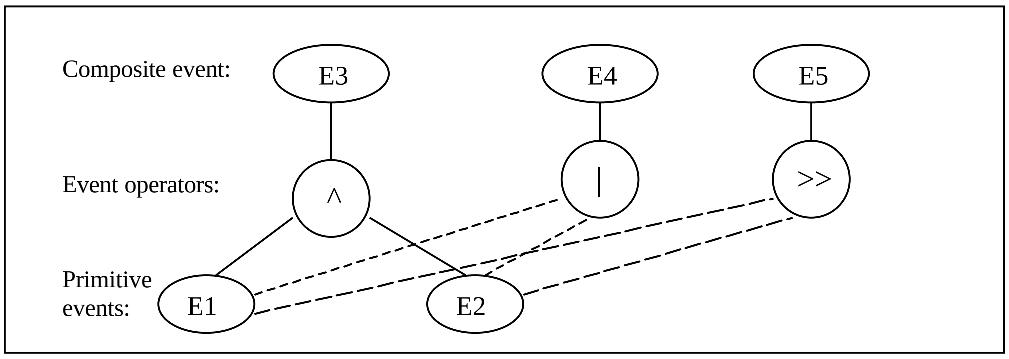


Figure 5.7 Event Tree in the Java LED

CHAPTER 6
IMPLEMENTATION OF COMPOSITE EVENTS

In this prototype, not only primitive events are supported but also composite events are supported while Relational database systems (RDBMSs) do not support composite events. In this chapter, we discuss the implementation of composite events. It includes the syntax of composite events, the parsing and generation of composite events, event notification, ECA action and parameter context.

6.1 The Syntax of Composite Events

To provide a composite event capability, we need to extend a trigger definition. Figure 6.1 shows the syntax of a composite event definition.

```
create trigger  trigger_name
event event_name [= Snoop_expression] [coupling_ mode] [parameter_context] [priority]
begin SQL_statements end;
coupling_mode := RECENT | CHRONICLE | CONTINUOUS
                      | CUMULATIVE
parameter_context := IMMEDIATE | DEFERED | DETACHED
priority := positive integer
        Snoop_expression ::= E1
                        E1 ::= E1 OR E2 | E2
                        E2 ::= E2 AND E3 | E3
                        E3 ::= E3 SEQ E4 | E4
                        E4 ::= NOT (E1, E1, E1) | A (E1, E1, E1) | A* (E1, E1, E1)
                               | P (E1, [time string], E1) | P* (E1, [time string], E1)
                               | [time string] | E1 PLUS [time string] | (E1) | event_name
```

Figure 6.1 Syntax of Composite Events

We use the keyword, *event* to indicate composite events. The Coupling mode is used to define event execution time (The default mode is IMMEDIATE). Parameter Context is also used to collect event instances in the Java LED (The default context is RECENT). We use Snoop, event definition Language to declare a type of composite event in SQL trigger statement. The BNF of Snoop is also shown in Figure 6.1. Figure 6.2 shows the composite event definition.

```
create trigger t_addDel  event  addDel = addStk ^ delStk IMMEDIATE  RECENT 1
for each row
begin
dbms_output.put_line('trigger t_addDel occurs on composite event, addDel');
end;
```

Figure 6.2 Composite Event Definition

6.2 The Processing of Composite Events

When clients define SQL trigger command as composite event, the command from clients goes through the Language Filter and then it is sent to the Composite Event Parser. The Composite Event Parser will parse it and generate appropriate code for activating the Java LED and persistent manager. Figure 6.3 shows the workflow of the Composite Event Parser.

There are several steps to process composite events:

1) Syntax check : The command is checked. If a syntax error is found, an error message is returned to clients.

2) Duplication check on trigger name: The duplication of trigger name is not allowed in the RDBMS. Trigger name of an event is checked. If trigger name is duplicated, an error massage is returned to clients.

