A GENERALIZED METHOD TO EXTENDING THE ACTIVE CAPABILITY OF RELATIONAL DATABASE SYSTEMS

By

ZECONG SONG

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

Copyright 2000

by

ZECONG SONG

To my parents

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Sharma Chakravarthy, for his continuous guidance and support throughout the course of this research work, and for giving me an opportunity to work on this interesting topic.

I would like to thank Dr. Stanley Su and Dr. Joachim Hammer for serving on my committee.

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

I sincerely thank Hongen Zhang for his invaluable help and fruitful discussions during the implementation of this work. I would like to thank Weera Tanpisuth, Seokwon Yang and Rejesh Dasari for their invaluable help and patience they showed whenever they help me solve problems. 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, by the Rome Laboratory, DARPA.

I also thank my family for their constant support and encouragement throughout my academic career.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS 4

LIST OF TABLES viii

LIST OF FIGURES x

ABSTRACT xii

INTRODUCTION 1

1.1 Triggers in RDBMS 1
1.2 ECA Rules 4
1.2.1 Events 4
1.2.2 Condition 5
1.2.3 Action 5
RELATED WORK 6

2.1 Sentinel 6
2.2 Starburst 6
2.3 Ode 8
2.4 An Agent-Based Approach 9
DESIGN ISSUES OF ECA AGENT 11

3.1 Architecture 11
3.2 Client/Server 13
3.2.1 Socket 14
3.3 Multi-Thread 15
3.3.1 Thread 15
3.4 JDBC 16
3.5 Java LED 18
3.6 Snoop 19
3.6.1 Event Operators 19
3.6.2 Parameter context 20
3.6.3 Coupling modes 22
3.7 DB2 Universal Database 22
IMPLEMENTATION ISSUES 24

4.1 System Tables 24

4.1.1 Table 'SysPrimitiveEvent' 24
4.1.2 Table 'SysCompositEvent' 25
4.1.3 Table 'SysEcaTrigger' 26
4.1.4 Table 'SysContext' 26
4.1.5 Table 'EventContext' 27
4.1.6 Table 'Version' 28
4.2 Naming Mechanism 29
4.3 Pre-Processor 29
4.4 Language Filter 30
4.5 Persistent Manager 32
4.5.1 Architecture of Persistent Manager 32
4.5.2 Generate Persistent Code 33
4.5.3 Restore ECA events and rules 34
IMPLEMENTATION OF PRIMITIVE EVENTS 36

5.1 Syntax of Primitive Events 36
5.2 Parsing and Generating Primitive Event 37
5.3 Creating Triggers on Existing Event 42
5.3.1 Syntax of creating triggers on existing event 42
5.3.2 Implementation of Triggers on Existing Event 43
5.4 Dropping a Trigger on a Primitive Event 45
5.4.1 Syntax of drop trigger command 45
5.4.2 Implementation of drop trigger on primitive event 45
IMPLEMENTATION OF COMPOSITE EVENTS 47

6.1 Syntax of composite event 47
6.2 Composite event parser 48
6.3 Create Events and Rules 52
6.4 Event Notification and Detection 54
6.4.1 Primitive Event Detection 55
6.4.2 Primitive Event Notification 57
6.4.3 Composite Event Detection 58
6.5 ECA Action 60
6.6 Parameter Context 62
CONCLUSIONS, CONTRIBUTIONS AND FUTURE WORK 67

7.1 Conclusions 67
7.2 Contributions 68
7.3 Future work 68
USER MANU 70

Java ECA Agent Server 70
Start the ECA Agent 70
Java ECA Agent Client 71
Start the ECA Agent Client interface 71
DB2 interface 74
DEMO 76

Preparing the Demo 76
Demo 76
FILES USED IN THE DEMO 78

File 1: createSystables.txt 78
File 2: cleanDemo.txt 79
File 3: createEvents.txt 81
File 4: test.txt 83
File 5: results.txt 83
SOME JAVA CLASS FILES 92

File “zsong0addDel.java” 92
File “CallDynamicMethod.java” 93
BIOGRAPHICAL SKETCH 97

LIST OF TABLES

Table Page

Table 4.1 SysPrimitiveEvent 24

Table 4.2 SysPrimitiveEvent 25

Table 4.3 SysCompositEvent 25

Table 4.4 SysCompositEvent 26

Table 4.5 SysEcaTrigger 26

Table 4.6 SysEcaTrigger 26

Table 4.7 SysContext 27

Tablev 4.8 SysContext 27

Table 4.9 EventContext 27

Table 4.10 EventContext 28

Table 4.11 Version 28

Table 5.1 stock 39

Table 5.2 stock_inserted or stock_deleted 40

Table 6.1 stock 52

Table 6.2 stock_inserted_tmp 52

Table 6.3 stock_inserted_tmp 64

Table 6.4 EventContext 64

Table 6.5 SysContext 64

Table 6.6 SysContext 65

LIST OF FIGURES

Figure Page

Figure 1.1 Syntax of Trigger Creation in DB2 2

Figure 3.1 Architecture of this project 12

Figure 3.2 Architecture of ECA Agent 13

Figure 3.3 Client and server communicate through sockets 14

Figure 3.4 Multiple threads in a single program 15

Figure 3.5 APIs for creating events and rules 19

Figure 4.1 Flow Chart of Language Filter 31

Figure 4.2 Architecture of Persistent Manager 32

Figure 4.3 Code for restoring events and rules 34

Figure 5.1 Flow Chart for Parsing and Generating Primitive Event 38

Figure 5.2 Repeat Primitive Event Syntax 43

Figure 5.3 Drop Trigger Syntax 45

Figure6.1: Composite Event Definition 48

Figure6.2: Flow Chart of Composite Event Parser 49

Figure 6.3 file “zsong0addStk.java” 53

Figure 6.4 code for dynamic compile Java file 53

Figure 6.5 code for register events 54

Figure 6.6 Java file “Led.java” 59

Figure 6.7 parameter context processing 65

Figure A.1 ECA Agent Server Interface....70

Figure A.2 ECA Agent Server DOS Environment....71

Figure A.4 ECA Client Interface....72

Figure A.5 DB2 Interface....74

Figure B.1 ECA Agent Client Interface....77

Abstract of Dissertation 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 METHOD TO EXTENDING THE ACTIVE CAPABILITY OF
RELATIONAL DATABASE SYSTEMS
By

Zecong Song

August 2000

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

In the database research community, active databases have received widespread attention for at least ten years. Active databases have been the focus of several researches to extend the functionality of traditional (passive) databases. Active databases generally use active rules to express their active behavior. These rules are called ECA rules (Event-Condition-Action). There are several commercial databases using active rules, such as DB2, Informix or Oracle. The problem is that the active rules in these databases are very limited. We discussed this problem in this thesis. And we also proposed a general method of turning a passive database to an active database.

We add a mediator between the SQL server and the clients termed ECA Agent. ECA rules are completely supported in the ECA Agent and both primitive events and composite events can be detected in the ECA Agent. Java LED are used to detect the composite events. JDBC is used as a bridge to connect between SQL server and the SQL requirements.

ECA Agent also provides all the usual functionality of a conventional passive database system. And the active behaviors (events, rules and actions) become a persistent part of the database.

We present the architecture and implementation details of ECA Agent in this thesis. DB2 are used as the test database.

CHAPTER 1
INTRODUCTION

Active database management systems (ADBMS's), as examples of active systems, are able to monitor and react to specific circumstances of relevance to an application [Nor98]. Traditional DBMSs are passive in the sense that commands are executed by the database when requested by the user or application program. However, some situations cannot be modeled effectively by passive systems.

Active database systems enhance traditional database functionality with powerful rule processing (or "trigger") capabilities. Active database systems are significantly more powerful than their passive counterparts in the following aspects [JEN96]:

- Active database systems can efficiently perform functions that in passive database systems must be encoded in application.
- Active database systems suggest and facilitate applications beyond the scope of passive database systems.
- Active database systems can perform tasks that require special-purpose subsystems in passive database systems.

1.1 Triggers in RDBMS

Several commercial relational database management systems support active database rules, usually referred to as triggers. The functionality of commercial database trigger systems is generally rather limited as compared to the active database research prototypes, such as 'Sentinel', 'Starburst' and 'Ode'. Nevertheless, the capabilities of

many commercial systems are already sufficient to provide relatively complex active database behavior. Figure 1.1 illustrates the syntax for the creation of a trigger in DB2 [DON98].

```
>>-CREATE TRIGGER--trigger-name---+-NO CASCADE BEFORE-+---------->
                                  +-AFTER-------------+

>--+-INSERT---------------------------+ON--table-name----------->
   +-DELETE---------------------------+
   +-UPDATE--+----------------------+-+
             |    +-,------------+  |
             |    V              |  |
             +-OF---column-name--+--+

>--+----------------------------------------------------------+->
   |              +-------------------------------------------+  |
   |              V (1) (2)        +-AS-+                     |  |
   +-REFERENCING-------------+-OLD-+----+--correlation-name--++--+
                             |     +-AS-+                      |
                             +-NEW-+----+--correlation-name--+
                             |           +-AS-+                |
                             +-OLD_TABLE-+----+--identifier--+
                             |           +-AS-+                |
                             +-NEW_TABLE-+----+--identifier--+

>--+-FOR EACH ROW-------------+MODE DB2SQL---| triggered-action |-><
   | (3)                      |
   +------FOR EACH STATEMENT--+

triggered-action

|--+--------------------------------+---------------------------->
   +-WHEN--(--search-condition--)--+

>--+-triggered-SQL-statement-----------------------------+-----|
   |                +-----------------------------+       |
   |                V                             |       |
   +-BEGIN ATOMIC----triggered-SQL-statement--;--+--END--+
```

Figure 1.1 Syntax of Trigger Creation in DB2

From the above trigger syntax we see that DB2 do support active database rules.

But currently, DB2 active capabilities as the other commercial active database suffer from four main shortcomings [JEN96]:

1. They lack standardization. Consequently, the various products have a wide variance in both the syntax and execution behavior of triggers. This results in a lack of uniformity, and the inability to use trigger applications on differing products.
2. They lack clearly defined execution semantics. A number of alternative constructs may be provided (such as both tuple-level and statement-level triggering, or both immediate and deferred execution), but often it is not specified precisely how triggers will behave when multiple triggers with different options are present.
3. They lack a number of useful “advanced features” that have been included in research prototypes. Some of the them are application-specific events, event composition techniques, binding of events to conditions and of conditions to actions, use of net effects, use of enhanced transaction models to support sophisticated coupling modes or parallelism, lack of external procedure calls, and so on.
4. They often incorporate a number of restrictions, such as limitations on the number of triggers that may be defined, or on the interactions between triggers.

Because of these shortcomings, the development of an active database needs to consider the following issues [JEN96]:

- An active database system must provide all the usual functionality of a conventional passive database system. Meanwhile, it is desirable that the

performance of conventional database tasks is not degraded by the fact that the database system is active.

- An active database system must provide some mechanism for users and applications to specify the desired active behavior, and these specifications must become a persistent part of the database.
- An active database system must efficiently implement any active behavior that can be specified; it must monitor the behavior of the database system and, when appropriate, automatically initiate additional behavior.
- An active database system must provide database design and debugging tools similar to those provided by conventional database systems, extended to incorporate active behavior.

1.2 ECA Rules

Active database systems are centered around the notion of rules [JEN96]. Rules in active database systems are defined by users or applications. They specify the desired active behavior. In most general form, active database rules consist of three parts: Event, Condition, and Action. We also denote it ECA rules.

1.2.1 Events

In an active database rule, the event specifies what causes the rule to be triggered. In a relational database system, event can be **insert**, **delete**, or **update** on a particular table. Types of event can be Primitive event and Composite event.

- Primitive Event: event that is pre-defined in the system. In relational database system, primitive event can be insert, deleted, or update on a particular table.

- Composite Event: event that is formed by applying a set of operators to primitive and composite events.

1.2.2 Condition

In an active database rule, the condition specifies an additional condition to be checked once the rule is triggered and before the action is executed. In ECA rules, the condition is generally optional, or a dummy condition true can be given.

1.2.3 Action

In an active database rule, the action is executed when the rule is triggered and its condition is true. Actions may update the structure of the database, perform some behavior invocation within the database or an external call.

In our research, we use LED (Java version) to implement the ECA rules. We'll talk about the detail of LED in Chapter 3.

CHAPTER 2
RELATED WORK

The field of active database research has been one of the most prominent areas of database research during the late 1980s and early 1990s. There are a lot of research projects in this field over these years. We will review some of the most important projects in this chapter.

2.1 Sentinel

Sentinel (from University of Florida) is an integrated active OODBMS that supports Event-Condition-Action (ECA) rules and their management. It uses the Open OODB Toolkit (from Texas Instruments, Dallas, Texas) as the underlying platform. Event and rule specifications are seamlessly incorporated into the C++ language. Any method of an object class is a potential primitive event. Applying a set of operators to primitive events and composite events can form composite events. Sentinel supports multiple rule executions, nested rule executions as well as prioritized rule executions. Sentinel supports all the four parameter-contexts specified in HiPAC, namely, *recent*, *chronicle*, *continuous* and *cumulative* contexts. Sentinel currently supports immediate and deferred modes of rule execution [CHA94][CHA94a][CHA94b].

2.2 Starburst

The Starburst system is a prototype extensible relational DBMS developed at the IBM Almaden Research Center [JEN96+]. Startburst's extensibility allows the database

system to be customized for advanced and non-traditional database applications. One of Starburst's extensions 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.

The syntax of the Starburst rule language is based on the extended version of SQL supported by the Starburst database system. The Starburst rule language includes five commands for defining and manipulating rules: create rule, alter rule, deactivate rule, activate rule, and drop rule.

The syntax of **create rule** is:

create rule *name* **on** *table*

When *triggering-operations*

[**if** *condition*]

then *action-list*

[**precedes** *rule-list*]

[**follows** *rule-list*]

The *name* names the rule, and each rule is defined on a *table*. Square brackets indicate clauses that are optional.

The components of a rule can be changed after the rule has been defined. This is done using the **alter rule** command. The syntax of this command is:

alter rule *name* **on** *table*

[**if** *condition*]

[**then** *action-list*]

[**precedes** *rule-list*]

[**follows** *rule-list*]

[**nopriority** *rule-list*]

An existing rule can be deleted by issuing the **drop rule** command:

drop rule *name* **on** *table*

We can deactive rules using the **deactivate rule** command:

deactivate rule *name* **on** *table*

To reactivate a rule that has been deactivated, use the **activate rule** command:

activate rule *name* **on** *table*

From the above syntax of rule language we see that the Starburst rule language is flexible and general.

