A MEDIATOR BASED APPROACH TO SUPPORT ECA RULES IN DB2

by

NELLAINAYAGAM CHETTIKULAM SUBRAMANIAM

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2002

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this challenging topic and providing me great guidance and support through the course of this research.

I would like to thank Dr. JungHwan Oh and Dr. Leonidas Fegaras for serving on my committee.

I am grateful to Raman Adaikkalavan, Sreekant Thirunagari, Nishanth Vontela, Ganesh Gopalakrishnan, and Arvind Mysore for their invaluable help and advice during the implementation of this work. I also would like to thank Pratyush Mishra for administering the research network. I would like to thank all my friends in the ITLAB for their support and encouragement.

This work was supported, in part, by the Office of Naval Research, the SPAWAR System Center-San Diego & by the Rome Laboratory (grant F30602-01-2-05430), and by NSF (grant IIS-0123730).

I would also like to thank my parents and my brother for their endless love and constant support throughout my academic career.

April 4, 2002

ABSTRACT

A MEDIATOR BASED APPROACH TO SUPPORT ECA RULES IN DB2

Publication No.__________

Nellainayagam Chettikulam Subramaniam, M.S.

The University of Texas at Arlington, 2002

Supervising Professor: Sharma Chakravarthy

Active database systems typically use ECA (even-condition-action) rules for specifying events of interest, conditions to be checked, and action to be performed. Though triggers provide active capability in commercial RDBMSs, they provide only a limited specification of events (e.g., insert/delete/modify) and do not allow combination of events. Triggers also do not support parameter contexts, coupling modes and priority.

In this thesis, we use a mediator-based approach to provide full ECA functionality to a commercial RDBMS without changing the underlying system. IBM DB2 6.1 is used as the test RDBMS. Triggers can be created on primitive events (insert/delete/modify) and composite events (combination of events) and support parameter contexts, coupling modes and priority. Though the approach used in this thesis is a generalized approach that can be used for other RDBMSs, the Mediator implementation is unique in certain respects as it is based on the underlying trigger support specific to DB2.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In a DBMS, a user has to explicitly execute queries to monitor changes on the state of the DBMS. Hence the traditional DBMS is referred to as a passive database system. For example, if information about a web site is stored in a database, it is the responsibility of the user or the user application to continuously monitor the database in order to execute specific actions based on changes to the web site. In order to continuously monitor and automatically initiate actions in response to changes on the database state, there is need to provide active capability to a passive DBMS.

Though triggers provide active capability in a DBMS, they are not at a level of abstraction that is required for real world applications. During the last decade significant amount of work has been done to incorporate active capability into DBMSs. A number of issues need to be considered in developing an active database system [1]:

1. An active database system must provide all the usual functionality of a conventional database system. Meanwhile, it is desirable that the performance of conventional database tasks is not degraded by the fact that the database system has become active.

2. 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.

3. 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.

4. An active database system must provide database design and debugging tools similar to those provided by conventional database systems, extended to incorporate active behavior.

Active database systems typically use ECA (even-condition-action) rules on events of interest. Once the rules are declared, the system continuously monitors occurrence of events, checks the condition specified and executes the action if the condition evaluates to true.

There are many commercial DBMSs that provide active capability. But their active capability is limited to supporting triggers. Commercial active database capabilities suffer from four main shortcomings [1]:

1. They lack standardization. The various products have a wide variance in both the syntax and execution behavior of triggers. This results in 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, such as application-specific events, event composition, 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 forth.

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.

A number of research prototypes of active database systems such as (HiPAC [2], Ariel [3], Sentinel [4], Starburst [5], Postgres [6], SAMOS [7] etc.) have been developed. Most of these prototypes are based on the integrated approach that requires access to the underlying source code of the DBMS. Integrated approach cannot be used for commercial DBMSs, unlike the research prototypes, as access to the underlying source code cannot be obtained. Hence, it is imperative that active capability be enhanced without changing the underlying system.

A mediator based ECA Agent was implemented by Lijuan Li[8] at the University of Florida in order to provide ECA functionality to a commercial DBMS (Sybase). This ECA Agent used the Gateway OpenServer present in Sybase to provide ECA functionality. Hence this implementation was not portable to other commercial DBMSs. The next step was a Generalized ECA Agent that was implemented by Zecong Song [9] at the University of Florida. In the Generalized ECA Agent, the user can specify the desired active behavior by defining events and rules. The events can be primitive or composite. A primitive event is a database operation (insert/delete/update). A composite event is a set of primitive events or composite events connected by event operators. The notions of events, event operators and rules have been discussed in detail in the event specification language Snoop [10, 11]. Though the ECA Agent is a generalized approach that can be ported to other commercial DBMSs., the implementation does not support all the composite events that have been defined by the event specification language Snoop. The Generalized ECA Agent supports only three composite events: And, Or, Seq. Further the Generalized ECA Agent does not support dropping of composite events and rules defined on them.