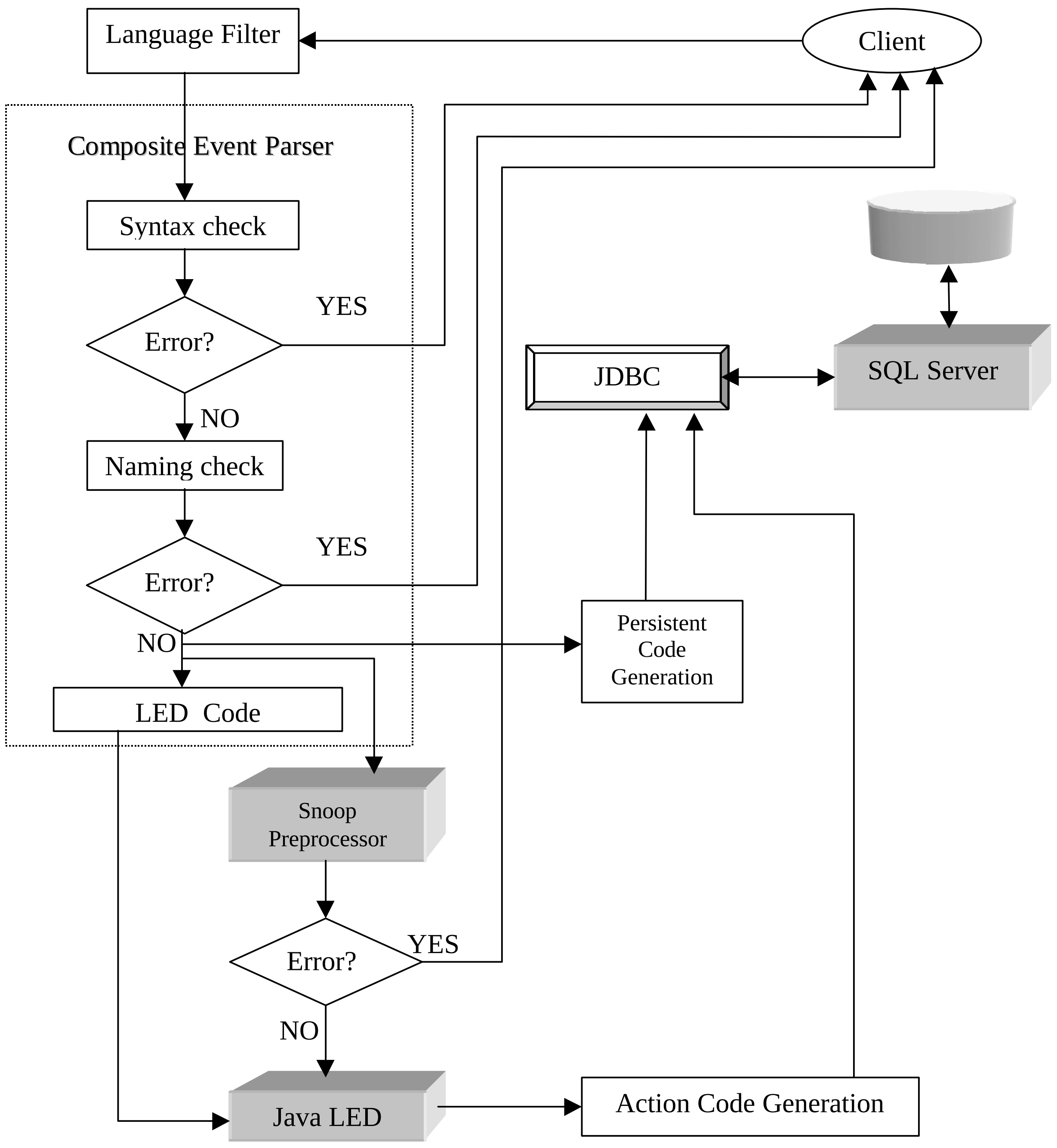


Figure 6.3 The Parsing and Generating of Composite Events

Otherwise, event definition string (for example, event addDel = addStk ^ delStk) is sent to Snoop preprocessor. The code generation for Java LED action is done by the ECA parser. Figure 6.4 shows the code for Java LED.

```
import Sentinel.*;                          //ykimaddDel.java
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;
public class ykimaddDel{
static String rdbms = "";
static String url = "";
static String username = "";
static String password = "";
public static EventHandle addDel =null;

public static void call_addDel( String Prdbms, String Purl, String Pusername,
String Ppassword) {
        rdbms = Prdbms;
        url = Purl;
        username = Pusername;
        password = Ppassword;
        ECAAgent myAgent = ECAAgent.initializeECAAgent();
         ………………
}
```

Figure 6.4 Java LED Code Generation

We can see the whole picture of ykimaddDel.java in Figure 4.3. The ECA parser generates this file, compiles this file and calls the method call_addDel( ) using Java Dynamic Method inside the method Createcomposite( ). The parameters of calll_addDel are abtained using Java Reflection Call inside the method Createcomposite( ) of ECAParser.java.