2.3 Ode

Ode is an object-oriented database that based on the C++ object paradigm. The primary interface for the Ode database is the database programming language O++, which is an upward-compatible extension of the C++. O++ extends C++ by providing facilities suitable for database applications, including the association of constraints and triggers with objects.

Ode provides two kinds of active facilities: “constraints” for maintaining database integrity and “triggers” for automatically performing actions depending upon the database state [GJ91].

Ode supports two kinds of triggers: once-only (default) and timed triggers. A once-only trigger is automatically deactivated after the trigger has “fired”, and it must then be explicitly activated again, if desired. A timed trigger must fire within the specified period.

Ode trigger model is an event-action (E-A) model. When an event occurs, the associated action is executed.

Ode supports primitive events and composite events. Primitive events are defined and composite events are constructed by applying operators to primitive events. The basic events that are supported are object state events. The event operators supported are prior, sequence, first, firstAfter, happened, every, prefix, etc.

2.4 An Agent-Based Approach

Lijuan Li implemented “An Agent-Based approach to extending the native active capability of relational database systems” in May 1998 from University of Florida [LIJ98]. She implemented an ECA Agent, which is a mediator between clients and Sybase SQL Server. She used Sybase Gateway Open Server to extend the active capability of Sybase.

In this thesis, we tried to solve the same problem, which is to extend the native active capability of relational database systems. So we use the same idea to design our project. The difference here is her design is based on Sybase Gateway Open Server and

our design is not based on any specified RDBMS. We want to find a generalized method that any RDBMS can use it.

In the next chapters, we'll discuss our design and implementation in detail.

CHAPTER3
DESIGN ISSUES OF ECA AGENT

This chapter discusses the design issues of ECA Agent. It includes the architecture, Java LED, and Snoop. We also include the basis of JDBC, client/server and multi-thread in this chapter because these techniques are the basic techniques we used in this project.

3.1 Architecture

The goal of this project is to implement a generalized method to extend the active capability of RDBMS. It's a generalized method, so we must design it as it can be used by any RDBMS, such as Oracle, DB2, Informix, etc. Figure 3.1 shows the architecture of this project. From this figure, we can see that there are multi-clients for each RDBMS server, and one ECA Agent for one SQL server. ECA Agent is a mediator between the client and the server. When clients have some requests, these requests must first be sent to ECA Agent, and then the ECA Agent sends it to SQL server.

Here, ECA Agent will do some work to extend the active capability of RDBMS. Figure 3.2 shows the architecture of ECA Agent.

From Figure 3.2 we can see that ECA Agent includes the following function modules:

- Language Filter: when clients have request, the request first is sent to Language Filter. Language Filter will filter the request. If it's ECA command, it is sent to ECA Parser, otherwise sent it to JDBC.

- ECA Parser: ECA Parser will parse the ECA command. If there are no errors, the ECA Parser will create corresponding events and rules which depend on the LED. Also, ECA Parser will send the events and rules to the Persistent Manager for persistent storing.
- Persistent Manager: All events and rules defined by a client need to be persistent. Persistent Manager will store the information using RDBMS. When ECA Agent starts or recovers, Persistent Manager restores and creates all events and rules.
- JDBC: We use JDBC to connect between SQL server and client. JDBC gets request from client and sends it to SQL server, and then JDBC gets result from SQL server and sends it back to client.
- LED (Local Event Detector): In RDBMS, trigger can only detect primitive events. So we use LED to detect composite events.
- ECA Action: When event occurs, the action defined on this event should be executed. In our project, ECA Actions are SQL statements. It will call JDBC to send the SQL statements to SQL server and get results.

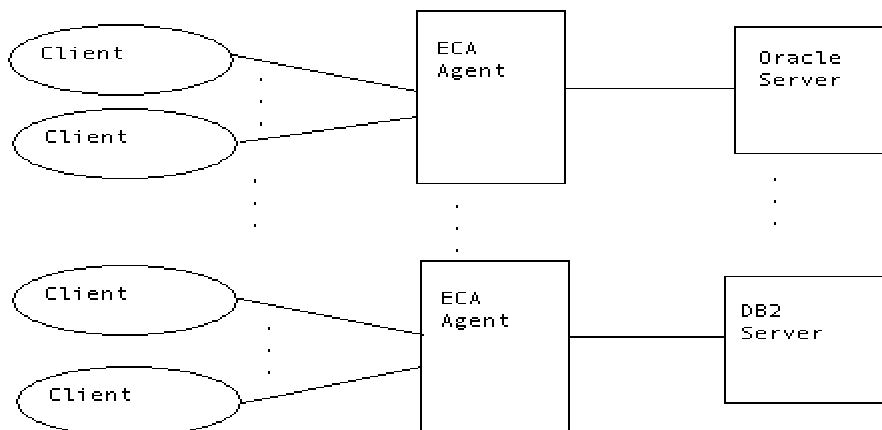


Figure 3.1 Architecture of this project

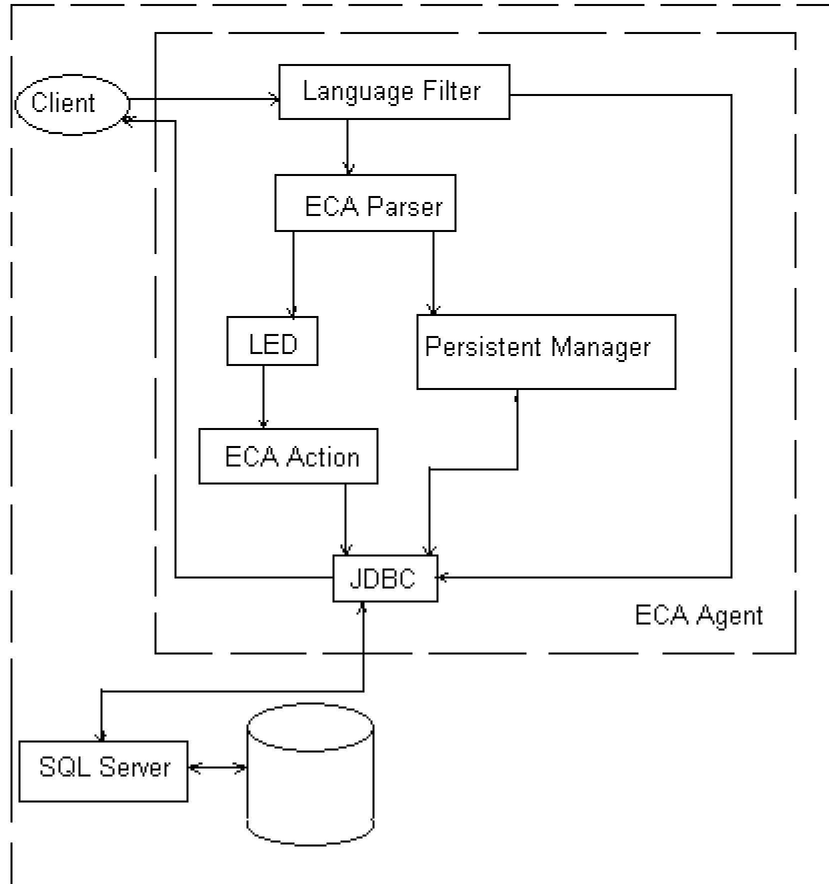


Figure 3.2 Architecture of ECA Agent

3.2 Client/Server

Today's popular database software tools are based on the client/server paradigm. Our program also is based on the client/server paradigm. We have "Java_ECA_Agent_Server" program for the server side and "Java_ECA_Agent_Client" for the client side. Socket classes are used to represent the connection between a client program and a server program.

3.2.1 Socket

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits to listen to the socket for a client to make a connection request.

On the client side, the client knows the hostname of the machine on which the server is running and the port number to which the server is connected. To make a connection request, the client tries to rendezvous with the server on the server's machine and port.

If everything goes well, the server accepts the connection.

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets. Figure 3.3 shows the communication between client and server [JavaTutorial].

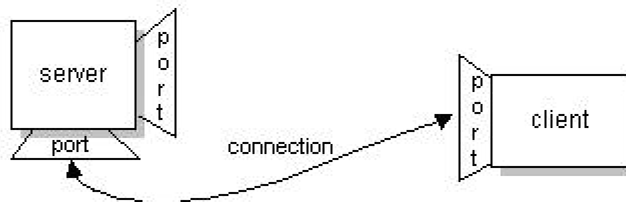


Figure 3.3 Client and server communicate through sockets

3.3 Multi-Thread

In a typical server, you want to be able to deal with many clients at once. The solution is multithreading. In Java, multithreading is about as simple as possible because threading in Java is reasonably straightforward. Making a server that handles multiple clients is relatively easy.

3.3.1 Thread

A thread--sometimes called an *execution context* or a *lightweight process*--is a single sequential flow of control within a program. You use threads to isolate tasks. Each thread is a sequential flow of control within the same program (the browser).

Multiple threads in a single program are illustrated by Figure 3.4 [Java Tutorial].

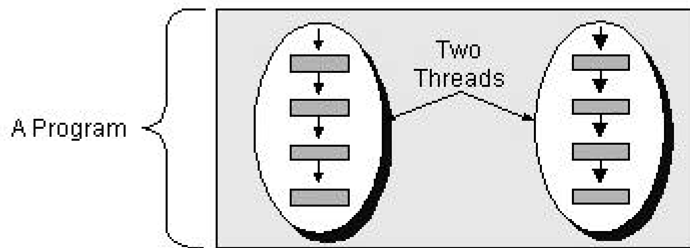


Figure 3.4 Multiple threads in a single program

Multiple threads run at the same time and perform different tasks. The server can service multi-clients simultaneously through the use of threads - one thread for each client connection. The basic flow of logic in such a server is this:

```
while (true) {
   accept a connection ;
   create a thread to deal with the client ; }
end while
```

The thread reads from and writes to the client connection as necessary.

3.4 JDBC

The Java Database Connectivity (JDBC) is developed from the need to enable Java applications to connect to SQL databases. It consists of a set of classes and interfaces written in the Java programming language. JDBC provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API. Because of Java's features, it is uniquely suitable for network access to a variety of databases. And because Java itself is a platform-independent language, there is a compelling reason to develop applications that are independent of a particular database vendor [ART99].

In this thesis, we use JDBC to send SQL statements to RDBMS, so we'll talk about the basic JDBC programming.

1. Load driver

   The first step in using JDBC is to load the JDBC driver. This is usually accomplished using the **forName** static method of the **class** object. The call is made as follows:

   Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");

   When this call is made, the Java system searches for the class requested and loads the driver.

2. Create connection

   The loading of the JDBC database driver does not connect to the database. It merely creates an environment in the program where this can be done. Before any database-specific SQL statements can be executed, a connection must be established to the database. This is accomplished through a call to the **getConnection** method in

DriverManager class to find a specific driver that can create a connection to the URL requested. The call is made as follows:

String url = "jdbc:db2: database";

Connection con = DriverManager.getConnection (url, username, password);

3. Create statement

In order to interact with the database, SQL statements must be executed. This requires that a **Statement** object to be created to manage the SQL statements. This is accomplished with a call to the **createStatement** method in **Connection** class as follows:

Statement stmt = con.createStatement( );

This call creates a **Statement** object using the established database connection. The **Statement** class provides methods for executing SQL statements and retrieving the results from the statement execution.

4. Execute statement and return **ResultSet** or result count

The SQL **Statement** object does not have a specific SQL statement associated with it. The SQL statement to be executed is determined when the call to **executeQuery** is made, as follows:

String qs = "select * from stock";

ResultSet rs = stmt.executeQuery(qs);

This call sends the query to the database and returns the results of the query as a **ResultSet**.

5. Iterate **ResultSet** if returned

The **ResultSet** represents the collection of results from the query. First, you must make a call to the first element of the result set, as follows:

Boolean more = rs.next();

The call to the next method returns a boolean value. The boolean value of true indicates that the call was successful and the pointer is positioned, thus there is data to retrieve. A boolean value of false indicates that the call was unsuccessful and there are no rows to retrieve.

Next, we can get the first value of the first column of the result set as follows:

returnstring = rs.getString(1);

6. Close the result set, statement, and the connection

rs.close();

stmt.close();

conn.close();

3.5 Java LED

In our research, we use Java LED to detect composite events. Java LED is the Java version of Local Event Detector. It incorporates active capability in a Java environment.

In Java LED there is an event detector for detecting events in Java applications and executing rules defined on events. Both primitive event and composite event have been detected in various parameter contexts. It also implemented most event operators for composite event, they are: AND, OR, SEQUENCE, NOT, APERIODICA(A), APERIODIC-STAR(A*), PLUS, PERIODIC(P) and PERIODIC-STAR(P*).

In Figure 3.5, we show some APIs that are used to create events and rules.

```
public EventHandle createPrimitiveEvent(String eventName,
                                        String className,
                                        EventModifier eventModifier,
                                                 String methodSignature)

public EventHandle createCompositeEvent(EventType eventType,
                                        String eventName,
                                        EventHandle leftEvent,
                                        EventHandle rightEvent)

public void createRule(String ruleName,
                       EventHandle eventHandle,
                       String condName,
                       String actionName)
```

Figure 3.5 APIs for creating events and rules

3.6 Snoop

Snoop is the event specification language used in Sentinel for specifying ECA rules. Snoop defines the event expressions and a set of event operators for constructing composite events.

3.6.1 Event Operators

The Snoop event operators and the semantics of composite events formed by these event operators are as follows:

- **OR (V):** E1 V E2, occurs when either E1 occurs or E2 occurs.
- **AND (^):** E1 ^ E2, occurs when both E1 and E2 occurs, irrespective of their order of occurrence.

- **SEQUENCE (;):** E1;E2, occurs when E2 occurs provided E1 has already occurred. This implies that the time of occurrence of E1 is guaranteed to be before the time of occurrence of E2.
- **NOT (~):** ~(E2) [E1 ,E3], detects the non-occurrence of the event E2 in the closed interval formed by E1 and E3.
- **A (Aperiodic):** A(E1,E2,E3), detects the occurrence of E2 during the half-open interval formed by E1 and E3.
- **A*:** A*(E1, E2, E3), detects when E3 occurs provided E1 has already occurred. The occurrences of E2 are accumulated during the half-open interval formed by E1 and E3. A* is a cumulative variant of the A operator.
- **P (Periodic):** P (E1, E2, E3), detects for every time period specified by E2 during the half-open interval (E1, E3], where E2 is a relative temporal event.
- **P*:** P*(E1, E2, E3), detects only once when E3 occurs provided the E1 has already occurred. The time specified in E2 is accumulated whenever E2 occurs. P* is a cumulative variant of P operator.