With this as the background, in this thesis we improve upon the Generalized ECA Agent to overcome the above deficiencies. This implementation uses IBM DB2 6.1 as the test database. Like the Generalized ECA Agent, this implementation also uses a Mediator that is external to the underlying DBMS. This provides transparency to the clients. Also the Mediator is portable and extensible. Triggers can be created on both primitive and composite events. The Mediator supports most of the events defined in Snoop including parameter contexts, coupling modes and priority. Both primitive and composite events can be dropped. The events and rules are persisted using the underlying DBMS. Multiple clients can use the same Mediator to create events and rules on multiple databases, namely, the multi-user/multi-database capability of the underlying DBMS is preserved.

This thesis is organized as follows. Chapter 2 discusses some of the related work. Chapter 3 discusses about DB2 triggers. Chapter 4 discusses the issues involved in the design of the Mediator. Implementation of primitive and composite events is discussed in chapters 5 and 6 respectively. Conclusions and future work are discussed in chapter 7.

CHAPTER 2

RELATED WORK

There has been a significant amount of work over the last decade in the field of Active Databases. Several research prototypes have been developed. This chapter discusses a few of them.

2.1 Starburst

The Starburst system [1] is an extensible relational DBMS developed at the IBM Almaden Research Center. Starburst provides an integrated active database rule processing facility called the *Starburst Rule System*. The primary goals in the development of the Starburst Rule System are:

1. Need for a rule definition language with a clearly defined and flexible execution semantics.

2. Rapid implementation of the rule system as a fully integrated component of the database system. This was achieved by exploiting the extensibility features of Starburst.

The Starburst Rule System is fully integrated into the database system with database query and transaction processing, including concurrency control, authorization, rollback recovery, and error handling. The rule definition language consists of five commands: create rule, alter rule, deactivate rule, activate rule, drop rule. Rules may be grouped into rule sets. The commands for manipulating rule sets are: create ruleset, alter ruleset, and drop ruleset. The implementation of the Starburst Rule System is based on the three extensibility features of the Starburst database system: attachments, table functions, and event queues.

Starburst Rule system is an integrated approach which requires access to the underlying source. Hence this approach cannot be used for commercial RDBMSs as they do not allow access to the underlying source. Moreover, this system only allows rules triggered by operations on a single table. It does not support the notion of monitoring a combination of events on different tables.

2.2 Sentinel

The Sentinel system [4] is a follow-on system to HIPAC [12]. This system uses an integrated approach to incorporate active capability in an Object Oriented Database System. The platform used for this system is the Open OODB Toolkit from Texas Instruments [ref]. The Sentinel system extends the passive Open OODB system by incorporating extensions to the Open OODB kernel. These extensions include:

1. Implementation of a Sentinel pre-processor and a Sentinel post-processor to convert the high-level user specification of ECA rules into appropriate code for event detection, parameter computation, and rule execution. The ECA rules are specified using the event specification language Snoop.

2. Implementation of local event detector for detecting events (primitive and composite). There is also provision for parameter computation in various contexts when composite events are detected.

3. Implementation of transaction manager for supporting nested transactions used for concurrent execution of rules.

4. Implementation of a rule debugger for visualizing the interaction among rules, among events and rules, and among rules and database objects.

Again this system, uses an integrated approach which cannot be used for commercial RDBMSs. Also this system was implemented for an Object Oriented Database System. Hence this approach is not feasible for a commercial RDBMS.

2.3 Sybase ECA Agent

The Sybase ECA Agent was implemented by Lijuan Li [8]. This system uses a mediator ECA Agent between the clients and the Sybase SQL server. The Gateway Open Server(GOS) provided by Sybase allows clients to connect to the ECA Agent. The ECA Agent provides ECA service to the clients. From the client's point of view, it is a virtual SQL server. If the client sends a simple SQL command, it is sent directly to the SQL server. If the client sends an ECA command it is sent to the other parts of the ECA Agent through the GOS. GOS also accepts SQL requirements from other parts of the ECA Agent and forwards it to the SQL server.

In this approach, events and rules are defined by extending the native trigger syntax of Sybase database system. Both primitive and composite events can be defined in this approach. The Sybase ECA Agent provides full transparency to the client and also provides persistence to user created events using the native DBMS capability. The Sybase ECA Agent was developed in C++. The Sybase ECA Agent used the C++ Local Event Detector (LED) to support composite events. The Sybase ECA Agent is dependent on the Gateway OpenServer to provide ECA functionality. Also since it uses the C++ LED there are portability constraints to be considered. Hence this approach is not a generalized approach that can be ported to other commercial RDBMSs such as DB2, Oracle.