3) Snoop Preprocessor: The Snoop preprocessor parses the event definition string. If there is no error when the string is checked, the Snoop Preprocessor creates the composite event in the Java LED. Event definition and Rule definition are generated by the Snoop Preprocessor. Figure 6.5 shows the composite event definition and rule definition after Snoop preprocessing. “ykimdelStk.delStk” and “ykimaddStk” are the event handlers obtained by compiling ykimdelStk.java and ykimaddStk.java generated in the ECA Parser. We create a class with one variable(event handler) for each event. We see these event handlers in Figure 4.3. This code is inserted into the Java LED code of Figure 6.4. The Snoop Preprocessor creates two files. One is *eventlist.txt* (this file includes the events need to define the composite event, is generated by getEventlist method in Snoop.java and is used to make eventContext table). The other is *compositeevent.txt* (this file includes the event definition and rule definition converted from event definition string).

```
addDel = myAgent.createCompositeEvent(EventType.AND,
"addDel" ,(EventHandle)ykimdelStk.delStk,
(EventHandle)ykimaddStk.addStk);
 myAgent.createRule("Rule addDel",addDel,"Led.True",
"ykimaddDel.ykim" , 1,CouplingMode.DEFAULT,Context.RECENT);
```

Figure 6.5 Event and Rule Definitions for the Java LED in compositeevent.txt

4) ECA Action Generation: ECA Action for the composite event is executed by calling the method “addDelykim(……..)”. The action method, addDelykim(……..)

includes the following SQL actions shown in Figure 6.6. The method addDelykim( ) is generated in GetEcaAction( ) of AnalysisCompositeEvent.java when a user defines a composite event. GetEcaAction( ) is called in Createcomposite( ) of ECAParser.java. The action method addDelykim( ) is executed when the composite event, addDel is detected by the Java LED.

```
delete from stock_deleted_tmp;

insert into stock_deleted_tmp select * from stock_deleted, sysContext
where sysContext.context='RECENT'
and sysContext.TABLENAME='stock'
and stock_deleted.vNo=sysContext.vNo;

delete from stock_inserted_tmp;

insert into stock_inserted_tmp select * from stock_inserted, sysContext
where sysContext.context='RECENT'
and sysContext.TABLENAME='stock'
and stock_inserted.vNo=sysContext.vNo;
```

Figure 6.6 ECA Actions in Java LED

5) Persistent Code Generation: The following SQL codes are persisted:

   Insert into SysEcaTrigger values (‘ORCL’, ’ykim’, ’t_addDel’, ’t_addDel_proc’, sysdate, ’addDel’);

   Insert into SysCompositeEvent(‘ORCL’,’ykim’,’addDel’,’addStk ^ delStk’, sysdate, IMMEDIATE, RECENT, 1);

6.3 Event Notification and Composite Event Detection

When primitive events occur, the SQL server executes a trigger action. This trigger action will be in the SQL server. The Java LED should recognize those primitive events to detect a composite event. In our implementation, we have created the table, *notify* to put the information of the primitive events when a primitive occurs. Table 6.1 shows the structure of notify table.