3.6.2 Parameter context

Snoop supports *parameter context*. Parameter contexts indicate the order in which successive occurrences of the same constituent events are grouped [CHA94] [CHA94b]. The notion of parameter contexts was primarily introduced for the purpose of capturing application semantics while computing the parameters of composite events when they are not unique. They serve the purpose of disambiguating the parameter computation and at the same time accommodate a wide range of application requirements [KRI94].

The parameter contexts proposed by Snoop are *recent, continuous, cumulative, and chronicle*. The contexts are defined using the notion of *initiator* and *terminator* events. An initiator of a composite event is a constituent event that can start the detection of the composite event whereas a terminator is a constituent event that can detect the occurrence of the composite event [CHA94].

- **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. When an event occurs, the event is detected and all the occurrences of events that cannot be the initiators of that event in the future are deleted.
- **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, i.e., in chronological order of occurrence. In this context, the same primitive event occurrence is used at most once for computing the parameters of the composite 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. This context is especially useful for tracking trends of interest on a sliding time point governed by the initiator event. There is a subtle difference between the chronicle and the continuous contexts. In the former, pairing of the initiator is with a unique terminator of the event whereas in the latter multiple initiators are paired with a single terminator of that event.
- **Cumulative:** in this context, for each constituent event, all occurrences of the event are accumulated until the composite event is detected. Whenever a composite event is detected, all the constituent events that are used for detecting that composite event are

deleted. Unlike the continuous context, an event occurrence does not participate in two distinct occurrences of the same event in the cumulative context.

3.6.3 Coupling modes

Coupling modes specify when a rule is to be executed relative to the event firing the rule. Three coupling modes are described below:

- **Immediate:** in this coupling mode, the fired rule is executed immediately after the event is detected.
- **Deferred:** in the deferred mode, the execution of a fired rule is deferred to the end of the transaction. In our case (a non-transaction-based environment) deferred rules are executed by an explicit event raised by the application.
- **Detached:** in the detached mode, the rule is executed in a separate transaction but after the triggering transaction has committed. Since there are no transaction in our case, the detached mode is not supported.

3.7 DB2 Universal Database

In this thesis, we use DB2 as the test RDBMS.

DB2 Universal Database (UDB) is developed at IBM's laboratories in Toronto, Canada, and San Jose, California. UDB uses new technology based on the Starburst architecture developed at Almaden Research Center. UDB is portable to many hardware and software platforms, including Intel/Windows NT, Intel/OS/2, PowerPC/AIX, SPARC/Solaris, and HPPA/HPUX.

UDB is a substantial advance over traditional relational systems. It integrates object-oriented ideas with the SQL language to produce an object-relational database management system. It includes major innovations in query optimization, recursive

queries, active databases, and stored procedures. It integrates technology from DB2 Parallel Edition to support parallel processing, both on symmetric multiprocessors and on massively parallel, shared nothing platforms. UDB has also made substantial advances in usability, providing graphical user interfaces and wizards to help you perform administrative tasks.

We use DB2 as the test database because DB2 trigger has some active database capabilities, but it has some limitations. We extend its trigger command to extend the active capability.

CHAPTER 4
IMPLEMENTATION ISSUES

In this chapter, we describe the implementation of ECA Agent. In Section 4.1, we detail the format and purpose of the system tables because in this research we use these system tables to store persistent information for primitive events, composite events, and triggers. Naming mechanism is introduced in Section 4.2. Preprocessor, Language Filter, and Persistent Manager are described in Section 4.3, 4.4 and 4.5 respectively.

4.1 System Tables

4.1.1 Table 'SysPrimitiveEvent'

Table "SysPrimitiveEvent" is used to store the information for primitive events that the user defined on a table for an operation. The structure of this table is illustrated in Table 4.1.

Table 4.1 SysPrimitiveEvent

DBName	UserName	EventName	TableName	Operation	BeAfOperation	Timestamp	VNo

In this table, BeAfOperation means "before/after" and Vno is used to record the occurrence of this event.

If we have a primitive event addStk defined on table "stock" for "insert" operation and this primitive event has the following definition:

Create trigger t_addStk after insert on stock event addStk ….

Then one tuple should be added into the table “SysPrimitiveEvent”, which is shown in Table 4.2.

Table 4.2 SysPrimitiveEvent

DBName	UserName	EventName	TableName	Operation	BeAfOperation	Timestamp	VNo
‘ECAdb’	‘zsong’	‘addStk’	‘stock’	‘insert’	‘after’	Current timestamp	0

Every time when the event ‘addStk’ occurs, VNo will be increased by 1.

4.1.2 Table ‘SysCompositEvent’

Table “SysCompositEvent” is used to store the information for composite events that the user defined. The structure of this table is shown in Table 4.3.

Table 4.3 SysCompositEvent

DBName	UserName	EventName	EventDescribe	Timestamp	Coupling	Context	Priority

In Table 4.3, Coupling mode can be ‘IMMEDIATE’, ‘DEFERED’ or ‘DETACHED’. Context can be ‘RECENT’, ‘CHRONICLE’, ‘CONTINUOUS’, or ‘CUMULATIVE’. Priority is used to define the priority of this composite event.

If we have a composite event addDel defined as follows:

Create trigger t_addDel event addDel = addStk ^ delStk RECENT ….

Then, the table “SysCompositEvent” should have one more tuple like in Table 4.4.

Table 4.4 SysCompositEvent

DBName	UserName	EventName	EventDescribe	Timestamp	Coupling	Context	Priority
‘ECAdb’	‘zsong’	‘addDel’	‘addStk ^ delStk’	Current timestamp	IMMEDIATE	RECENT	1

4.1.3 Table ‘SysEcaTrigger’

Table “SysEcaTrigger” is used to store the information for triggers that the user defined. The structure of this table is shown in Table 4.5.

Table 4.5 SysEcaTrigger

DBName	UserName	TriggerName	TriggerProc	Timestamp	EventName

In Table 4.5, ‘TriggerProc’ is the procedure defined on this trigger. That means if the trigger fires, the procedure will execute. ‘EventName’ is the event name that the trigger is defined on. Different trigger can be defined on same event.

If we have a trigger defined as follows:

Create trigger t_addStk after insert on stock event addStk…

Then, the table “SysEcaTrigger” should have one more tuple shown in Table 4.6:

Table 4.6 SysEcaTrigger

DBName	UserName	TriggerName	TriggerProc	Timestamp	EventName
‘ECAdb’	‘zsong’	‘t_addStk’	‘t_addStk_Proc’	Current timestamp	‘addStk’

4.1.4 Table ‘SysContext’

Table “SysContext” is used to store the occurrence number for a certain event defined on a certain context. This information can be used for composite events. The structure of this table is illustrated in Table 4.7.

Table 4.7 SysContext

EventName	Context	Vno

In Table 4.7, ‘VNo’ is used to record the occurrence number of a certain event for a certain context.

If we have an event ‘addStk’ defined on the context ‘RECENT’, and this event has occurred for three times (suppose no other event occurs), then tuples will insert into the table “SysContext” as shown in Table 4.8.

Tablev 4.8 SysContext

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

4.1.5 Table ‘EventContext’

When the user creates composite event, table “EventContext” is used to store the information of primitive event and context. When primitive event occurs, we can insert tuples into table ‘SysContext’ using the joining results from table ‘EventCotext’ and ‘SysPrimitiveEvent’. The structure of table ‘EventContext’ is shown in Table 4.9.

Table 4.9 EventContext

EventName	Context

If we have a composite event addDel defined as follows:

Create trigger t_addDel event addDel = addStk ^ delStk RECENT ….

Then, the table "EventContext" should have two more tuples shown in Table 4.10.

Table 4.10 EventContext

EventName	Context
'addStk'	'RECENT'
'delStk'	'RECENT'

4.1.6 Table 'Version'

Table "Version" is used to store the occurrence number for a primitive event. We can get the version number from the following SQL statements:

```
delete from version;
insert into version select VNo from SysPrimitiveEvent where eventname='eventname'
```

Table 4.11 Version

VNO

We use table 'Version' because we want to simplify the SQL query language. For example, if we want to insert tuple into table 'stock_inserted', we can write SQL statements like this if we have table 'Version':

```
insert into stock_inserted select * from stock, version;
```

Otherwise, if we do not have table 'Version', we need to write SQL statements like this:

```
insert into stock_inserted
   select Symbol, Co_name, price, date, VNo from stock, SysPrimitiveEvent
        where SysPrimitiveEvent.eventname='addStk'
```

4.2 Naming Mechanism

Relational DBMS, such as Sybase, Oracle and DB2 support multi-user, multi-database environment, a user can assign a name for an object in the system, and the system will turn it into a system-wide internal name. For example, user 'mark' uses database 'mining' in DB2, and if he creates a trigger 'miningTrigger', then the system-wide internal name for trigger 'miningTrigger' is 'mining.mark.mingTrigger'.

In our research, we also follow the system-wide internal name for an object. That means, when the user creates an object, we'll turn its name to system-wide internal name use the following mechanism:

DatabaseName.userName.objectName

4.3 Pre-Processor

We use pre-processor to parse the composite event in this research. Pre-processor is developed using JavaCC (Java Compiler Compiler, Version 1.0) which is developed by Sun Microsystems. JavaCC has the following features:

- 100% PURE JAVA (hence portable). JavaCC is certified 100% PURE JAVA. This means JavaCC can run on any Java compliant platform version 1.0.2 or later. JavaCC has been successfully used on over 40 different hardware/software platforms.
- TREE BUILDING PREPROCESSOR. JavaCC comes with a tree building pre-processor called JJTree.
- DOCUMENTATION GENERATION. A translator that converts grammar files to documentation files (optionally in html) is now an integral part of the JavaCC release. This translator is called JJDoc.

Because of the above nice features we select JavaCC as the pre-processor tool. In this research, we use pre-processor to parse the composite event. The input for the parser is a composite event. When the pre-processor parses the composite event, if there are any syntax errors, the parser will give the error message. If there is no syntax error, the parser will generate two files, one is "eventlist.txt" and the other is "compositevent.txt".

File "eventlist.txt" is used to keep events, which consist of this composite event. Primitive events (leaves in this composite event tree) will be inserted into table "EventContext". When primitive events occurs, tuples that is the results of joining table 'EventContext' and 'SysPrimitievEvent' will be inserted into table "SysContext".

File "CompositEvent.txt" is used to keep the content of creating composite event in LED. For example, if we have a following composite event definition:

Create trigger t_addDel event addDel = addStk ^ delStk RECENT ….

Then the content of file 'CompositeEvent.txt' looks like this:

EventHandle addDel = myAgent.createCompositeEvent(EventType.AND, "event addDel", addStk, delStk)

And the content of file 'eventlist.txt' looks like this:

addStk delStk

4.4 Language Filter

When the client sends a request to the ECA_Agent_Server, first the request goes to the 'Language Filter'. The 'Filter' will analyze the request first, and then send the request to the right way. The flow chart is illustrated in Figure 4.1.

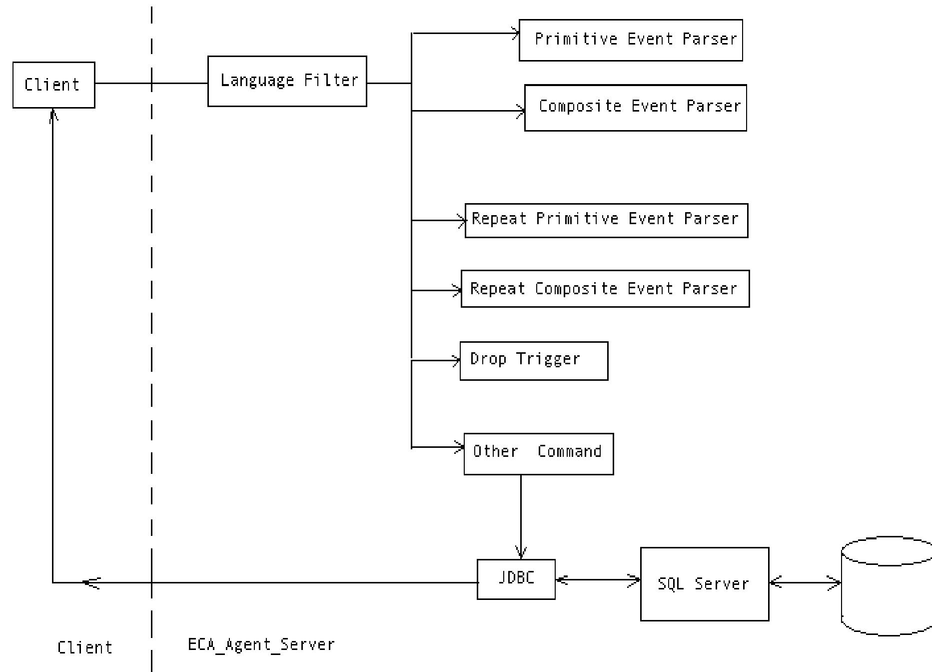


Figure 4.1 Flow Chart of Language Filter

As we see from the above flow chart, if the request is about primitive event or composite event, there are five sub-modules:

- Primitive Event Parser: this module parses the primitive event when it was defined at the first time.
- Composite Event Parser: this module parses the composite event when it was defined at the first time.
- Repeat Primitive Event Parser: this module parses the repeat primitive event, which means if a trigger is defined on an existing primitive event, we'll use this module to parse it.

- Repeat Composite Event Parser: this module parses the repeat composite event. When a trigger defined on an existing composite event, we'll use this module to parse it.
- Drop Trigger: when the user's request is 'drop trigger' command, we'll call this module.

If the request is 'Other Command', it means the request is not about primitive event or composite event. The ECA_Agent_Server will send the request to 'JDBC', and 'JDBC' sends the request to SQL Server. At last, the results are returned to the client.

4.5 Persistent Manager

We can get some values from memory when a program is running. But when the program terminates, the values stored in memory will disappear. In order to keep the values that we've gotten from the program, we need to use 'Persistent Manager'. In this research, 'Persistent Manager' is used to keep ECA rules and generate persistent code. Also, when ECA Agent starts and recovers, Persistent Manager will restore the events and rules. Next we'll talk this in detail.

4.5.1 Architecture of Persistent Manager

In this research, we use JDBC to develop a generalized method for all RDBMS to extend the active capabilities. To implement the 'Persistent Manager', we call JDBC to connect to SQL server, as in Figure 4.2.

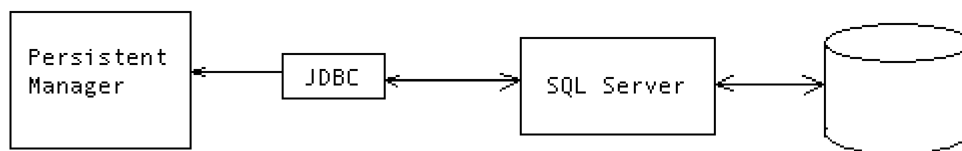


Figure 4.2 Architecture of Persistent Manager

When persistent command is sent to the 'Persistent Manager', the 'Persistent Manager' will send the command to JDBC and JDBC will send the command to SQL server. Finally the command will be executed in the server.