2.4 Generalized ECA Agent

The Generalized ECA Agent was implemented by Zecong Song[9]. This system is a follow-on system to the Sybase ECA Agent. This system is also based on the mediated approach. The functionality of the Generalized ECA Agent is similar to the Sybase ECA Agent. The Generalized ECA Agent uses IBM DB2 Universal Database 5.1 as the test database system. This system has been developed in Java. This interfaces with the Java Local Event Detector (Java LED) to detect primitive and composite events. The system uses Java Database Connectivity (JDBC) to connect to the underlying database system. Since JDBC is independent of any specific RDBMS, the Generalized ECA Agent approach can be used for other database systems such as Oracle, Sybase.

The Generalized ECA Agent allowed users to create primitive and composite events by specifying event triggers. The Generalized ECA Agent supported only a limited set of composite events. Also some of the ECA features such as parameter contexts, coupling modes and priority were not properly implemented in this approach. Furthermore, dropping of events and rules were not properly implemented.

Summary:

This chapter discussed some of the approaches used in enhancing active capability of the underlying DBMS. Systems that use the integrated approach such as Starburst, Sentinel were discussed. Also systems that use the mediator-based approach such as Sybase ECA Agent, Generalized ECA Agent were also discussed.

CHAPTER 3

DB2 TRIGGERS

In this implementation, the events are specified by extending the native trigger syntax of DB2 6.1. This chapter gives a brief description of DB2 triggers [13]. A trigger executes a series of SQL statements whenever data is inserted, deleted, or updated in a specific table. The syntax of a CREATE TRIGGER statement in DB2 is shown in figure 3.1.

3.1 Trigger Syntax Description

Name:

The trigger-name is a two-part name that includes a schema name. The trigger name must be unique within its schema.

Triggering event:

The triggering event is the event that causes the trigger to be activated. A triggering event is either an insert or delete or update event on the table to which the trigger is attached.

Activation Time:

The trigger is activated either before or after its triggering event. There can be multiple triggers on a table (no limit). Multiple triggers for the same triggering event are executed in the order in which they are created.

Examples of activation times,

```
BEFORE INSERT ON employee
AFTER INSERT ON employee
```