Table 6.1 *notify* table

Eventname	Tablename	Vno

After the primitive event (addStk) occurs, this table is changed as the following Table 6.2:

Table 6.2 notify table after a primitive event

Eventname	Tablename	Vno
addStk	stock	1

This information on the primitive event in SQL server needs to be notified to ECA Agent to detect a composite event. The Java LED is used to detect composite events. The Java file, Led.java shown in Figure 6.7 is used to detect composite events:

```
import Sentinel.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;
public class Led {
public void PrimEvent(String eventname,String tablename, int vno) {

    EventHandle[] eventname = ECAAgent.getEventHandles(eventname);

    ECAAgent.insert(eventname,"eventname",eventname);
    ECAAgent.insert(eventname,"tablename",tablename);
    ECAAgent.insert(eventname,"vno",vno);

    ECAAgent.raiseBeginEvent(eventname,this);
  System.out.println("raiseBeginEvent :" + eventname);
  }
}
```

Figure 6.7 Led.java File for Composite Event Detection

The method, PrimEvent(………) is used to raise the specified primitive event by event name. Event name, table name and version number are notified from SQL Server. When a primitive event occurs, Led.java will get the event notification from SQL Server and call the method, *PrimEvent* to raise the primitive event as shown in Figure 6.8.

```
If (test1=null) test1 = new Led( );  // Led class is initiated only one time.
test1.PrimEvent(eventname, tablename,vno);   //raise primitive event.
```

Figure 6.8 Method Call for Raising Primitive Events

Suppose that the composite event (event addDel = addStk ^ delStk) is need to be detected. When both addStk and delStk primitive events are notifed and raised, the composite event addDel is detected. Figure 6.11 shows how the primitive events are notified from

SQL server to the Java LED and how the composite event is detected by the Java LED. When a primitive event is defined using trigger definition, the ECA Agent generate the Java files such as ykimaddStk.java and ykimdelStk.java to get the event handlers for the primitive events and compiles those Java files The Java LED method, createPrimitiveEvent( ) is called using Java Dynamic Method Call. When a composite event is defined, the Java file such as ykimaddDel.java is generated by ECA Agent. The trigger on the primtive events puts the event information such as event name, table name and version number into notify table. We use the contents of notify table to raise primitive event in the Java LED. LED.java file is used to call raiseBeginEvent method in the Java LED.

In the Java LED, composite event can be detected if we use raiseBeginEvent (EventHandler, ..) of the primitive events after we get the event handlers of all the events by createPrimitiveEvent( ) and createCompositeEvent( ). We use the same way to notify primitive events to the Java LED and detect a composite event from the Java LED. The codes below shows how to pass the notify table informaiton as parameters for nofifying primitive events to the Java LED in serveOneclient.java. Notice that the notify table has no contents before insert and delete operation.

```
//check if there is primitive event raised.
 String qs2 = "select eventname from notify";
 Jdbc selectSql = new Jdbc(rdbms,url,username,password,qs2);
 String en = selectSql.GetFromNotify().trim ();
 if (!en.equals("F") && !en.equals("Empty"))      //   if a primitive event have
 //occurred.
```

```
  {  qs2 = "select TABLENAME from notify";
       selectSql = new Jdbc(rdbms,url,username,password,qs2);
       String tn = selectSql.GetFromNotify().trim ();
       qs2 = "select VNO from notify";
       selectSql = new Jdbc(rdbms,url,username,password,qs2);
       String vnos = selectSql.GetFromNotify().trim ();
       Integer vnoi = new Integer(vnos);
       int vno = vnoi.intValue();
    if(test1 == null) test1 = new Led();
       test1.PrimEvent(en,tn,vno);   //raise primitive event inside PrimEvent( )
}
```

6.4 ECA Action

The rule action part of a composite event is associated with primitive events. In our implementation, we create the file, *ykimaddDel* to store rule action of composite event. It is compiled implicitly and automatically by the Java Runtime class in the ECA Parser. Using Java Dynamic Method Call by the Java Reflection, we can call the method, *call_addDel( ),* and activate the rule action (call_addDel( ) includes the event definition and the rule definition for the composite event. The rule definition calls the action method ykimaddDel( )). Figure 6.9 shows the Java Dynamic Call by Java Reflection. Figure 6.10 shows the contents of rule action method addDelykim( ) by SQL commands.

```
From ECAparser class:
Class[] paramTypes = new Class[4];
      paramTypes[0] = (new String()).getClass();
      paramTypes[1] = (new String()).getClass();
      paramTypes[2] = (new String()).getClass();
      paramTypes[3] = (new String()).getClass();
      Object[] params = new Object[4];
      params[0] = rdbms;
      params[1] = url;
      params[2] = username;
      params[3] = password;

ConstructClass.WriteToFile(className + ".java", contents);
ConstructClass.CompileFile(className );
CallDynamicMethod.ExecuteMethod(className, "call_addDel", paramTypes,
     params);

In CallDynamicMethod class:
Public …ExecuteMethod(String className, String methodName, Class[] paramTypes, Object[] params){
…………………
java.lang.reflect.Method meth = null;
meth = cla.getMethod(method_name, paramTypes);   // find the particular method
Object obj = null;
meth.invoke(obj, params);      // execute the particular method.
 }
```