4.5.2 Generate Persistent Code

For generating persistent code, the 'Persistent Manager' will do the following tasks:

- Maintain ECA Agent system tables.
- Insert tuples into ECA Agent system tables.
- Create trigger command for primitive event.
- Keep track of the occurrence of each primitive event.

Now, we give examples to interpret how the 'Persistent Manager' works:

When the client defines primitive event, for example:

Create trigger t_addStk after insert on stock event addStk

REFERENCING NEW_TABLE AS newtable

FOR EACH STATEMENT MODE DB2SQL

Insert into stock_copy select * from newtable

The Persistent Manager will take the following actions:

1. Insert tuples into system tables:

   Insert into SysEcaTrigger values('zsong', 't_addStk', 't_addStk_proc', current timestamp, 'addStk')

   Insert into SysPrimitiveEvent values('zsong0', 'addStk', 'stock', 'insert', 'after', current timestamp, 0)

2. Keep track of the occurrence of the primitive event:

   Update SysPrimitiveEvent set vNo = vNo+1 where eventname = 'addStk';

3. Create triggers for primitive event:

```
Create trigger t_addStk after insert on stock
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
    Insert into stock_copy  select * from newtable;
```

4.5.3 Restore ECA events and rules

When the user defines events and rules, the Persistent Manager inserts tuples into system tables. In order to register the events and rules using Java Led, our program generates a Java file, compiles it and then registers it at the run time of ECA Agent (we'll discuss this in detail in chapter 6).

If we restart the ECA Agent, we need to restore the events and rules. The tuples that we inserted into system tables are still in the system tables so we do not need to worry about this. The only thing we need to do is to re-register the events and rules using Java Led.

Because we generated a file for event and we already compiled it, we just need to call the API to register it. This is the same as we call the API when we register it at the first time. We'll use the following code to do this:

```
// to do "zsong0addStk.call_addStk();", use the following code
CallDynamicMethod.ExecuteMethod(classname,"call_" + eventname, null, null);

//class "zsong0addStk"
public class zsong0addStk{
   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);
         }
}
```

Figure 4.3 Code for restoring events and rules

CHAPTER 5
IMPLEMENTATION OF PRIMITIVE EVENTS

In this chapter, we describe how to implement primitive events in ECA_Agent_Server. First, the syntax of primitive events is introduced in Section 5.1. In Section 5.2, we describe the parsing and generating primitive event. How to create triggers on existing event and how to drop a trigger on a primitive event are described in Section 5.3 and 5.4 respectively.

5.1 Syntax of Primitive Events

In this research we extend the trigger definition to extend the active capability of RDBMS. For example, if we have the "create trigger" syntax of DB2 as follows:

```
Create trigger t_addStk after insert on stock
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
        Insert into stock_copy  select * from newtable
```

We'll extend the create trigger syntax as follows for the primitive event:

```
Create trigger t_addStk after insert on stock  event addStk
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
        Insert into stock_copy  select * from newtable
```

As we can see we just add the primitive event definition into 'create trigger' syntax.

The primitive event definition is:

event *event_name* [*coupling_mode] [parameter_context] [priority*].

Where

parameter_context := RECENT|CHRONICLE|CONTINUOUS|CUMULATIVE.

coupling_mode := IMMEDIATE|DEFERED|DETACHED.

priority := positive integer.

The default coupling mode is IMMEDIATE and the default parameter context is RECENT.

5.2 Parsing and Generating Primitive Event

Figure 5.1 is the flow chart for parsing and generating primitive event

From Figure 5.1, we can see that there are four steps for parsing and generating primitive event:

1. Syntax check: check if there are syntax errors for this primitive event. If there are syntax errors, return error message to the client.
2. Duplicate Name check: check the trigger name is duplicate or not, because in RDBMS, trigger name can not be duplicate. If the trigger name is duplicated, return error message to the client.
3. If there are no errors, create primitive event using LED.
4. Generate persistent code.

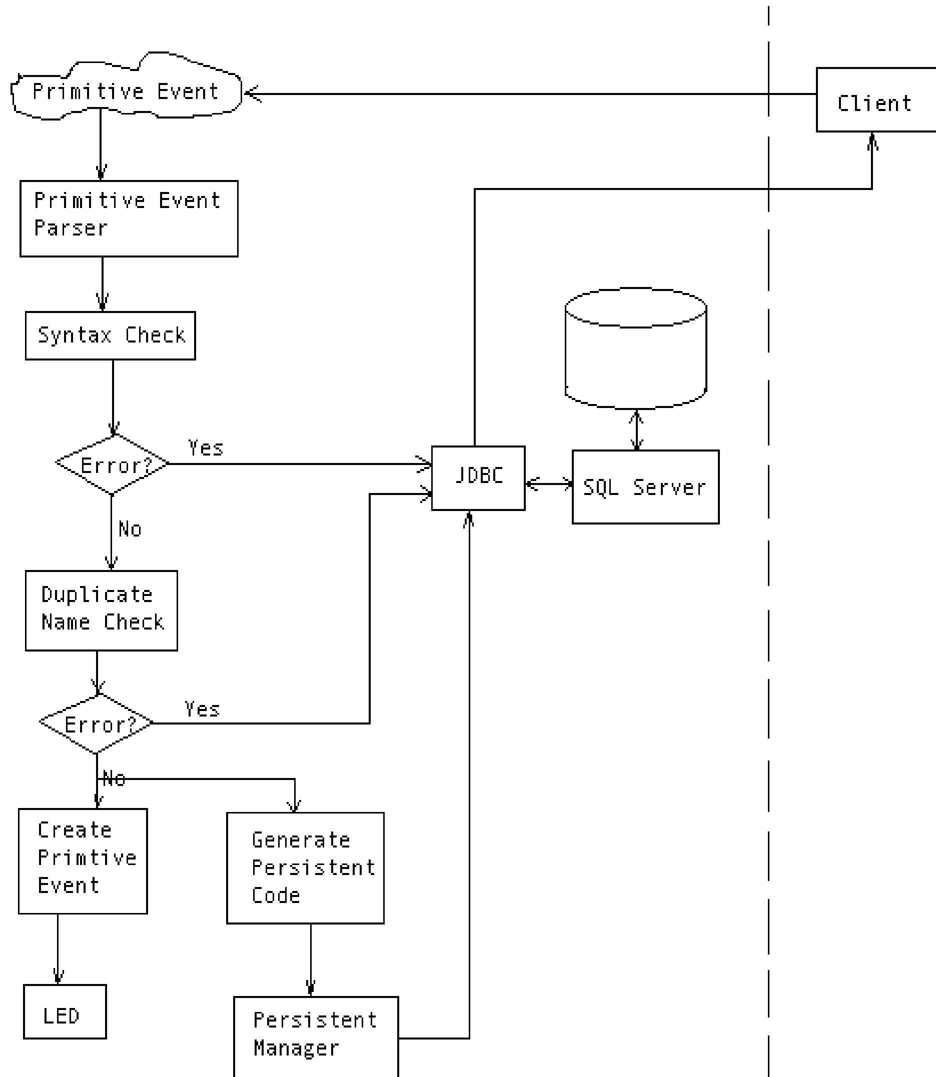


Figure 5.1 Flow Chart for Parsing and Generating Primitive Event

Now, we use an example to show how to apply these steps:

Example: **Create trigger t_addStk after insert on stock event addStk**

REFERENCING NEW_TABLE AS newtable

FOR EACH STATEMENT MODE DB2SQL

Insert into stock_copy select * from newtable

1. Syntax check: there are no syntax errors. Go to step 2.

2. Duplicate Name Check: trigger name does not duplicate. Go to step 3.

3. Create primitive event in LED.

Suppose we have already initialized ‘myAgent’ using LED:

```
ECAAgent myAgent = ECAAgent.initializeECAAgent();
```

We create primitive event ‘addStk’ using LED like this way:

```
EventHandle addStk =  myAgent.createPrimitiveEvent(“addStk”, “Led”,
    EventModifier.BEGIN, “void addStk()”, DetectionMode.SYNCHRONOUS);
```

4. Generate persistent code.

```
Insert into SysEcaTrigger values(‘zsong’, ‘t_addStk’, ‘t_addStk_proc’, current
        timestamp, ‘addStk’)
Insert into SysPrimitiveEvent values(‘zsong0’, ‘addStk’, ‘stock’, ‘insert’, ‘after’,
        current timestamp, 0)
```

5. Create inserted and deleted table.

Table ‘inserted’ is used to store the inserted tuples of a table. That means when you insert a tuple into a table, the inserted tuple will also be inserted into table ‘inserted’. In addition, the ‘inserted’ table also records the number of insertion.

For example, if we have table ‘stock’ as Table 5.1,

Table 5.1 stock

Symbol	Co_name	Price	Time

Then, we create table ‘stock_inserted’ and table ‘stock_deleted’ as Table 5.2.

Table 5.2 stock_inserted or stock_deleted

Symbol	Co_name	Price	Time	vNo

We see that the only difference of table 'stock' and table 'stock_inserted'(or table 'stock_deleted') is that table 'stock_inserted' has an additional attribute 'vNo'. The attribute 'vNo' is used to record the insertion number of table 'stock'. That means whenever the client inserts tuples into table 'stock', the 'vNo' will be increased by one with each insertion. This just records the unique event (here, for event 'addStk') occurrence value. The value of this attribute will be used for composing parameters for the parameter context specified (will be used for composite event).

6. Create trigger for primitive event. We need to do some works whenever after the tuples are inserted into table 'stock'. So we put this work into the trigger part. The trigger will be fired after the client inserts tuples into table 'stock'. The works we need to do include the following steps:

- User defined trigger action.

  Insert into stock_copy select * from newtable;

- Get event occurrence number:

  First, increase 'vNo' in table 'SysPrimitiveEvent'.

  Update SysPrimitiveEvent set vNo = vNo+1 where eventname = 'addStk';

  Then, put the value of 'vNo' into table 'Version'.

  Delete from Version;

  Insert into Version select vNo from SysPrimitiveEvent where eventname = 'addStk';

- Insert inserted tuple into table 'stock_inserted'.

Insert into stock_inserted select * from newtable, Version;

(note: 'newtable' stores the new inserted tuples in DB2)

- Insert tuples into table 'SysContext' (for composite event).

Insert into SysContext select * from EventContext, Verion where

EventContext.eventname = 'addStk';

- Send notification to Event Notifier (for composite event detection).

Update notify set eventname ='addStk';

In all, the "create trigger" command looks like this:

```
Create trigger t_addStk after insert on stock
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
BEGIN ATOMIC
        Insert into stock_copy  select * from newtable;
        Update SysPrimitiveEvent set vNo = vNo+1 where eventname = 'addStk';
        Delete from Version;
        Insert into Version select vNo from SysPrimitiveEvent where eventname =
                'addStk';
        Insert into SysContext select * from EventContext, Verion where
                EventContext.eventname = 'addStk';
        Insert into stock_inserted select * from newtable, Version;
         Update notify set eventname = 'addStk';
END;
```

(Note: in DB2 version 5.0, multiple sql statements are not supported in create trigger command, but in DB2 6.0, multiple sql statements are supported)
(Note: by now, we use DB2 version 5.0, we write multiple triggers for a certain event to implement the multiple sql statements in one trigger)

5.3 Creating Triggers on Existing Event

In DB2, we can create multiple-triggers for the same database operation on the same table, this is different from Sybase. In Sybase, user can only create one trigger for the same database operation on the same table, if you create the second trigger on the same table for the same database operation, the second one will replace the first one.

So, to implement 'create triggers on existing event' is much simple in DB2 than in Sybase. Next we give the syntax of creating triggers on existing event.

5.3.1 Syntax of creating triggers on existing event

In the earlier part of this chapter, we define the syntax of creating primitive event as follows:

create trigger *trigger_name* after/before insert/delete/update on *table_name* event *event_name*

REFERENCING NEW_TABLE AS newtable

FOR EACH STATEMENT MODE DB2SQL

SQL_statements

After we define a primitive event, we know which table this primitive event defined on and we know what kind of operation this primitive event defined on this table. So, when we define another trigger on this primitive event, we need not to define the table name and operation in the create trigger command, the follows is the syntax:

create trigger ***trigger_name*** **event** ***event_name***

REFERENCING NEW_TABLE AS newtable

FOR EACH STATEMENT MODE DB2SQL

SQL_statements

The primitive event definition is:
event *event_name [coupling_mode] [parameter_context] [priority]*.
Where
parameter_context :=
ECENT|CHRONICLE|CONTINUOUS|CUMULATIVE.
coupling_mode := IMMEDIATE|DEFERED|DETACHED.
priority := positive integer.
The default coupling mode is IMMEDIATE and the default parameter context is RECENT.

Figure 5.2 Repeat Primitive Event Syntax

When the user creates triggers on existing event, our program will call the 'Repeat Primitive Event Parser' to parse the command. Next we'll discuss how this parser works.

5.3.2 Implementation of Triggers on Existing Event

Because DB2 supports multiple-triggers in the same event, it is simple for us to deal with the repeated primitive event in our program. We use 'Repeated Primitive Event Parser' to parse it, the following steps are what we need to do:

1. Syntax checking:
2. Duplicate object name checking:
3. Code Generation:
4. Persistent code generation:

Now, we use an example to show how this works. Suppose we have already defined primitive event ‘addStk’, now define another trigger based on this event:

```
create trigger t1_addStk event addStk
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
   insert into PF values('Jin Kim', 'IBM', 1000, 200, current date)
```

We’ll work through the above steps:

1. Syntax checking: since there are no syntax errors, we’ll go to step 2.
2. Duplicate object name checking: here, trigger name ‘t1_addStk’ is not a duplicate name and event name ‘addStk’ already defined, so there are no errors, go to step 3.
3. Code Generation: When first time we define primitive event ‘addStk’, we already generate some codes to extend the active capability for the RDBMS. So, we need not to do that again, we just need to create a trigger, put the action into the trigger. The trigger command is:

```
create trigger t1_addStk after insert on stock
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
   insert into PF values('Jin Kim', 'IBM', 1000, 200, current date)
```

   Here, we just change the *event addStk* to the event definition (*after insert on stock*). This is acceptable for DB2.

4. Persistent code generation: we need to insert tuple into table ‘SysEcaTrigger’:

Insert into SysEcaTrigger values(‘zsong’, ‘t1_addStk’, ‘t1_addStk_proc’, current timestamp, ‘addStk’)

Right now, we have finished the work for the trigger defined on an exiting primitive event.