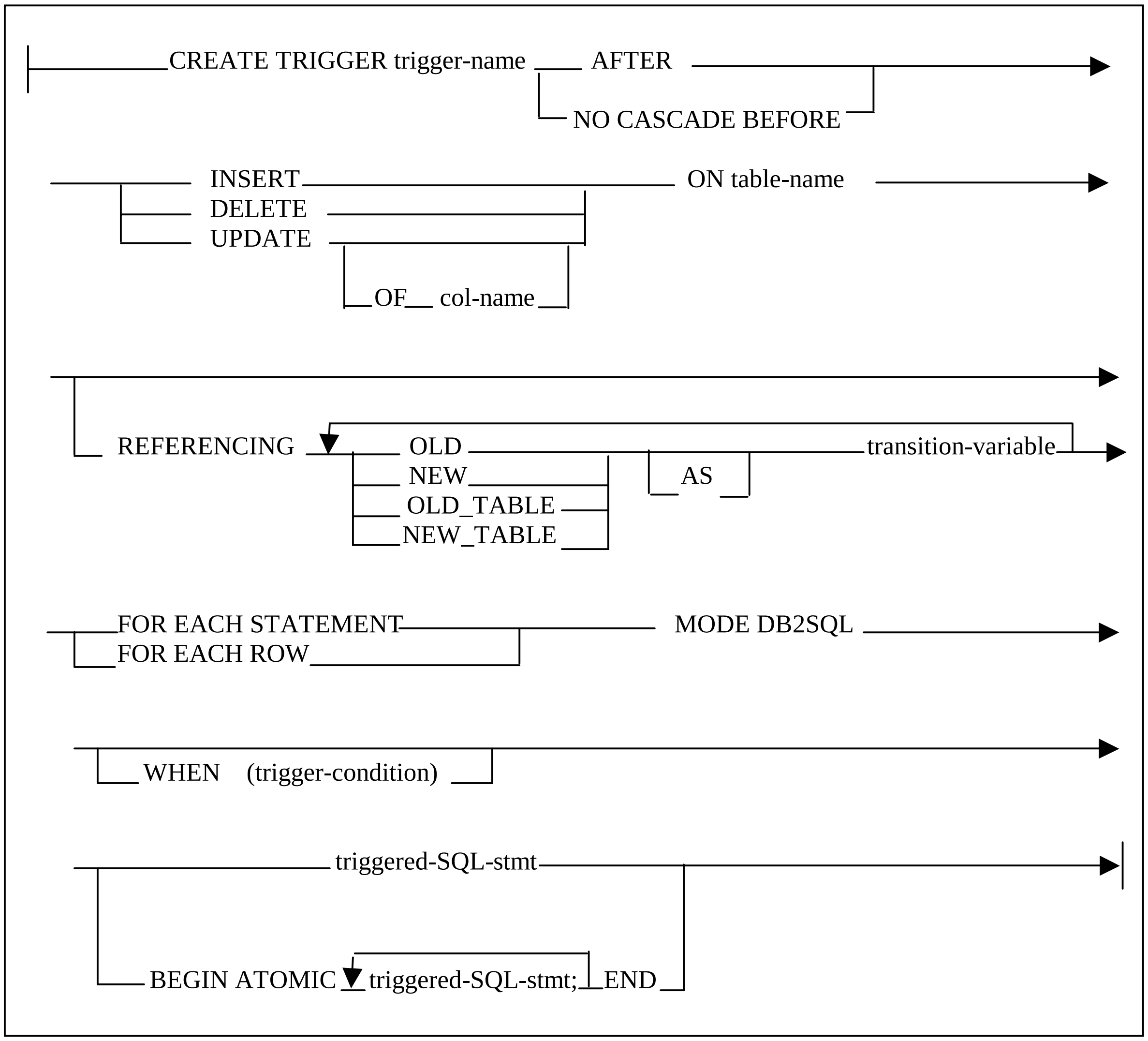


Figure 3.1. DB2 Trigger Syntax.

Granularity:

Based on their granularity, triggers can be classified into statement or row triggers. A statement trigger specifies that the trigger be applied only once for the SQL statement that caused the triggering event. A row trigger specifies that the trigger be applied once for each row of the table that is affected by the triggering SQL statement. The phrases FOR EACH STATEMENT, FOR EACH ROW are used to make this distinction.

Transition Variables:

Transition variables hold information about the specific database change that activated a trigger. This information is available to the action body of the trigger. There are four kinds of transition variables. They are:

1. Old row variable, which represents the value of the modified row before the triggering event.

2. New row variable, which represents the value of the modified row after the triggering event.

3. Old table variable, which represents a hypothetical read-only table containing all the modified rows as they appeared before the triggering event.

4. New table variable, which represents a hypothetical table containing all the modified rows as they appeared after the triggering event.

The REFERENCING clause is used to give names to the transition variables.

Examples of row transition variables,

```
REFERENCING OLD AS  oldrow
REFERENCING NEW AS newrow
```

Examples of table transition variables,

```
REFERENCING  OLD_TABLE  AS  oldtable
REFERENCING  NEW_TABLE  AS  newtable
```

Certain triggers can access only some of the four possible transition variables. The transition variables for each of the trigger types are summarized in table 3.1.

Table 3.1. Transition Variable Access Summary

Triggering Event and Activation Time	**Row Trigger Can Use**	**Statement Trigger Can Use**
BEFORE INSERT	New row	(Invalid)
BEFORE UPDATE	Old row, new row	(Invalid)
BEFORE DELETE	Old row	(Invalid)
AFTER INSERT	New row New table	New table
AFTER UPDATE	Old row, new row Old table, new table	Old table, new table
AFTER DELETE	Old row Old table	Old table

Trigger Condition:

The trigger condition evaluates to true or false. The trigger condition starts with WHEN clause. The associated action is performed only if the specified condition evaluates as true. If the trigger condition is not specified, the associated triggered action is always executed.

Examples of trigger conditions are,

WHEN(newrow.temperature > oldrow.temperature)

WHEN(SELECT count(*) FROM newtable) > 50

Trigger Body:

The trigger body contains the action to be performed when a trigger is activated. A triggered-action is composed of one or several SQL statements. If there is more than one SQL statement in the triggered-action for a given trigger, they must be enclosed within BEGIN ATOMIC and END keywords.

In DB2 6.1, it is not possible create a stored procedure from within the body of the trigger. A stored procedure is written in any of the programming languages supported by

DB2 such as C, C++, COBOL, FORTRAN, REXX, or JAVA. Since the trigger body allows only SQL statements, it is not possible to create a stored procedure from within the body of the trigger. But in other RDBMSs such as Sybase and Oracle it is possible to create a stored procedure from within the body of the trigger.

Summary:

This chapter discussed issues concerning creation of triggers in DB2. The triggering event is an insert/delete/update operation on a table. The granularity of triggers were also discussed. The transition tables and their accessibility within the body of triggers were also discussed.

CHAPTER 4

DESIGN ISSUES

The Generalized ECA Agent discussed in chapter 2 suffers from several shortcomings. Though it showed the feasibility of using an external ECA Agent (Mediator in our case) to support ECA functionality, some of the features such as composite event functionality, multi-user/multi-database, dropping of events and rules were not properly implemented. The Mediator design proposed in this thesis overcomes these shortcomings.

The general view of the system is shown in figure 4.1. The Mediator receives the input from the client and processes it before sending it to the DB2 server. It allows multiple clients to connect to multiple databases.