Figure 6.9 The Java Dynamic Call by Java Reflection

6.5 Parameter Context.

A composite event has a parameter context to collect event instances to be participated in detection of composite event. We defines the basic operations of the RDBMS as primitive events. If we create trigger for insert operation using event name, addStk, the event, addStk can occur many times. We can persist this event several times considering the repeated event as the event on an existing event. Whenever the same event occurs repeatedly, we can know how many times the event have occurred by version number. When a primitive event is detected, the tables termed

Stock_inserted_tmp and *Stock_deleted_tmp* are created. The structure of those tables is shown in Table 6.3.

Table 6.3 *Stock_inserted_tmp* and *Stock_deleted_tmp*

Symbol	Co_name	Price	Time	Vno	EventName	Context	VNo1

When user defines a composite event, all primitive evens for the composite event and the contexts of the composite event are put into the table *EventContext*. Table 6.4 shows the contents of *EventContext* when a composite is defined.

Table 6.4 *EventContext*

EventName	Context
‘addStk’	‘RECENT’
‘delStk’	‘RECENT’

When a primitive event occurs, two tables, *Stock_inserted* and *EventContext* are joined. The result is inserted into the table *SysContext*. Table 6.5 shows *SysContext* when the primitive event, addStk occurs.

Table 6.5 *SysContext*

EventName	Context	Vno
‘addStk’	‘RECENT’	1

Table 6.6 shows the content of *SysContext* when the same primitive events occur several times.

Table 6.6 SysContext after the same events occurs

EventName	Context	Vno
‘addStk’	‘RECENT’	1
‘addStk’	‘RECENT’	2
‘addStk’	‘RECENT’	3
‘delStk’	‘RECENT’	1
‘delStk’	‘RECENT’	2

When a composite event occurs, two tables, *Stock_inserted* and *SysContext* are joined and the results are inserted into the table, *Stock_inserted_tmp*. Two tables, *Stock_deleted* and *SysContext* are also joined and the results are inserted into the table, *Stock_deleted_tmp*. If the parameter context is RECENT, the contents of *Stock_inserted_tmp* and *Stock_deleted_tmp* is shown in Table 6.7 and 6.8 respectively.

Table 6.7 *Stock_inserted_tmp* after join

Symbol	Co_name	Price	Time	VNo	EventName	Context	VNo1
ykim	IBM	3456	Jun-01-04	1	AddStk	RECENT	1
ykim	Sybase	3456	Jun-01-05	2	AddStk	RECENT	2
ykim	Oracle	3456	Jun-01-06	3	AddStk	RECENT	3

Table 6.8 *Stock_deleted_tmp* after join

Symbol	Co_name	Price	Time	VNo	EventName	Context	VNo1
ykim	IBM	3456	Jun-01-04	1	delStk	RECENT	1
ykim	Sybase	3456	Jun-01-05	2	delStk	RECENT	2