5.4 Dropping a Trigger on a Primitive Event

In DB2, we have ‘drop trigger’ command. In our program, we extend the ‘create trigger’ command. So if we want to drop a trigger, we need to do the reverse steps according to its ‘create trigger’ command. First we’ll discuss the syntax of drop trigger command.

5.4.1 Syntax of drop trigger command

In DB2, the syntax of drop trigger command as the follows:

```
drop trigger trigger_name
```

Figure 5.3 Drop Trigger Syntax

In our program, we still use the same syntax, this is transparent to the user.

When the user requests a drop trigger command, our program first check this trigger defined on a primitive event or a composite event. If this trigger is defined on a primitive event, ‘Drop trigger on primitive” will be used to parse it. If this trigger is defined on a composite event, ‘Drop trigger on composite” will be used to parse it. If this trigger did not defined on our primitive event or composite event, then this trigger just a traditional RDBMS trigger, we simply send this drop trigger command to SQL server to drop it.

Next we’ll discuss the steps to drop a trigger on primitive event.

5.4.2 Implementation of drop trigger on primitive event

When user define primitive events, our ECA Agent performs some actions. In order to drop a trigger, our ECA Agent needs to perform the reverse actions. The following is the steps we performed for creating a primitive event:

1. Generate persistent code.
2. Create trigger in DB2.
3. Create primitive event using LED.

We'll do the following steps:

1. Delete tuple from table 'SysEcaTrigger'.
2. Drop trigger in DB2.
3. Check if there is another trigger defined on this primitive event. If there are no other triggers defined on this primitive event, delete this primitive event tuple from table 'SysPrimitiveEvent'. If there are triggers also defined on this primitive event, we need not to delete this primitive event from table 'SysPrimitiveEvent'.
4. Drop primitive event from LED. First, we need to check if there are composite events defined based on this primitive event. If there are, we can not drop the primitive event. If there is no composite event defined based on this primitive event, we need to drop it from LED. To drop the primitive event from LED, the only thing we need to do is to delete the Java file that we created for this primitive event.

CHAPTER 6
IMPLEMENTATION OF COMPOSITE EVENTS

Composite events are not supported by RDBMS. In this research, we extend the trigger definition so composite events are supported in ECA Agent. In this chapter, we'll describe the details of implementation of composite events. In Section 6.1, we describe the syntax of composite event. In Section 6.2, Composite Event Parser is introduced. Event Notifier, ECA Action, and Parameter Context are described in Section 6.3, 6.4, and 6.5 respectively.

6.1 Syntax of composite event

We extend the trigger definition for composite event as we did for primitive event. For example:

```
create trigger t_addDel event addDel = addStk ^ delStk  RECENT
            BEGIN ATOMIC
                insert into temp values('Mark', 4) ;
            END
```

We add the keyword 'event' in the trigger command, and follow the keyword 'event' is the composite event syntax.

We use 'Snoop'- the event specification language to specify composite events in the trigger command.

Figure 6.1 shows the syntax of a composite event definition (Figure 6-1).

The default coupling_mode is 'IMMEDIATE' and the default prameter_contxt is 'RECENT'.

```
create trigger trigger_name
event event_name [= Snoop_Event_exp] [coupling_mode][parameter_context][priority]
BEGIN ATOMIC SQL_statements; END
        Coupling_mode := RECENT|CHRONICLE|CONTINUOUS|CUMULATIVE
        Parameter_context := IMMEDIATE|DEFERED|DETACHED
        Priority := positive integer
        Snoop_Event_exp ::= 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]: parameter, E1)
               | P* (E1, [time string], E1)
               | P(E1, [time string]: parameter, E1)
               | [time string]
               | E1 PLUS [time string]
               | (E1)
               | event_name
        event_name ::= name
```

Figure6.1: Composite Event Definition

6.2 Composite event parser

When the user defines a composite event, Composite Event Parser will parse it. There are four steps as shown in Figure 6.2:

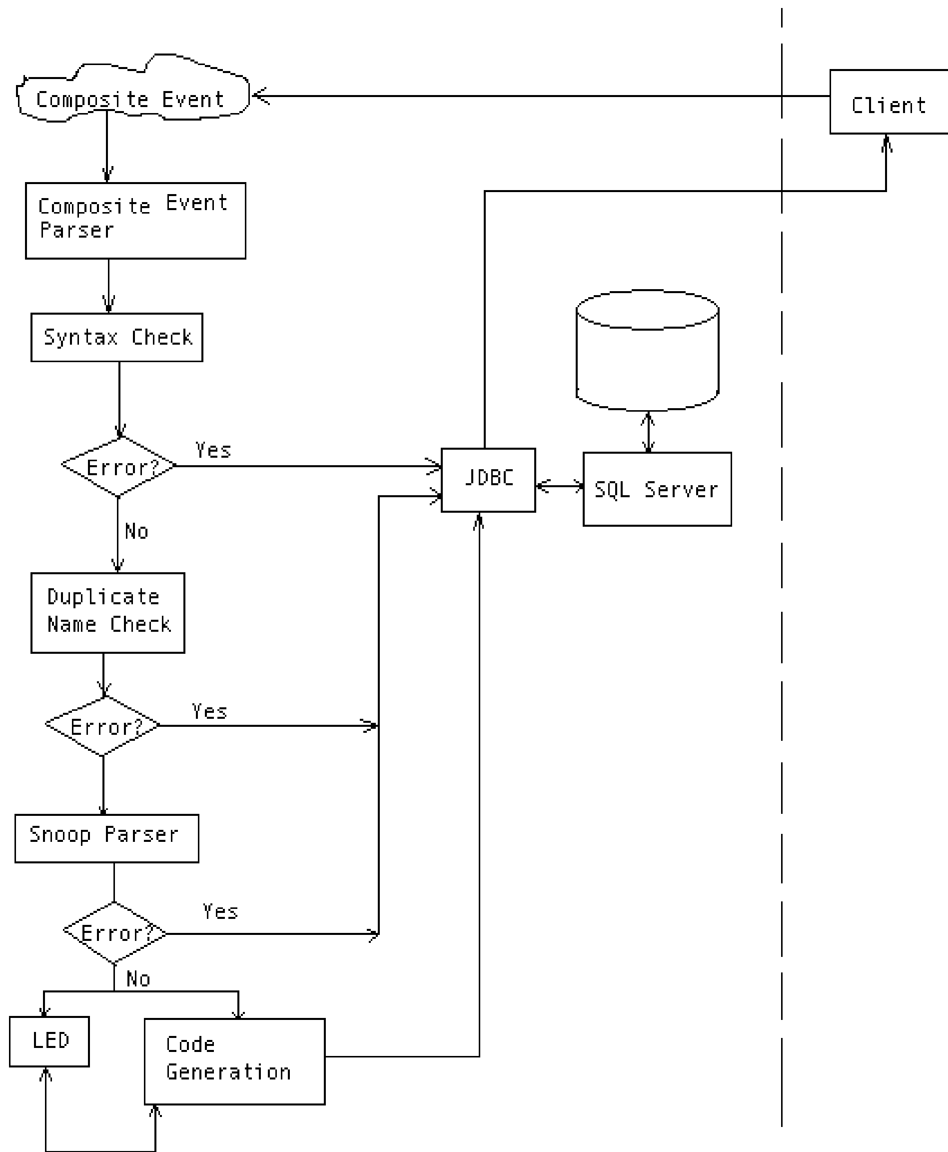


Figure6.2: Flow Chart of Composite Event Parser

1. Syntax check: check if there are syntax errors for this composite event. If there are syntax errors, return error message to the client.
2. Duplicate Name check: check if the trigger name is duplicate or not, because in RDBMS trigger name can not be duplicate. If the trigger name is duplicated, return error message to the client.

3. Send the composite event definition to Snoop parser. Snoop parser will parse the composite event syntax definition. If there are errors, return error message to the client. If there are no errors, snoop parser will create composite event in LED.
4. Code Generation.

Now, we use the following example to show how to apply these steps:

```
create trigger t_and  event addDel = addStk ^ delStk  RECENT
                    BEGIN ATOMIC
                         insert into temp values('Mark', 4) ;
                    END
```

1. Syntax check: there are no syntax errors. Go to step 2.
2. Duplicate Name Check: trigger name does not duplicate. Go to step 3.
3. Send the composite event definition (event addDel = addStk ^ delStk RECENT) to snoop parser. Snoop parser parses this composite event. No error is found. So create composite event and rule in LED.

The output of Snoop parser consists of two files, one is 'eventlist.txt' and the other is 'compositeevent.txt'.

File 'eventlist.txt' contains the events that used to define the composite event. We keep the event list to check whether these events have been defined or not. If there are events which are not defined, errors must be send to the client.

File 'compositeevent.txt' contains the API for creating the composite event.

Suppose we have already initialized 'myAgent' using LED:

```
ECAAgent myAgent = ECAAgent.initializeECAAgent();
```

We create composite event 'addDel' using LED like this:

```
EventHandle addDel = myAgent.createCompositeEvent(EventType.AND,
              "event addDel", addStk, delStk)
```

Create rule in LED:

```
myAgent.createRule("rule addDel", addDel, "Led.true", "Led.addDel", 1,
        CouplingMode.DEFAULT, Context.RECENT)
```

4. Persistent Code generation.

```
Insert into SysCompositEvent values('EcaAgent', 'zsong0', 'addDel', 'addStk ^ delStk',
        'RECENT', 'IMMEDIATE', 1)
Insert into SysEcaTrigger values('zsong0', 't_and', 't_and_proc', current timestamp,
        'addDel')
```

5. Create ECA_Action in LED. When composite event occurs, the following two tasks have to be done:

    - Trigger action that the user defined in this composite event should be executed.
    - We should keep the parameter context for this composite event. That means we'll keep the tuples (inserted or deleted or both of them) that made this composite event occurred and the composite event's parameter context.

The following is the ECA Action for the example composite event written by SQL statements. We implement these SQL statement use JDBC.

```
Delete from stock_inserted_tmp;
Insert into stock_inserted_tmp
   Select * from stock_inserted, SysContext
      where SysContext.context = 'RECENT' and
             SysContext.eventname = eventname(leftEventName and
        rightEventName) and
             Stock_inserted.vNo = SysContext.vNo;
Insert into temp values('Mark',4) ;
```

In this example, table ‘stock_inserted_tmp’ is a table generated from table ‘stock’ and table ‘SysContex’. Recall when we create primitive event, we have table ‘stock’ defined like Table 6.1.

Table 6.1 stock

Symbol	Co_name	Price	Time

We create table ‘stock_inserted_tmp’ and table ‘stock_deleted_tmp’ as Table 6.2.

Table 6.2 stock_inserted_tmp

Symbol	Co_name	Price	Time	VNo	EventName	Context	VNo1

When the composite event occurs, tuples will be inserted into table ‘stock_inserted_tmp’ and ‘stock_deleted_tmp’ according to the specific composite event.

6.3 Create Events and Rules

In this project, we use Java Led to detect composite events, so we need to register primitive events and composite events use the API of Java Led. For example, if user defined a primitive event ‘addStk’, then we’ll create primitive event use the following API:

```
EventHandle addStk = myAgent.createPrimitiveEvent("addStk","Led",EventModifier.BEGIN, "void addStk", DetectionMode.SYNCHRONOUS);
```

Because users define events dynamically, we must register events dynamically. In order to register events dynamically, we'll create a Java file called "userName+eventName.java", and then compile it dynamically. In our program, we'll call method "call_addStk()" to register this primitive event. Figure 6.3 shows the Java file, Figure 6.4 shows the code for compiling the Java file dynamically, and Figure 6.5 shows the code of how to register the primitive event.

```
import Sentinel.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

public class zsong0addStk{
   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);
         }
}
```

Figure 6.3 File "zsong0addStk.java"

```
// using 'javac' to compile the generated java file
String cmd = "javac " + className + ".java";
java.lang.Runtime rt = Runtime.getRuntime();

try  {
     Process pro = rt.exec(cmd);     // execute the command
     int a = pro.waitFor();          // wait until the current process terminate,
                                     // so that the command completed
 } catch(Exception e)  { }
```

Figure 6.4 Code for dynamic compile Java file

```
// to do "zsong0addStk.call_addStk();", use the following code
CallDynamicMethod.ExecuteMethod(classname,"call_" + eventname, null,
null);
```

Figure 6.5 Code for register events

In figure 6.5, we use "CallDynamicMethod.ExecuteMethod()" to do "zsong0addStk.call_addStk();" because in our program, we have a class "CallDynamicMethod", and we have a method called "ExecuteMethod" in this class. This method is used to execute the dynamic generated method. We put this class in appendix d.

To register composite events and rules we use the same method. Here we give an example.

```
create trigger t_and  event addDel = addStk ^ delStk  RECENT
        BEGIN ATOMIC
            insert into temp values('Mark', 4) ;
        END
```

We'll create file "zsong0addDel.java", see appendix d file 'zsong0addDel.java'.

And then we compile this file and register the composite event and rule just like we did for the primitive event.

6.4 Event Notification and Detection

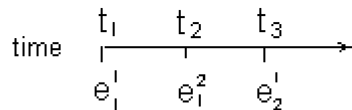
A composite event is composed of two or more primitive events using one or more of the snoop operators. Every composite event has an initiator event that initiates

the detection, and a terminator event that completes the detection of the event. The composite event is detected only when the terminator event is detected.

If we have a composite event using the AND operator:

event andEvent = AND(e_1, e_2)

Consider the event occurrences shown on the timeline below:



The AND event is detected when e_2 occurs.

In our case, e_1 and e_2 are database operations such as 'insert', 'delete' or 'update'. When the database operations occur (primitive events occur), we know this is be done by SQL server, how can we know these operations occur? – We are not the Servers. If we do not know the primitive events occur, how can we detect the occurrence of composite event? Solving these problems is our way to detect the composite event.

As we know, we use Java LED to detect the composite event. In order to detect the composite event, first we need to detect primitive events, as the above example, when e_2 occurs, the AND event is detected. How can we detect primitive events is our next topic.

6.4.1 Primitive Event Detection

In Java LED, we use the API to define a primitive event:

```
EventHandle addStk =  myAgent.createPrimitiveEvent("addStk", "Led",
      EventModifier.BEGIN, "void addStk()", DetectionMode.SYNCHRONOUS);
```

When the primitive event is defined as the above, an event handle corresponding to that event is returned. The event handle is used to signal the method invocation to the event detector. In order to signal the invocation of a method (a primitive event occurrence), the user can call an API inside a method that is defined as a primitive event.

```
Void addStk() {
        EventHandle[] myEvent = ECAAgent.getEventHandles (“addStk”);
        ECAAgent.raiseBeginEvent (myEvent, this);
}
```

First, the event handles corresponding to the primitive event are obtained using the name of the primitive event. Second, the event handles and the instance which invokes the method (this) are passed through the “raiseBeginEvent’ API.