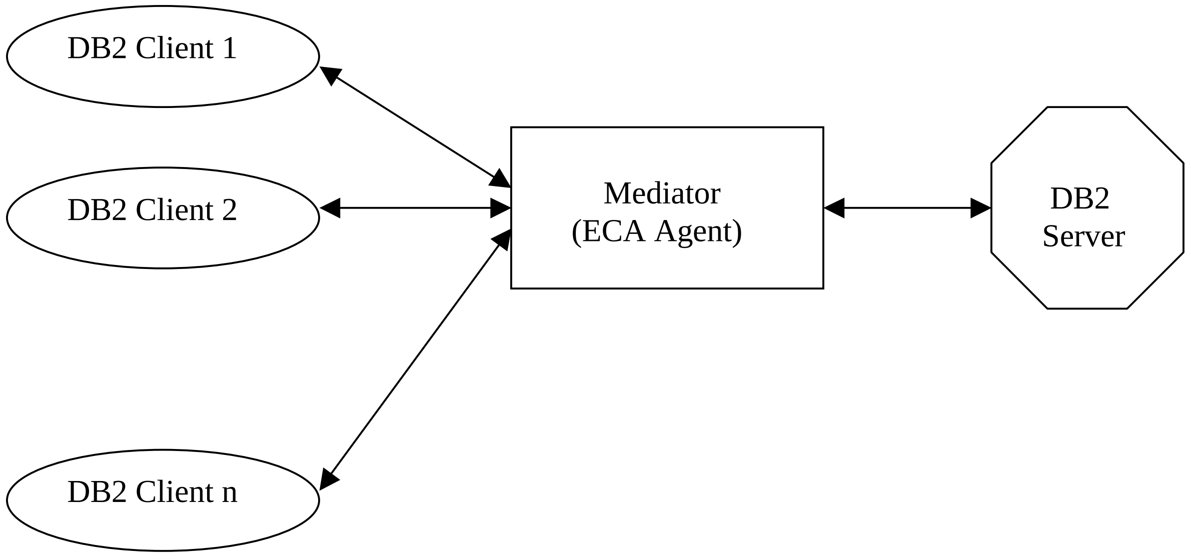


Figure 4.1. General View of the System.

4.1 Event Triggers

Most commercial DBMSs support only triggers on primitive events(operations) insert/delete/update on a table. They do not support the notion of triggers on composite events. In this implementation, each event is associated with a unique name so that the names can be used for composition of events. A user can specify a named event by specifying triggers. The following sections discuss the syntax for creating named events using triggers.

4.1.1 Primitive Event Triggers

A user can specify a primitive event by creating a primitive event trigger. In order to provide transparency to the user, the primitive event trigger syntax should be similar to the native trigger syntax of the DB2. The primitive event trigger syntax is shown in figure 4.2. This syntax is similar to the native trigger syntax of DB2 except for the addition of the keyword event followed by the event_name. The rest of the trigger syntax description is the same as explained for a DB2 trigger.

Example 4.1: Primitive Event Trigger

create trigger t_addwT after insert on WeatherTexas ***event addwTexas***

```
        referencing new as newrow
        for each row mode DB2Sql
        begin atomic
        insert into temp values('addWeather');
        insert into info values('insert operation');
        end
```