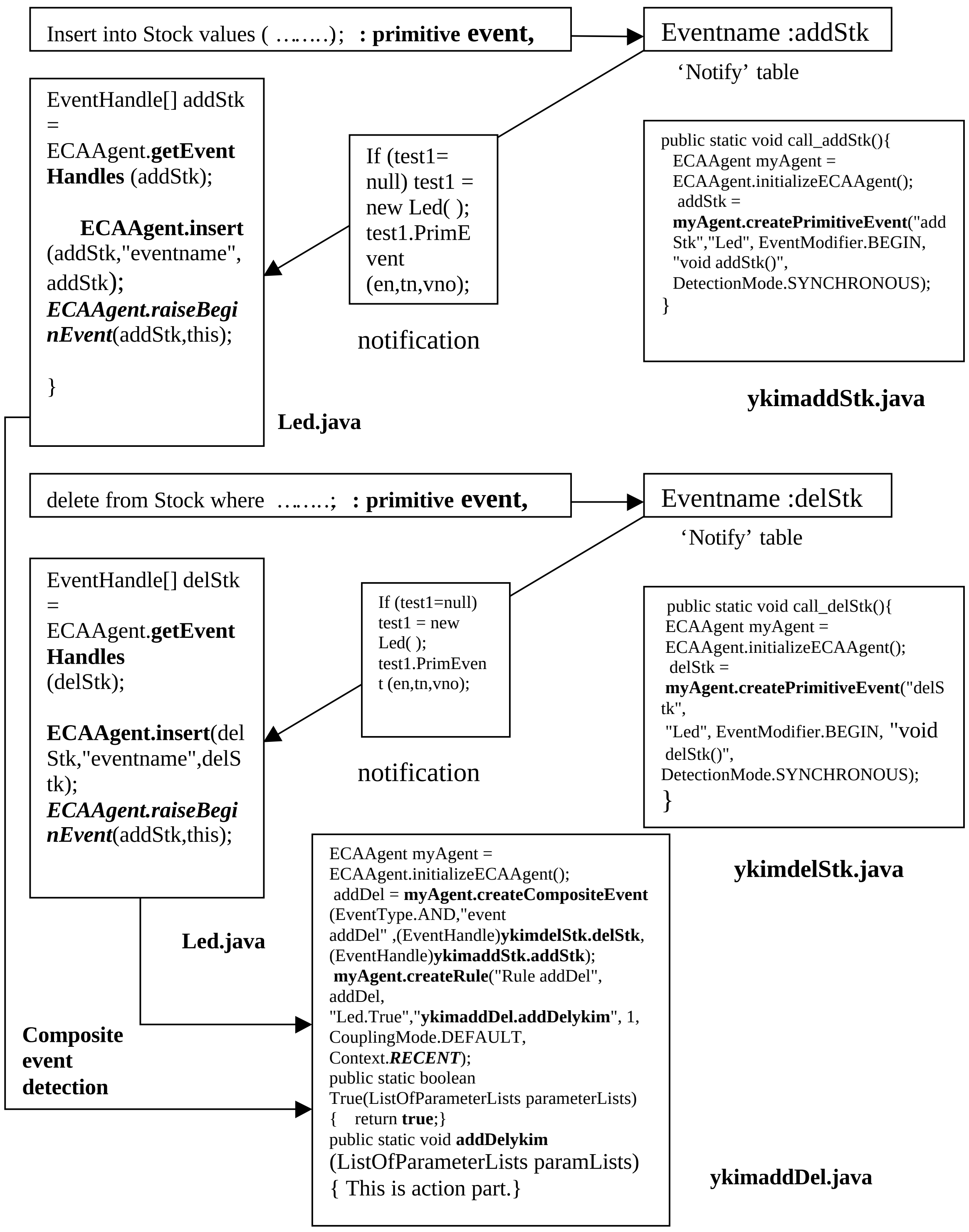
VNO1 represents the version number of SysContext table. For example, the version number of SysContext will be VNO1 of Stock_inserted_tmp table after two tables are joined.

```
event 'addStk':
Delete stock_inserted_tmp;
Insert into stock_inserted_tmp
Select * from stock_inserted, SysContext Where   SysContext.context='RECENT'   and
SysContext.eventname = 'addStk'   and
Stock_inserted.vNo = SysContext.vNo

event 'delStk':
Delete stock_deleted_tmp;
Insert into stock_deleted_tmp
Select * from stock_deleted, SysContext Where   SysContext.context='RECENT'   and
SysContext.eventname = 'delStk'   and
Stock_deleted.vNo = SysContext.vNo
```

Figure 6.10 SQL Code Generation for Parameter Context.