By now, the primitive event “addStk” is detected.

In our case, the primitive events are database operations, for example we define the primitive event “addStk” inside the create trigger command:

```
Create trigger t_addStk after insert on stock  event addStk
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
        Insert into stock_copy  select * from newtable
```

After we insert a tuple into table ‘stock’, the primitive event ‘addStk’ will occur, but the operation ‘insert’ will be done in the SQL server side, and the SQL statements inside the trigger also will be done in the server side, that implies the primitive event ‘addStk’ will occur in the server side. We do not want these, if the primitive events occur in the server side, how can these primitive events trigger the composite event in our application?

What we want is after the ‘insert’ operation occurs, the SQL server should notify our application, let our application to raise the primitive event. Only in this way, the composite event can be triggered.

How to implement this issue is our next discussion.

6.4.2 Primitive Event Notification

After the ‘insert’ operation occurs, the SQL statements inside the trigger will be executed by the SQL server. The primitive event ‘addStk’ occurs in the server side, the server needs to notify the application that this primitive event occurred. The solution is that we call method ‘void addStk()’ inside the trigger to raise the primitive event, but this will raise the primitive event in the server side, not in the application.

In Sybase, it has a build-in function, “sybase-SendMesg(port number, IP address, method)”. Using this build-in function, we can raise primitive event in the specified IP address.

But in DB2, we do not have this kind of build-in function. We solve this problem using the following steps:

1. We create a table named notify. It has only one attribute, called “event name”. And this table has only one tuple at any time.
2. When a primitive event occurs, we put the occurred primitive event name into table ‘notify’. We do this by put one SQL statement into the trigger command like the follows:

Update table notify set eventname=’addStk’.

After we insert a tuple into table stock, this trigger will be fired, and this SQL statement will be executed automatically.

In our program, we use multi-thread to deal with each query requested by clients. Before the results are sent back to client, we check the table *notify*. If a primitive name exists in the table, our application will raise this primitive event by calling the method "void addStk()". Thus, this primitive event is notified to our application and is detected by LED.

6.4.3 Composite Event Detection

We use Java LED to detect composite events. When we create a composite event, we'll also create a rule. This rule contains "Event-Condition-Action". Once the event occurs, it will check the condition. If the condition is true, the Action will be executed. For example, we will create the following composite event:

Event addDel = addStk ^ delStk;

We'll also create the following rule:

myAgent.createRule("rule addDel", addDel, "Led.true", "Led.addDel", 1, CouplingMode.DEFAULT, Context.RECENT)

When the composite event 'addDel' is detected, and the condition is satisfied, the Action "Led.addDel" will be executed automatically.

Next, we'll discuss how to implement this in our case.

First, we have a Java file called "Led.java", like the follows:

```
import Sentinel.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

public class Led {
    //PrimitiveEventMethod
    public void PrimEvent(String eventname) {
     EventHandle[] addStk = ECAAgent.getEventHandles(eventname);
     ECAAgent.insert(addStk,"eventname",eventname);
     ECAAgent.raiseBeginEvent(addStk,this);
    }

   //ECA_Condition
   public static boolean True(ListOfParameterLists parameterLists) {
      System.out.println("***** From Condition ***** ");
      return true;
   }

}
```

Figure 6.6 Java file “Led.java”

In Figure 6.6, we have a method “public static boolean True()”. This method defines the condition part of ECA. Here, we suppose the condition is always true, which means if the event occurs, the action will be executed.

We also have a method “public void PrimEvent(String eventname)”. This method is used to raise the specified primitive event. The event name will be passed as the parameter. After a primitive event occurred and our application got the notification from the SQL server, the application will call this method to raise this primitive event like the follows:

Led.PrimEvent(eventname);

When both of the primitive event ‘addStk’ and ‘delStk’ occurred, the composite event ‘addDel’ is detected, and the Action should be executed.

Next we'll discuss the ECA Action part.

6.5 ECA Action

In active database, we use ECA rules to implement the active capability. In our case, events are database operations, conditions are always true, and actions are some SQL statements related to the database.

When events are detected, actions should be executed automatically. Because we define events in the extended trigger part, so the action should include the SQL statements that are defined inside the trigger. We also implement the parameter context in the action part. So, the action includes two parts:

1. SQL statements user defined inside the trigger.
2. SQL statements used to implement the parameter context.

Events are created by users. We don't know what kind of trigger action will be defined and we don't know the event defined on which table, so the action part will be dynamically created according to the definition of events.

To implement the ECA Action part, we do like this way:

When the user defines a composite event,

1. Get the triggered SQL statements.
2. According to the definition of the composite event, get the table names defined on this composite event, then insert tuples into table 'tablename.inserted_tmp' to get parameter context.

Because we use Java LED to detect composite event, and in Java LED, only class function written by Java can be called as the Action part, we create a Java file to contain the action part as a function in the Java file. We call the Java file

'EventNameUserName.java' to distinguish the same event that has been defined by the different users.

In our program, when we define a composite event, we'll also define a rule. Through the rule definition, we know when the event is detected and which action function we should call.

We'll give an example to show how we implement this.

1. Define a composite event 'addDel':

```
create trigger t_and  event addDel = addStk ^ delStk  RECENT
        BEGIN ATOMIC
            insert into temp values('Mark', 4) ;
        END
```

2. Create composite event and rule in LED.

```
EventHandle addDel = myAgent.createCompositeEvent(EventType.AND,
        "event addDel", addStk, delStk)

myAgent.createRule("rule addDel", addDel, "Led.true",
        "addDelzsong0.addDel", 1, CouplingMode.DEFAULT,
        Context.RECENT)
```

3. From the rule, we know when this composite event is detected, LED will execute "zsong0addDel.call_addDel()" as the action.

   We'll create file 'zsong0addDel.java'. In this file, there is a method called 'call_addDel()'. This Java file is appended in appendix d.

Because this file is created when the 'composite parser' parses the composite event, we need to compile this Java file in the run time. This is the same as we did for creating events. The code for how to compile Java file dynamically is shown in Figure 6.4.

Now, the action part is ready. When the composite event is detected, this action part will be executed automatically.

6.6 Parameter Context

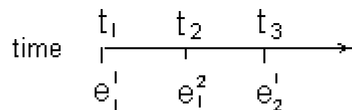
As mentioned before, composite events can be detected in more than one parameter context. In Java LED, the parameter contexts we supported are *recent, chronicle, continuous, and cumulative*. And we also need to be noted that for primitive events all parameter contexts are identical.

Here we still use the example we used in the earlier part of this chapter to discuss the different detection in different parameter contexts.

If we have a composite event using the AND operator:

event andEvent = AND(e_1, e_2)

Consider the event occurrences shown on the timeline below:



The AND event is detected when e_2 occurs. But we are not clear whether e_2^1 should be paired with e_1^1 or e_1^2. Parameter contexts are useful for distinguishing this ambiguity. In *recent* context, e_1^2 and e_2^1 are detected for 'andEvent'. In chronicle context, e_1^1 and e_2^1 are detected for 'andEvent'. In continuous context, two events e_1^1 e_2^1 and e_1^2 e_2^1 are detected at the same time for 'andEvent'. In cumulative context, a single event e_1^1 e_1^2 e_2^1 is detected for 'andEvent'.

In our case, primitive events are database operations. In order to keep the context for a primitive event in a certain composite event, we create a table named 'tablename.inserted_tmp' to contain the context for the primitive event. Next we'll discuss how to implement parameter context for primitive event.

1. When the user defines a primitive event, we create table 'tablename.inserted_tmp' and 'tablename.deleted_tmp'.
2. When the user defines a composite event, we put all primitive events (which consists of this particular composite event) and parameter context into table 'eventContext' to keep the primitive events and its context for the composite event.
3. When primitive events occur, join two tables 'tablename_inserted' and 'eventContext', and then the results are inserted into table 'SysContext'.
4. When composite events occurs, join two tables 'tablename_inserted' and 'SysContext', and then the results are inserted into table 'tablename.inserted_tmp' to get parameter context.

Now, we'll use an example to show how this works:

```
create trigger t_and  event addDel = addStk ^ delStk  RECENT
        BEGIN ATOMIC
          insert into temp values('Mark', 4) ;
        END
```

1. When the user defines primitive events 'addStk' and 'delStk', we'll create table 'stock_inserted_tmp' and table 'stock_deleted_tmp' as Table 6.3.

Table 6.3 stock_inserted_tmp or stock_deleted_tmp

Symbol	Co_name	Price	Time	VNo	EventName	Context	VNo1

2. When the user defines composite event ‘addDel’, we’ll input tuples into table ‘eventContext’ as Table 6.4.

Table 6.4 EventContext

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

3. When primitive event ‘addStk’ occurs, for example, we insert a tuple into table ‘stock’, we join table ‘stock_inserted’ and ‘eventContext’ to get tuples insert to table ‘SysContext’, as follows:

- **Insert into table stock values(‘sun’, ‘sun’, 123, current timestamp);**
- **Delete from Syscontext where eventname=’addStk’ and context=’RECENT’;**
- **Insert into Syscontext select eventname, context, vNo from eventContext, stock_inserted;**

Now, table ‘SysContext has the format of Table 6.5.

Table 6.5 SysContext

EventName	Context	Vno
‘addStk’	‘RECENT’	1

When we delete a tuple from table ‘stock’, primitive event ‘delStk’ will occur, we’ll do the follows just like we did after primitive event ‘addStk’ occured,

- **Delete from table stock;**
- **Delete from Syscontext where eventname=’delStk’ and context=’RECENT’;**
- **Insert into Syscontext select eventname, context, vNo from eventContext, stock_deleted;**

Now, table ‘SysContext has the format of Table 6.6.

Table 6.6 SysContext

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

4. Right now, the composite event ‘addDel’ is detected, we’ll get parameter context after we did the follows(Figure 6.7):

```
        //context processing for 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_tmp.vNo = SysContext.vNo

        //context processing for event ‘addStk’
Delete stock_inserted_tmp;

Insert into stock_inserted_tmp;

Select * from stock_inserted, SysContext

    Where SysContext.context=’RECENT’   and

           SysContext.eventname = ‘delStk’   and

            Stock_inserted_tmp.vNo = SysContext.vNo
```

Figure 6.7 parameter context processing

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

In this thesis, we presented the details of design, architecture and implementation of ECA Agent, to recall there are some sub-modules in our ECA Agent:

- Language Filter
- ECA Parser
    - Primitive Event Parser
    - Composite Event Parser
    - Repeat Primitive Event Parser
    - Repeat Composite Event Parser
    - Drop trigger Parser
        - Drop trigger defined on Primitive Event
        - Drop trigger defined on Composite Event
- Persistent Manager
- Java Led
- JDBC

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

- ECA rules are supported in our Agent.
- Both primitive events and composite events can be detected.

- Active behaves (events, rules, actions) are persistent in DBMS.
- Drop trigger and events as desired.
- Multiple parameter contexts are supported in our Agent.

Our ECA Agent is a mediator in the SQL server and clients, and we use JDBC to connect with SQL server and SQL requirements. So this design is a generalized method for any RDBMS to extend its active capability.

7.2 Contributions

The contributions of this thesis are:

- Designed a mediated approach that significantly extends the active capability of any RDBMS. This mediated approach has some advantages: it does not change the SQL Server/Client; it's transparency to the clients; it has extensibility, etc.
- Implemented the ECA Agent according to the design.
- Full-fledged active capability is supported.
- We use JDBC to connect the SQL Serer and the Clients. It's a generalized method. By using JDBC, you can connect any SQL Server and clients. You need not to worry about the specified functions of a specified RDBMS.

7.3 Future work

In our implementation, we use DB2 as the test database and we extended the active capability of DB2 Universal Database. Next we'll implement these by using Oracle as the test database. The only difference here is the SQL statements' syntax between DB2 and Oracle.

In this thesis, we use Java Led to detect composite events. Right now, Java Led can only detects events in a single application. In the future, it can be extended to detect events in a distributed system. At that time, our Agent will also support to detect events in a distributed system.

APPENDIX A
USER MANUAL

In this project, we implement two programs, one is “Java ECA Agent Server”, the other one is “Java ECA Agent Client”.

Java ECA Agent Server

This program is our ECA Agent, it should be run on the server machine, just like the Oracle SQL server or Sybase SQL server is running on the server machine. This program can be run on any machine, but first you must run this program before you run the client program.

Start the ECA Agent

If you want to start the ECA Agent, you should execute the following command in the “dos” environment:

java Java_ECA_Agent_Server

Then, the ECA Agent starts, you will see a little window like follows:

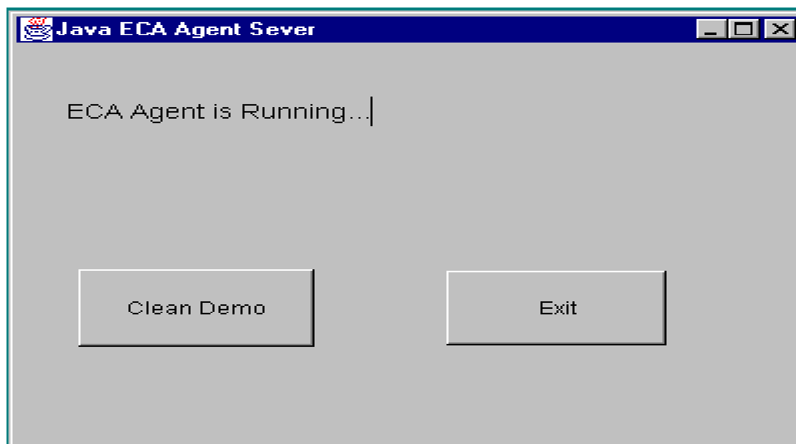


Figure A.1 ECA Agent Server Interface

In this window, it said: "ECA Agent is Running…" that means ECA Agent is running and right now you can start you client interface.

There is a button "Exit", when you click this button, the ECA Agent will shut down.