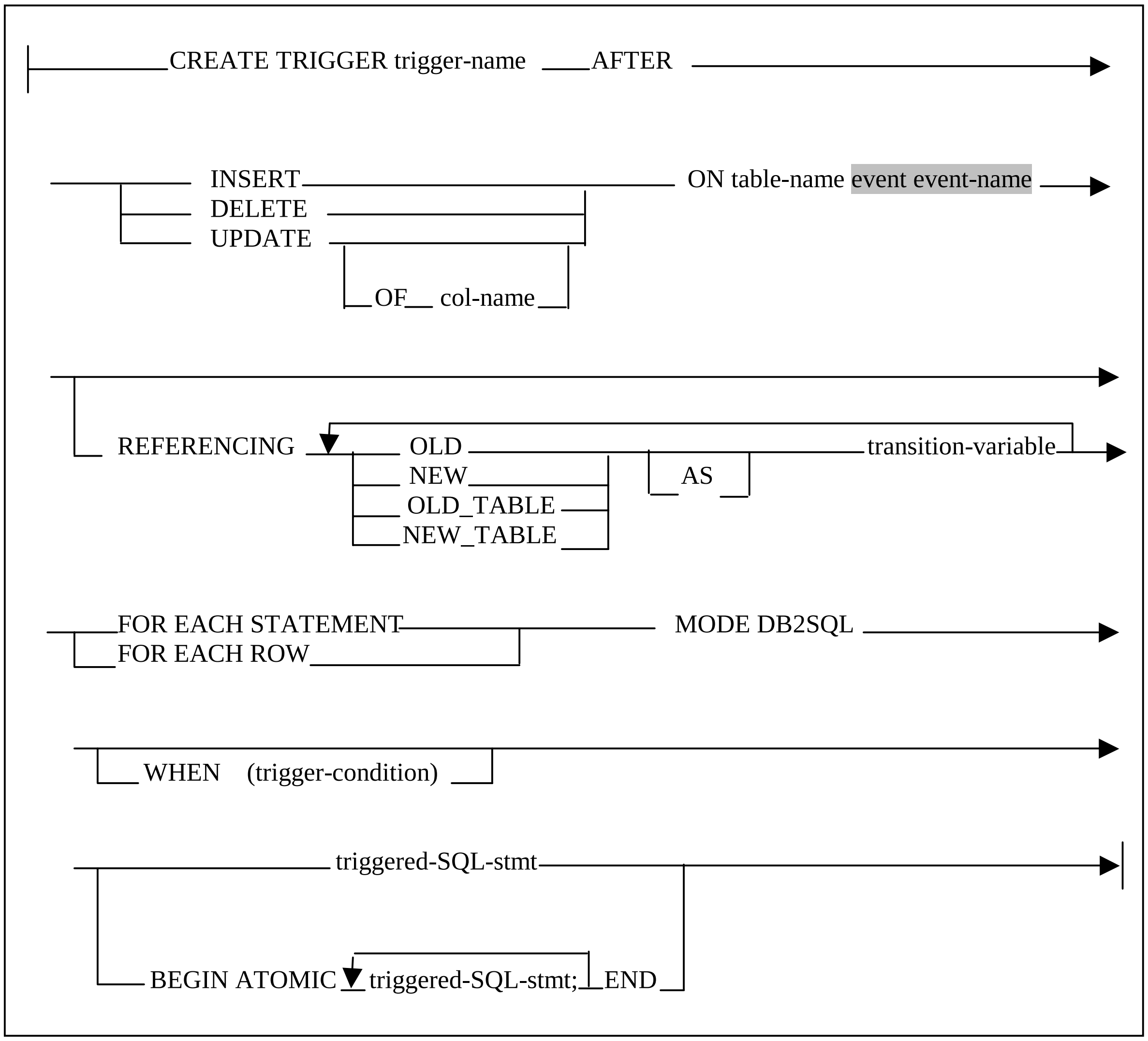


Figure 4.2. Primitive Event Trigger Syntax.

4.1.2 Composite Event Triggers

DB2 as well as other commercial DBMSs such as Oracle, Sybase do not support triggers on composite events. In this implementation, users can create triggers on composite events including parameter contexts, coupling modes and priority. In order to provide transparency, there is a need for a composite event trigger syntax that reflects the trigger syntax of the underlying DBMS. The client must also be able to specify ECA features such as parameter contexts, coupling modes and priority. Considering these issues, we have introduced composite event trigger syntax and is shown in figure 4.3.

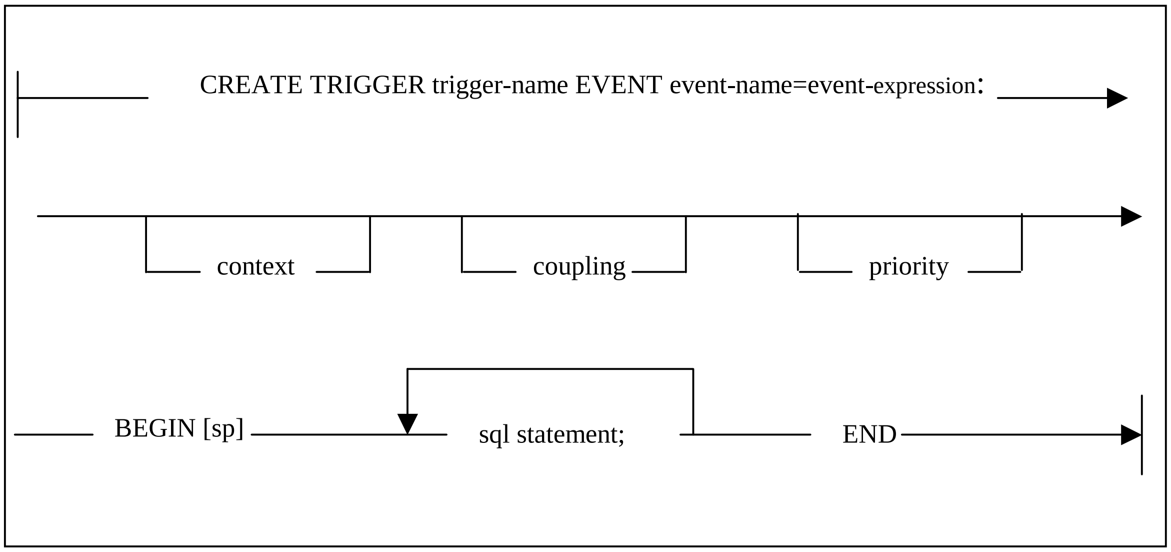

CREATE TRIGGER trigger-name EVENT event-name=event-expression:
context
coupling
priority
BEGIN [sp]
sql statement;
END

Figure 4.3. Composite Event Trigger Syntax.

Trigger-name- each trigger is associated with a unique trigger name.

Event-name – represents the name of the triggering event.

Event-expression – is the composite event expression in Snoop.

':' – is introduced to facilitate parsing of the composite event trigger.

Context – is the parameter context and can be recent, chronicle, continuous, cumulative; default value is recent.