Figure 6.11 shows the flow of parameter context processing as well as event notification. When addStk is defined, the ykimaddStk is created and compiled. This file create composite event using method, create*PrimitiveEvent( )* (Creating primitive event means the event node is prepared for the event in the Java LED). In terms of event notification, when addStk occurs, the event name is persisted in notify table. Led.java file will raise addStk. (Raising primitive event means the event is registered in the event node in the JavaLED). At this moment, Event name is inserted into parameter list for parameter context processing. In case of delStk, the process of the event is exactly the same as that of addStk. When both addStk and delStk are raised by the Java LED, the file, ykimaddDel.java can detect the composite event, addDel using event definition and rule definition. Rule action is executed by addDelykim( ). The Java LED is responsible for Parameter context processing. Since addStk and delStk are inserted into parameter list of the Java LED, the Java LED can execute composite event rule action according to parameter context. Note that the ECA Agent should be statically initialized whenever all the files are compiled.



```
EventHandle[] addStk
=
ECAAgent.getEvent
Handles (addStk);

    ECAAgent.insert
(addStk,"eventname",
addStk);
ECAAgent.raiseBegi
nEvent(addStk,this);

}
```

```
If (test1=
null) test1 =
new Led( );
test1.PrimE
vent
(en,tn,vno);
```

```
public static void call_addStk(){
  ECAAgent myAgent =
  ECAAgent.initializeECAAgent();
   addStk =
  myAgent.createPrimitiveEvent("add
  Stk","Led", EventModifier.BEGIN,
  "void addStk()",
  DetectionMode.SYNCHRONOUS);
}
```

```
EventHandle[] delStk
=
ECAAgent.getEvent
Handles
(delStk);

ECAAgent.insert(del
Stk,"eventname",delS
tk);
ECAAgent.raiseBegi
nEvent(addStk,this);
```

```
If (test1=null)
test1 = new
Led( );
test1.PrimEven
t (en,tn,vno);
```

```
 public static void call_delStk(){
 ECAAgent myAgent =
 ECAAgent.initializeECAAgent();
  delStk =
 myAgent.createPrimitiveEvent("delS
tk",
 "Led", EventModifier.BEGIN, "void
 delStk()",
DetectionMode.SYNCHRONOUS);
 }
```

```
ECAAgent myAgent =
ECAAgent.initializeECAAgent();
 addDel = myAgent.createCompositeEvent
(EventType.AND,"event
addDel" ,(EventHandle)ykimdelStk.delStk,
(EventHandle)ykimaddStk.addStk);
 myAgent.createRule("Rule addDel",
addDel,
"Led.True","ykimaddDel.addDelykim", 1,
CouplingMode.DEFAULT,
Context.RECENT);
public static boolean
True(ListOfParameterLists parameterLists)
{   return true;}
public static void addDelykim
(ListOfParameterLists paramLists)
{ This is action part.}
```

Figure 6.11 The Flow of Event Notification and Composite Event Detection

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this thesis, we presented the details of design, architecture, and implementation of the ECA Agent system. There are some sub-functional modules in the ECA Agent system:

- Language Filter
- ECA Parser
- Primitive Event Parser
  - Composite Event Parser
  - Repeated Primitive Event Parser
  - Repeated Composite Event Parser
  - Drop trigger Parser
    - Drop trigger defined on Primitive Event
    - Drop trigger defined on Composite Event
- Persistent Manager
- Java LED

At the same time that we implement these sub-modules, we meet the following goals:

- The ECA Agent is supported by ECA rules.

- Both primitive events and composite events (currently only AND) can be detected.
- Active behaviors (events, rules, actions) are persistent in DBMS.
- Multiple parameter contexts and coupling modes are supported in the ECA Agent system.

The ECA Agent works as a mediator between the SQL server and the clients, and we use JDBC to connect to any SQL server. The design is a generalized method for any RDBMS to extend its active capability.

7.2 Contributions

The contributions of this thesis are as follows:

- The ECA Agent system significantly extends the active capability of any RDBMS. This approach has some advantages:
    - It does not change the SQL Server/Client.
    - It has transparency to the clients.
    - It has extensibility.
- A Full-fledged active capability is supported.
- We use the JDBC to connect the SQL Server. By using the JDBC, you can connect any SQL Server.

7.3 Future Work

In our implementation, we use Oracle as the test database and we extended the active capability of Oracle Universal Database.