Also, we have a "dos" window like the follows:

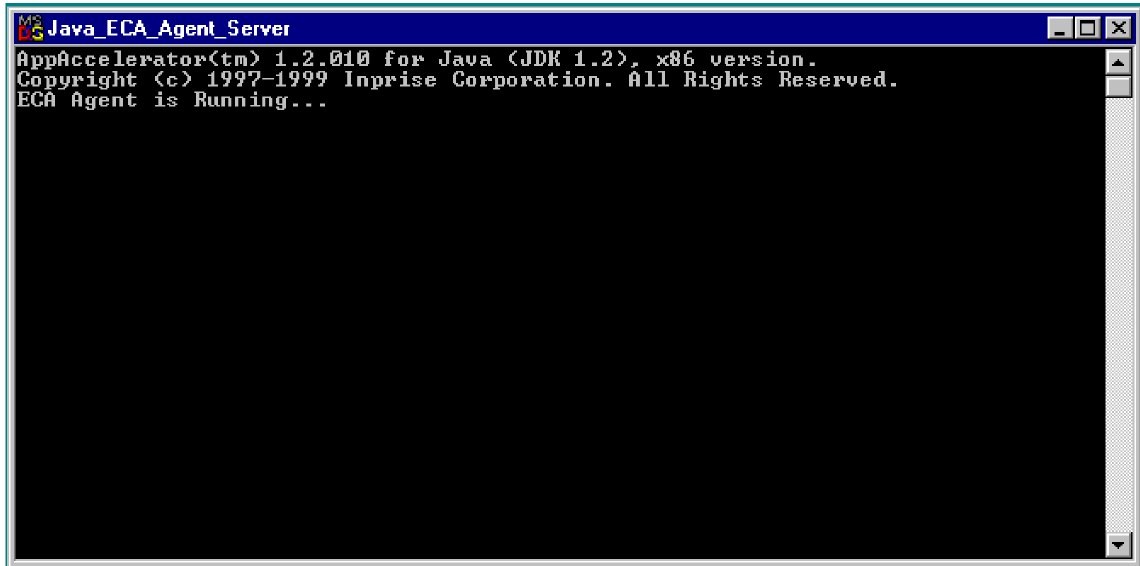


Figure A.2 ECA Agent Server DOS Environment

Use this window, you can get some run time information.

Java ECA Agent Client

This program is the interface for the client, clients can use this interface input the SQL requirements. It looks just like the original DBMS interface. We should notice that this program must be start after the ECA Agent server program started.

Start the ECA Agent Client interface

To start this program, just input the following command in the "dos" environment:

java Java_ECA_Agent_Client

You'll notice that the following window displays:

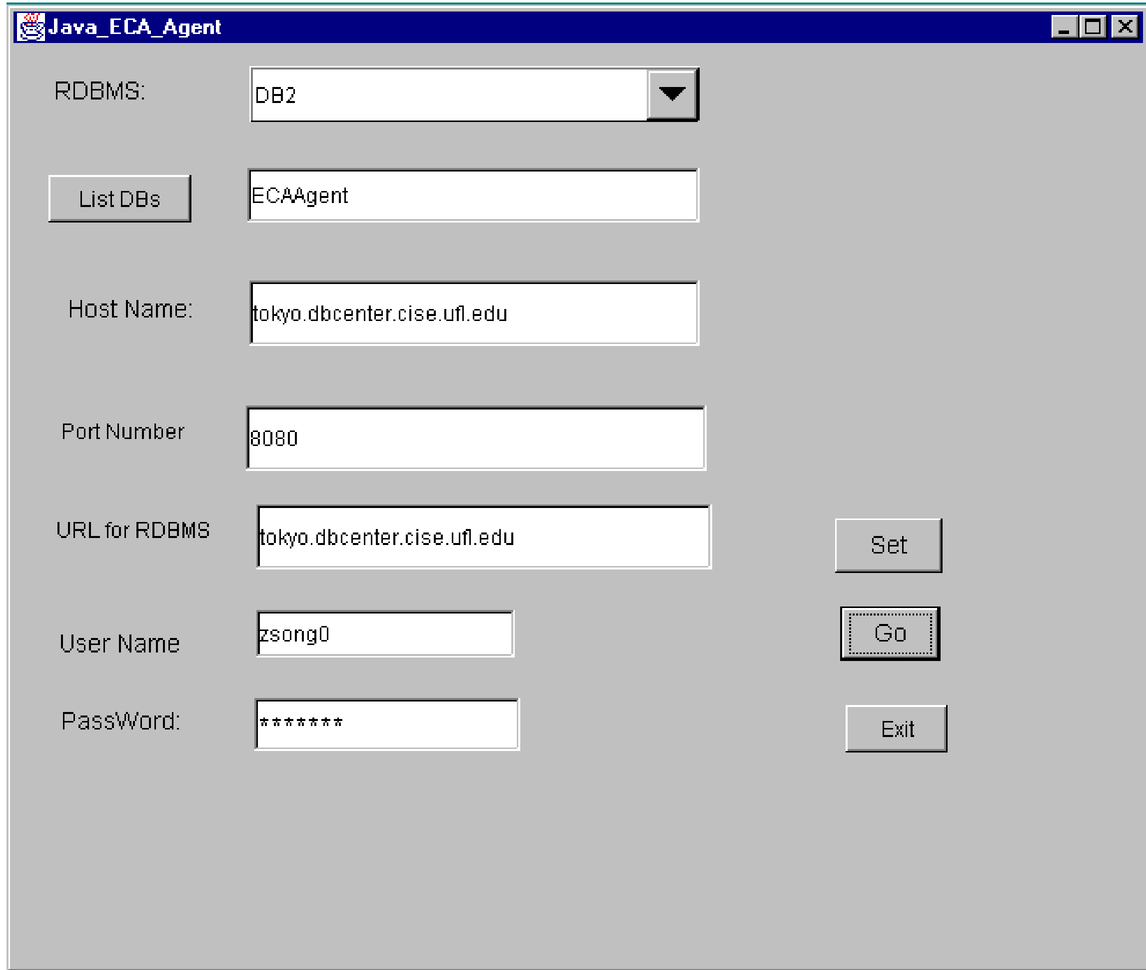


Figure A.4 ECA Client Interface

In this window, there is a lot of information. We'll talk it one by one.

- RDBMS: from the como box you can select "DB2", "Oracle", "Sybase" or "Informix". Because our Agent is a generalized Agent, it will be worked for all kinds of RDBMS. Right now, it works for "DB2", and it will work for "Oracle" in a short time.

- List DBs: you should input the database name that you'll use in the specified RDBMS into this text area. For example, we'll use database named "ECAAgent" in DB2. So we put "ECAAgent" in this text area.
- Host Name: this should be the machine name where you run the ECA Agent Server. For example, our ECA Agent is running on the machine "tokyo", so we input the name "tokyo.cise.ufl.edu" into this text area.
- PortNumber:
- URL for RDBMS: this is the URL for RDBMS, for example, our DB2 SQL server in running on machin "tokyo", and the URL for "tokyo" is "tokyo.cise.ufl.edu", so we put "tokyo.cise.ufl.edu" into this text area. Notice you can also input the IP address into here, the IP address of "tokyo" is '128.227.146.79', so we can input "128.227.146.79" in this text area.
- User Name: this is the account id that you use in the specified RDBMS. For example, we use the account id "zsong0" for DB2.
- Password: this is the password for the account used in the specified RDBMS.
- Button "Set": chick this button, the system will keep all the information you input into this window, it will be used latter.
- Button "Go": click this button, another window will show on, if you select "Oracle", the "Oracle" interface will show on, if you select "DB2", the "DB2" interface will show on.
- Button "Exit": click this button, the client program will terminate.

DB2 interface

If we select “DB2”, then the “DB2” interface will show on, it looks like the DB2 command center:

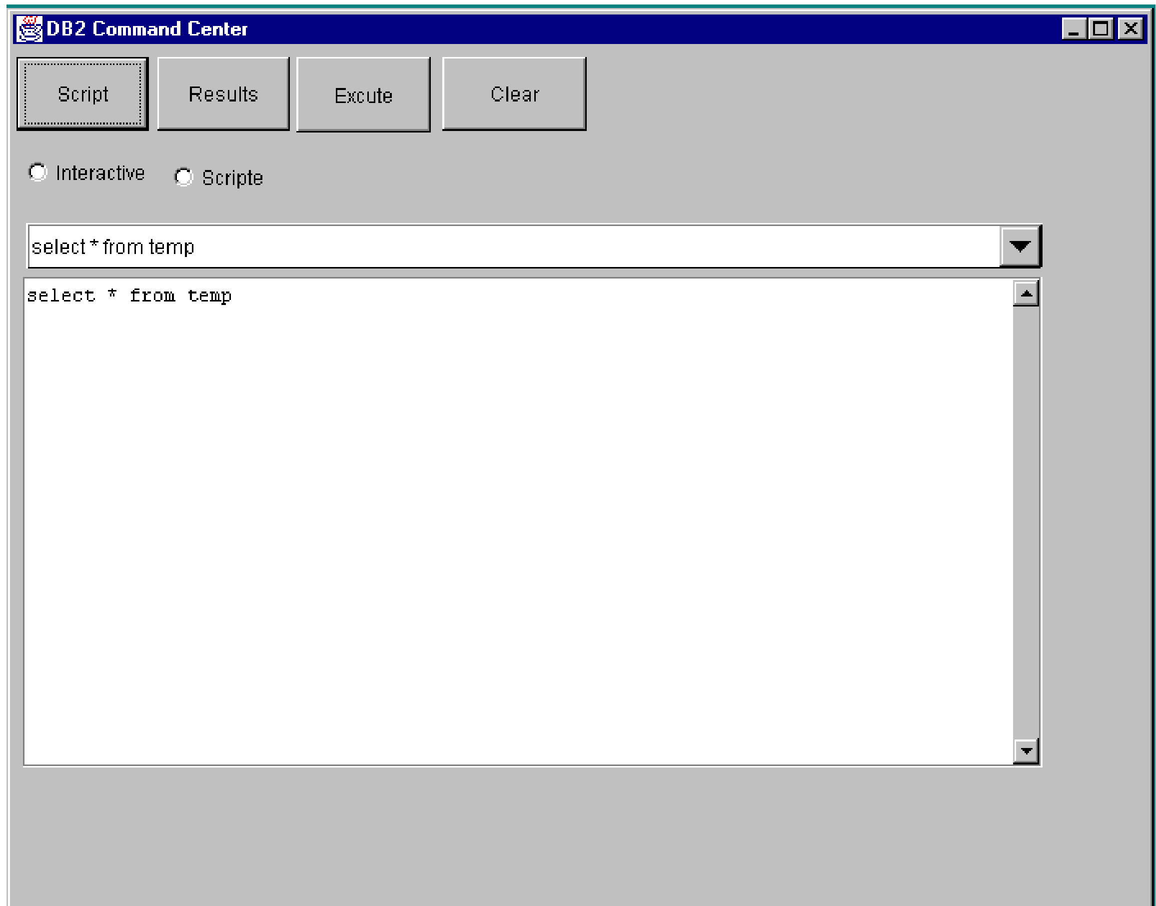


Figure A.5 DB2 Interface

This interface looks like the “DB2 command center”, there are some buttons:

- Script: when you click this button, the script you select from the “script list” will show in the text area window. You can also input SQL statements into text area window directly.
- Results: click this button, you can get the latest results.

- Execute: click this button, the SQL statements inside the text area window will be executed. Before you click this button, make sure the SQL statements inside the text area window are what you want.
- Clear: click this button, the text area window will be cleaned.

APPENDIX B
DEMO

Here, we'll use examples to show how this ECA Agent works. Before the demo, we need to do some work.

Preparing the Demo

First, we need to create some tables, see appendix c file "createSystables.txt".

Second, if necessary, clean demo use DB2 command center, the clean demo commands are in the file "cleanDemo.txt" in appendix c.

Demo

There are some steps for the demo:

1. Start ECA Agent Server by input the following command in the "dos" environment:

   java Java_ECA_Agent_Server

   We'll see a "dos window"-- "Java_ECA_Agent_Server" shows on. That means the ECA Agent is ready.

2. Start the ECA Agent Client by input the following command in the "dos" environment:

   java Java_ECA_Agent_Client

   Now, the ECA Agent Client interface shows on, we input some information in his window, like the follows:

Figure B.1 ECA Agent Client Interface

3. Click button “Set” to let the system keeping the information.
4. Click button “Go”. The DB2 interface displays.
5. Now, we create some primitive events and composite events, see appendix c file “createEvents.txt”.
6. After we create primitive events and composite events, now we can test it. See appendix c file “test.txt”.
7. Check the results, see appendix c file “results.txt”.

APPENDIX C
FILES USED IN THE DEMO

The following files are used for the demo.

File 1: createSystables.txt

```
drop table SysEcaTrigger;
drop table SysPrimitiveEvent;
drop table SysCompositeEvent;
drop table Version;
drop table ActiveRDBMS_ECA
drop table sysContext;
drop table eventContext;

create table SysEcaTrigger   (
  dbName          char(30),
  userName        char(30),
  triggerName     char(30),
  triggerProc     char(60),
  timeStamp       timestamp,
  eventName       char(30)
 )

create table SysPrimitiveEvent   (
   dbName    char(30),
   userName  char(30),
   eventName char(30),
   tableName char(30),
   operation char(30),
   beafoperation char(10),
   timeStamp timestamp,
   vNo       integer
  )

create table SysCompositEvent   (
   dbName    char(30),
   userName  char(30),
   eventName char(30),
   eventDescribe char(100),
   timeStamp       timestamp,
```

```
   coupling  char(10),
   context   char(12),
   priority  integer
  )

create table Version  (
   vNo integer
  )

create table ActiveRDBMS_ECA  (
   EcaVariables varchar(255),
   TriggerFunc  char(50)
 )

create table sysContext  (
   eventname char(20),
   context char(12),
   vNo integer
  )

create table eventContext   (
   eventname char(20),
   context char(12)
  )

grant all on SysEcaTrigger to public;
grant all on SysPrimitiveEvent to public;
grant all on SysCompositEvent to public;
grant all on Version to public;
grant all on ActiveRDBMS_ECA to public;
grant all on sysContext to public;
```

File 2: cleanDemo.txt

```
drop table stock_inserted;
drop table stock_deleted;
drop table stock_inserted_tmp;
drop table stock_deleted_tmp;
drop trigger t_addStk;
drop trigger t_addStk1;
drop trigger t_addStk2;
drop trigger t_addStk01;
drop trigger t_addStk02;
drop trigger t_addStk03;
```

```
drop trigger t_addStk04;
drop trigger t_delStk;
drop trigger t_delStk1;
drop trigger t_delStk2;
drop trigger t_delStk01;
drop trigger t_delStk02;
drop trigger t_delStk03;
drop trigger t_delStk04;

drop table stock;
drop table stock_copy;
create table stock(symbol char(10), Co_name char(20), price integer, time
timestamp);
create table stock_copy(symbol char(10), Co_name char(20), price integer, time
timestamp);
insert into stock values('ibm','ibm',320,current timestamp);

delete from SysEcaTrigger;
delete from SysPrimitiveEvent;
delete from SysCompositEvent;
delete from temp;
delete from SysContext;
delete from eventContext;

------------------------------

drop table PF_inserted;
drop table PF_deleted;
drop table PF_inserted_tmp;
drop table PF_deleted_tmp;
drop trigger t_buyStk;
drop trigger t_buyStk1;
drop trigger t_buyStk2;
drop trigger t_buyStk01;
drop trigger t_buyStk02;
drop trigger t_buyStk03;
drop trigger t_buyStk04;
drop trigger t_selStk;
drop trigger t_selStk1;
drop trigger t_selStk2;
drop trigger t_selStk01;
drop trigger t_selStk02;
drop trigger t_selStk03;
drop trigger t_selStk04;

drop table PF_copy;
```

```
drop table PF;
create table PF(name char(20), symbol char(6), amount integer, price integer, time date);
create table PF_copy(name char(20), symbol char(6), amount integer, price integer, time date);
insert into PF values('Sharma Cha', 'JNJ', 1000, 200, current date);
```