Coupling – is the coupling mode for the execution of rule and can be immediate, detached; default value is immediate.

Priority – is an integer value; default value is 1.

SP- indicates that the user-defined actions have to be executed in a single transaction as a stored procedure.

Example 4.2: Composite Event Trigger

```
create trigger t_andWeather event andWeather = addwTexas ^ addwCalif:
                recent    immediate    1
                begin
                insert into WeatherNational
                select * from WeatherTexas_inserted_tmp;
                insert into WeatherNational
                                select * from WeatherCalif_inserted_tmp;
               insert into notifyUser values('andWeather occurred');
              end
```

4.2 Overview of Relevant Work

Snoop:

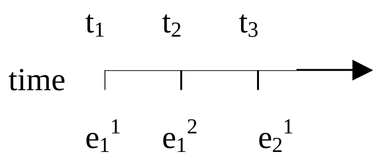
The event specification language Snoop was developed as part of the Sentinel system. Snoop supports the following events as discussed in [10, 11]: OR, AND, SEQ, NOT, APERIODIC, APERIODIC*, PERIODIC, PERIODIC* and PLUS. There are four different parameter contexts identified by Snoop as discussed in [10]. They are recent**,** chronicle, continuous and cumulative**.** Rules can be defined on both primitive and composite events. When an event is detected, the rule is fired. Each rule consists of a condition and an action. A condition evaluates to either true or false. The action is executed only if the condition evaluates to true. A rule can be specified with a coupling mode and a priority. Coupling mode describes when a rule is to be executed relative to the event firing the rule. The coupling modes defined in Snoop are Immediate, Deferred, and Detached. A rule is associated with a priority in addition to parameter context and coupling mode. The priority of a rule is an integer value. Rules with same priority are executed concurrently. Rules of higher priority are executed before rules of lower priority. The default priority of a rule is 1.

Java Local Event Detector (Java LED):

The Java LED was developed [15] to detect primitive and composite events specified in Snoop in a Java environment. Events can be registered with the Java LED by using the set of API provided by the Java LED. The application program calls an API to register the event with the Java LED. Both primitive and composite events can be registered with the Java LED. Java LED also provides API to create rules on events and also to drop events and rules. In order to illustrate the detection of composite events, let us consider the following example:

event andEvent = AND(e1,e2)

Let the event occurrences be as shown.



The AND event is detected when e_2^1 occurs. In the recent context, the AND event is detected by pairing e_1^2 and e_2^1. In the chronicle context e_1^1 and e_2^1 are paired. In the continuous context, two events $e_1^1e_2^1$ and $e_1^2e_2^1$ are detected at the same time. In the cumulative context, a single event is detected with constituent events $e_1^1e_1^2e_2^1$. A primitive event is associated with a single parameter list whereas a composite event is associated with a list of parameter lists. When an event is detected its parameters are propagated to all its subscriber events. It is possible to search through the list of parameter lists of a composite event to obtain a particular parameter list belonging to one of the constituent events. Java LED provides a set of API for this purpose.

4.3 Need to Use Java LED

The DB2 server detects only primitive events. We need a mechanism to detect composite events so that active capability can be enhanced. Java LED provides an option to detect composite events and also supports parameter context, coupling mode and priority of rule execution. Further Java LED is portable since it is a stand-alone application. Because of these features, we use Java LED to support composite events. By using the Mediator, we can map the database events to events on the Java LED. When the user specifies a primitive event trigger, the Mediator can register the event with Java LED by using the primitive event API. An event node is created in the Java LED for the event. Now when the user creates a composite event trigger, the Mediator can register the composite event using the composite event API. An event graph is created by the Java LED to maintain the relationship between the primitive and composite events.

4.4 MultiUser/MultiDatabase

Most of the commercial DBMSs are based on the client/server paradigm. Multiple users can connect to multiple databases. One of the important issues of an active database system is to provide active capability without degrading the functionality of the underlying system. Hence, our Mediator design is also based on the client/server paradigm in order to maintain the multiuser/multidatabase functionality of the underlying system. The Mediator consists of a server application (termed the Mediator-Server) and a client application (termed the Mediator-Client). The Mediator-Client and Mediator-Server communicate by means of sockets. By using the Mediator-Client interface a user can specify ECA events as well as conventional SQL statements. From the users point of view the Mediator-Server is a virtual SQL server. From the system point of view, the Mediator-Server is the middleware that processes the client request before sending it to the DB2 server. Multiple clients (users) can connect to different databases. The Mediator-Server spawns a new thread for each of the clients to service their requests. The Mediator-Server creates files that contain the API for registering events and rules with Java LED for each user, database combination. These files associated with each of the users are placed in separate directories so that they don't replace one another. The directory name is generated as follows:

directory Name= "userName"+ "_"+ "databaseName"