We can use the same approach for developing agents for other DBMSs. The ECA Agent system is a module that provides active capabilities to a RDBMS without changing the RDBMS itself. Nowadays, most of the commercial RDBMSs provide users with programming language virtual machine. For example, the Oracle 8.i has the Java Virtual Machine. This will allow us to put all the components into the RDBMS. The Java LED, the Snoop Preprocessor, and the ECA Agent system can run in the same address space as that of an RDBMS. Obviously, This will enhance the performance of the entire.

LIST OF REFERENCES

BER91 Berndtsson, M., “ACOOD: an Approach to an Active Object Oriented DBMS,” *Master’s thesis*, University of Skovde, September 1991.

BER92 Berndtsson, M. and Lings, B., “On Developing Reactive Object_Oriented Databases,” in *IEEE Quarterly Bulletin on Data Science, Special Issue on Active Databases*, 15(1-4):31 –34, 1992.

BER94 Berndtsson, M., “Reactive Object-Oriented Databases and CIM,” in *Proceedings of the 5th International Conference on Database and Expert Systems Applications*, volume 856 of *Lecture Notes in Computer Science*, pages 769--778. Springer, 1994.

CHA89 Chakravarthy, S., “Rule Management and Evalution: An Active DBMS Perspective,” in *Special issue of ACM Sigmod Record on rule processing in databases*, 18(3):20-28, 1989.

CHA93 Chakravarthy, S., Krishnaprasad, V., Anwar, E., and Kim, S.K., “Anatomy of a Composite Event Detector,” in *Technical Report UF-CIS-TR-93-039*, University of Florida, E470-CSE, Gainesville, FL, December 1993.

CHA94a Chakravarthy, S., Krishnaprasad, V., Anwar, E., and Kim, S.K., “Composite Events for Active Databases: Semantics, Contexts and Detection,” in *Proceedings International Conference on Very Large Databases*, Santiago, Chile, 1994, pp. 606-617.

CHA94b Chakravarthy, S., Anwar, E., Maugis, L., and Mishra, D., “Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules,” in *Information and Software Technology*, Vol. 36, pp. 559-568, 1994.

CHA94c Chakravarthy, S. and Mishra, D., “Snoop: An Expressive Event Specification Language for Active Databases,” in *Data and knowledge Engineering*, 13(3), Octorber 1994.

CHA98 Chamberlin, D., “A Complete Guide To DB2 Unviersal Database,” IBM Almaden Prototype Center, 1998.

DAS99 Dasari, R., “Events and Rules for Java: Design and Implementation of a Seamless Approach,” *Master’s thesis*, University of Florida, 1999.

DIA91 Diaz, O., "Rule Management in Object-Oriented Databases: A Unified Approach," in *Proceedings 17th Interntionl Conference on Very Large Data Bases*, Barcelona (Catalonia, Spain), Sept. 1991.

GEH91 Gehani, N. and Jagadish, H. V., "Ode as an active database: Constraints and triggers," in *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 327-336, Barcelona, Spain, September 1991.

KRI94 Krishnaprasad, V., "Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation," in *Master's thesis*, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, 1994.

LEE98 Lee, L., "An Agent-Based Approach to Extending the Native Active Capability of Relational Database Systems," *Master's thesis*, University of Florida, Gainesville, 1998.

STO91 Stonebreaker, M. and Kemnitz, G., "The POSTGRES Next-Generation Database Management System," in *Communications of the ACM* 34(10):78-92, 1991.

VAN96 Vance, D., "Supporting Active Database Semantics in Sybase," *Master's thesis*, University of Florida, Gainesville, 1996.

WID96 Widom, J., "The Starburst Active Database Rule System," in *IEEE Transactions on Knowlede and data engineering*, Vol.8, No. 4: August 1996, pp. 583-595.

BIOGRAPHICAL SKETCH

YoungHun Kim was born on January 3, 1971, in Andong, Korea. He received his Bachelor of Science degree in computer science and engineering from SungKyunKwan University, Seoul, Korea, in March 1995. After his graduation, he worked in Distributed Systems Laboratory at SungKyunKwan Graduate School, Seoul, Korea. He joined the Department of Computer and Information Science and Engineering at the University of Florida in January, 1999. He worked as a research assistant in the Database Systems Research and Development Center of the department. He will receive his Master of Science degree in December 2000. His research interests include RDBMS, Active databases and multi-tier and Web-based database applications.