<u>File 3: createEvents.txt</u>

- Create primitive events

```
create trigger t_addStk after insert on stock event addStk
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
    insert into stock_copy select * from newtable


create trigger t1_addStk event addStk
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
    insert into PF values('Jin Kim', 'IBM', 1000, 200, current date)


create trigger t_delStk after delete on stock event delStk
REFERENCING OLD_TABLE AS oldtable
FOR EACH STATEMENT MODE DB2SQL
    insert into stock_copy select * from oldtable


create trigger t_buyStk after insert on PF event buyStk
REFERENCING NEW_TABLE AS newtable
FOR EACH STATEMENT MODE DB2SQL
    insert into PF_copy select * from newtable


create trigger t_selStk after delete on PF event selStk
REFERENCING OLD_TABLE AS oldtable
FOR EACH STATEMENT MODE DB2SQL
      insert into PF_copy select * from oldtable
```

- Create composite events

```
create trigger t_and event addDel = delStk ^ addStk RECENT
BEGIN ATOMIC
        insert into temp values('Mark',4) ;
END

create trigger t_Or event buySel = buyStk | selStk RECENT
BEGIN ATOMIC
    insert into temp values('Jerry',3) ;
END


create trigger t_addDelBuy event addDelBuy = addDel ; buyStk CUMULATIVE
BEGIN ATOMIC
    insert into temp values('addDelBuy',4 ) ;
END


create trigger t_delSel event delSel = delStk ^ selStk CUMULATIVE
BEGIN ATOMIC
    insert into temp values('delSel', 4 ) ;
END


create trigger t_com event comEvent = addDelBuy ; delSel CUMULATIVE
BEGIN ATOMIC
     insert into temp values('comEvent',4) ;
END


create trigger t_com1 event comEvent1 = addDel ; buyStk ; ( delStk ^ selStk )
CUMULATIVE
BEGIN ATOMIC
    select * from PF.deleted ;
END

create trigger t_comOr event comOr = buyStk | selStk | addStk | delStk RECENT
BEGIN ATOMIC
    insert into temp values('Tom',4 ) ;
END
```

File 4: test.txt

1. insert into stock values(‘sun’,’sun’,123, current timestamp)
2. insert into stock values(‘soft’,’soft’,230, current timestamp)
3. insert into stock values(‘citrix’,’citrix’,100, current timestamp)

4. delete from stock where price=123
5. delete from stock where price=230

6. insert into stock values(‘Oracle’, ‘Oracle’, 320, current timestamp)

7. delete from stock where price=100

8. insert into PF values('Jone', 'Jone', 1500, 100, current date)
9. insert into PF values('Don', 'Don', 1800, 150, current date)

10. delete from PF where price=100

11. insert into stock values(‘Informix’, ‘Informix’, 310, current timestamp)

12. delete from stock where price=320

13. delete from PF where price=150

14. insert into PF values('RockWood', 'Don', 1300, 200, current date)

File 5: results.txt

Results from LED

DB2 Jdbc driver started...

Sequence number = 1
RAISING EVENT Ledzsong0beginvoidaddStk() 1
Notifying CLASS level event addStk...
LEDThread returning from get
notifying event addStk
Executing rules on event 'addStk' ...
Rules on event addStk
Number of rules = 0

applicationThread returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 2
RAISING EVENT Ledzsong0beginvoidaddStk() 2
Notifying CLASS level event addStk...
LEDThread returning from get
notifying event addStk
Executing rules on event 'addStk' ...
Rules on event addStk
Number of rules = 0

Thread-13 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 3
RAISING EVENT Ledzsong0beginvoidaddStk() 3
Notifying CLASS level event addStk...
LEDThread returning from get
notifying event addStk
Executing rules on event 'addStk' ...
Rules on event addStk
Number of rules = 0

Thread-14 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 4
RAISING EVENT Ledzsong0beginvoiddelStk() 4
Notifying CLASS level event delStk...
LEDThread returning from get
notifying event delStk
Executing rules on event 'delStk' ...
Rules on event delStk
Number of rules = 0

Event event addDel was triggered at the context of RECENT.

void addStk() 3
void delStk() 4 ***** From Condition *****
****From Composite Event Action of Rule****

Event event addDel was triggered at the context of CUMULATIVE.

void addStk() 1
void addStk() 2
void addStk() 3
void delStk() 4
Thread-15 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 5
RAISING EVENT Ledzsong0beginvoiddelStk() 5
Notifying CLASS level event delStk...
LEDThread returning from get
notifying event delStk
Executing rules on event 'delStk' ...
Rules on event delStk
Number of rules = 0

Event event addDel was triggered at the context of RECENT.

void addStk() 3
void delStk() 5 ***** From Condition *****
****From Composite Event Action of Rule****

Thread-16 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 6
RAISING EVENT Ledzsong0beginvoidaddStk() 6
Notifying CLASS level event addStk...
LEDThread returning from get
notifying event addStk
Executing rules on event 'addStk' ...

Rules on event addStk
Number of rules = 0

Event event addDel was triggered at the context of RECENT.

void delStk() 5
void addStk() 6 ***** From Condition *****
****From Composite Event Action of Rule****

Event event addDel was triggered at the context of CUMULATIVE.

void delStk() 5
void addStk() 6
Thread-17 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 7
RAISING EVENT Ledzsong0beginvoiddelStk() 7
Notifying CLASS level event delStk...
LEDThread returning from get
notifying event delStk
Executing rules on event 'delStk' ...
Rules on event delStk
Number of rules = 0

Event event addDel was triggered at the context of RECENT.

void addStk() 6
void delStk() 7 ***** From Condition *****
****From Composite Event Action of Rule****

Thread-18 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 8
RAISING EVENT Ledzsong0beginvoidbuyStk() 8
Notifying CLASS level event buyStk...
LEDThread returning from get
notifying event buyStk

Executing rules on event 'buyStk' ...
Rules on event buyStk
Number of rules = 0
DetectionMask at event event buySel: 1000

Event event buySel was triggered in context RECENT.

void buyStk() 8 ***** From Condition *****
****From Composite Event Action of Rule****

Event event addDelBuy was triggered at the context of CUMULATIVE.

void addStk() 1
void addStk() 2
void addStk() 3
void delStk() 4
void delStk() 5
void addStk() 6
void buyStk() 8 ***** From Condition *****
****From Composite Event Action of Rule****

Thread-19 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 9
RAISING EVENT Ledzsong0beginvoidbuyStk() 9
Notifying CLASS level event buyStk...
LEDThread returning from get
notifying event buyStk
Executing rules on event 'buyStk' ...
Rules on event buyStk
Number of rules = 0
DetectionMask at event event buySel: 1000

Event event buySel was triggered in context RECENT.

void buyStk() 9 ***** From Condition *****
****From Composite Event Action of Rule****

Thread-20 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 10
RAISING EVENT Ledzsong0beginvoidselStk() 10
Notifying CLASS level event selStk...
LEDThread returning from get
notifying event selStk
Executing rules on event 'selStk' ...
Rules on event selStk
Number of rules = 0
DetectionMask at event event buySel: 1000

Event event buySel was triggered in context RECENT.

void selStk() 10 ***** From Condition *****
****From Composite Event Action of Rule****

Event event delSel was triggered at the context of CUMULATIVE.

void delStk() 4
void delStk() 5
void delStk() 7
void selStk() 10 ***** From Condition *****
****From Composite Event Action of Rule****

Event event comEvent was triggered at the context of CUMULATIVE.

void addStk() 1
void addStk() 2
void addStk() 3
void delStk() 4
void delStk() 5
void addStk() 6
void buyStk() 8
void delStk() 4
void delStk() 5
void delStk() 7
void selStk() 10 ***** From Condition *****
****From Composite Event Action of Rule****

Thread-21 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 11
RAISING EVENT Ledzsong0beginvoidaddStk() 11
Notifying CLASS level event addStk...
LEDThread returning from get
notifying event addStk
Executing rules on event 'addStk' ...
Rules on event addStk
Number of rules = 0

Event event addDel was triggered at the context of RECENT.

void delStk() 7
void addStk() 11 ***** From Condition *****
****From Composite Event Action of Rule****

Event event addDel was triggered at the context of CUMULATIVE.

void delStk() 7
void addStk() 11
Thread-22 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 12
RAISING EVENT Ledzsong0beginvoiddelStk() 12
Notifying CLASS level event delStk...
LEDThread returning from get
notifying event delStk
Executing rules on event 'delStk' ...
Rules on event delStk
Number of rules = 0

Event event addDel was triggered at the context of RECENT.

void addStk() 11
void delStk() 12 ***** From Condition *****
****From Composite Event Action of Rule****

Thread-23 returning from put
finshiehd.

closing...

DB2 Jdbc driver started...

Sequence number = 13
RAISING EVENT Ledzsong0beginvoidselStk() 13
Notifying CLASS level event selStk...
LEDThread returning from get
notifying event selStk
Executing rules on event 'selStk' ...
Rules on event selStk
Number of rules = 0
DetectionMask at event event buySel: 1000

Event event buySel was triggered in context RECENT.

void selStk() 13 ***** From Condition *****
****From Composite Event Action of Rule****

Event event delSel was triggered at the context of CUMULATIVE.

void delStk() 12
void selStk() 13 ***** From Condition *****
****From Composite Event Action of Rule****

Thread-24 returning from put
finshiehd.
closing...

DB2 Jdbc driver started...

Sequence number = 14
RAISING EVENT Ledzsong0beginvoidbuyStk() 14
Notifying CLASS level event buyStk...
LEDThread returning from get
notifying event buyStk
Executing rules on event 'buyStk' ...
Rules on event buyStk
Number of rules = 0
DetectionMask at event event buySel: 1000

Event event buySel was triggered in context RECENT.

void buyStk() 14 ***** From Condition *****
****From Composite Event Action of Rule****

Event event addDelBuy was triggered at the context of CUMULATIVE.

void delStk() 7
void addStk() 11
void buyStk() 14 ***** From Condition *****
****From Composite Event Action of Rule****

Thread-25 returning from put
finshiehd.
closing...

APPENDIX D
SOME JAVA CLASS FILES

File "zsong0addDel.java"

```
import Sentinel.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

public class zsong0addDel{
  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();
        addDel = myAgent.createCompositeEvent(EventType.AND,"event addDel"
,(EventHandle)delStk.delStk, (EventHandle)addStk.addStk);

        myAgent.createRule("Rule addDel", addDel, "Led.True","addDel.addDelzsong0",
1,CouplingMode.DEFAULT,Context.RECENT);
}

public static void addDelzsong0(ListOfParameterLists paramLists) {
    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 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");

            //action function
    String spc = " insert into temp values('Mark',4)";
    Jdbc storedProCom = new Jdbc(rdbms,url,username,password, spc);
    storedProCom.ExecuteSqlUpdate( " insert into table temp ");
    System.out.println ("****From Composite Event Action of Rule****");
   }}
```

File “CallDynamicMethod.java”

```
import java.io.*;

public class CallDynamicMethod
{
  public static void ExecuteMethod(String className, String methodName,
Class[] paramTypes, Object[] params)
   {
    // load the new compiled class
    String strClass = className;        // "InsertStock", etc
    ClassLoader cl = null;
    cl = ClassLoader.getSystemClassLoader ();
    Class cla = null;
    try   {
      cla = cl.loadClass(strClass);     // load the class
    }
    catch(ClassNotFoundException e4)
    {
      System.out.println(e4.toString());
    }
```

```
    // get the method from the class
    java.lang.reflect.Method meth = null;
    try  {
     // find the particular method from the loadedclass
      meth = cla.getMethod(methodName, paramTypes);
     Object obj = null;
      meth.invoke(obj, params);      // execute the particular method.
      System.out.println("finshiehd.");
    }
    catch(Exception e5)
    {
      System.out.println(e5.toString());
    }
  }  // end of 'ExecuteMethod()'

}  // end of class
```

LIST OF REFERENCES

[LIJ98] Lijuan, L. (1998), An Agent-Based Approach to Extending the Native Active Capability of Relational Database Systems, Master's thesis, University of Florida, Gainesville, 1998

[DON98] Don Chamberlin. (1998), A Complete Guide To DB2 Unviersal Database. IBM Almaden Research Center.

[JEN96] Jennifer Widom, and Stefano Ceri(1996), Active Database Systems Triggers and Rules for Advanced Database Processing.

[Nor98] Norman W. Paton (1998), Active Rules in Database Systems.

[JEN96+] Jennifer Widom (1996), The Starburst Active Database Rule System, IEEE Transactions on Knowlede and data engineering, Vol.8, No. 4: August 1996, pp. 583-595

[CHA94] S.Chakravarthy, V. Krishnaprasad, E. Anwar, and S.K.Kim, "Composite Events for Active Databases: Semantics, Contexts and Detection," in *Proceedings International Conference on Very Large Databases*, Santiago, Chile, 1994, pp. 606-617.

[CHA94a] S.Chakravarthy, E. Anwar, L. Maugis, and D. Mishra, "Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules," in *Information and Software Technology*, Vol. 36, pp. 559-568, 1994

[GJ91] N.Gehani and H.V. Jagadish, 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.

[CHA94b] S. Chakravarthy and D. Mishra, Snoop: An Expressive Event Specification Language for Active Databases, *Data and knowledge Engineering*, 13(3), Octorber 1994.

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

[LEE96] Lee, H. Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing. Master's thesis, University of Florida, Gainesville, 1996.

[JavaTutorial] The Java Tutorial.

[ART99] ART Taylor. JDBC Developer's Resource.

BIOGRAPHICAL SKETCH

Zecong Song was born on March 05, 1972 in Baoding, Hebei, China. She received her Bachelor of Science degree in Computer Science from Hebei University, Baoding, Hebei, China in July 1994.

In the fall of 1998, she started her graduate studies in the department of Computer and Information Sciences and Engineering at the University of Florida. She will receive her Master of Science degree in Computer Science in August 2000 from the University of Florida, Gainesville, Florida. Her research interests include active databases and e-commerce.