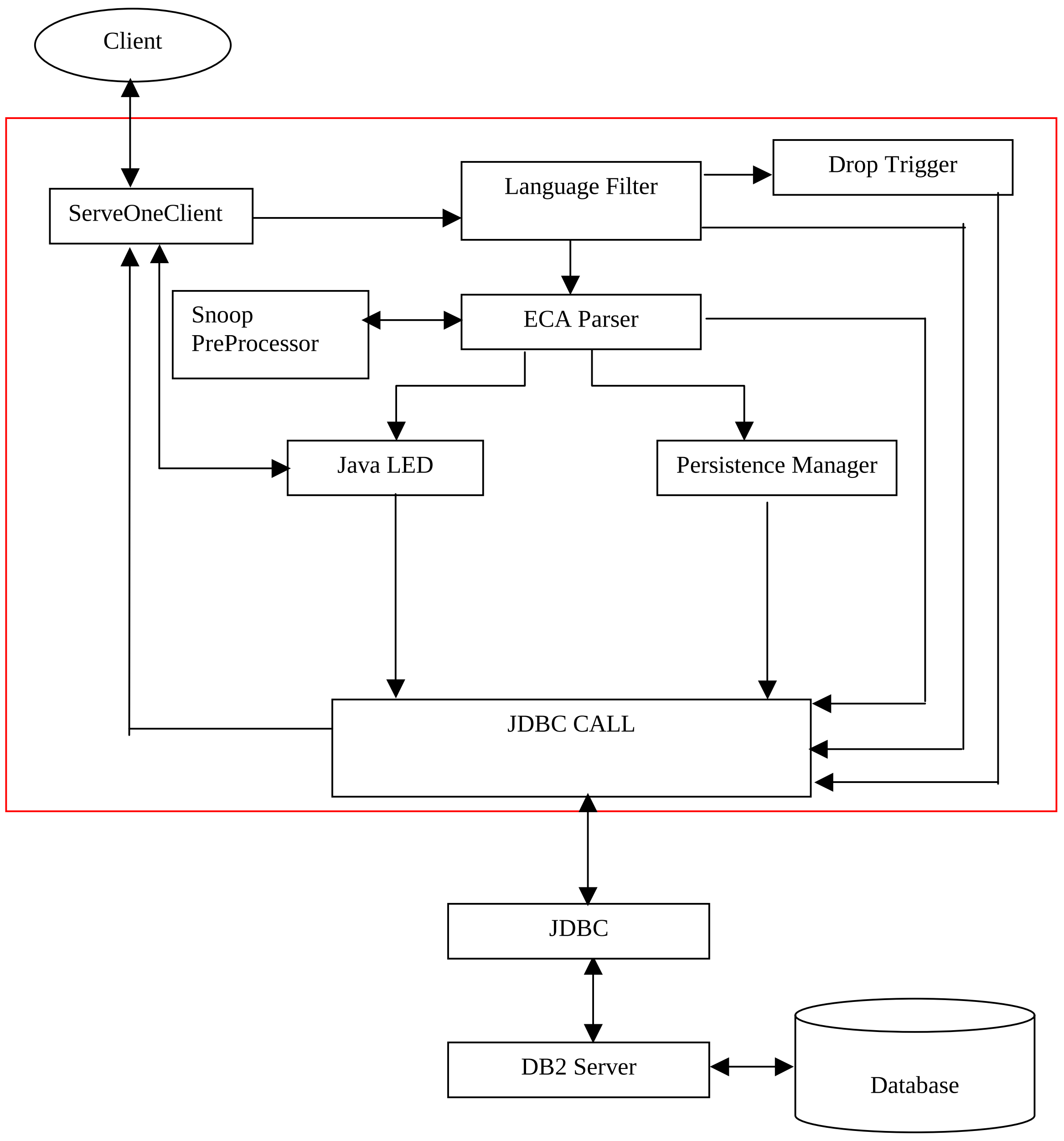


Figure 4.4. Mediator-Server Architecture.

4.5 Architecture of Mediator-Server

The architecture of the Mediator-Server is based on some of the design issues discussed in the previous sections. The Mediator-Server is shown in figure 4.4. The different functional modules of the Mediator are:

4.5.1 ServeOneClient

The Mediator-Server needs to authenticate a client before it can process it's request. This module performs the client authentication and then sends the request to the Language Filter. As already mentioned, detection of composite events is done by the Java LED. In order to detect composite events, the primitive events need to be detected by Java LED. But, the insert/delete/update operations on which primitive events are defined are database operations. The primitive events need to be notified to the Java LED so that they can be detected. The ServeOneClient module does this notification.

4.5.2 Language Filter

This module is used to filter an ECA command from other SQL commands. If it is a SQL command, the Mediator-Server does not have to do any processing. It sends the command to the DB2 server through JDBC. An ECA command can be a create event trigger or a drop event trigger command. If it is a drop event trigger command, it is sent to the Drop Trigger module. A primitive or composite event trigger command is sent to the ECA Parser. The Language Filter is shown in figure 4.5.

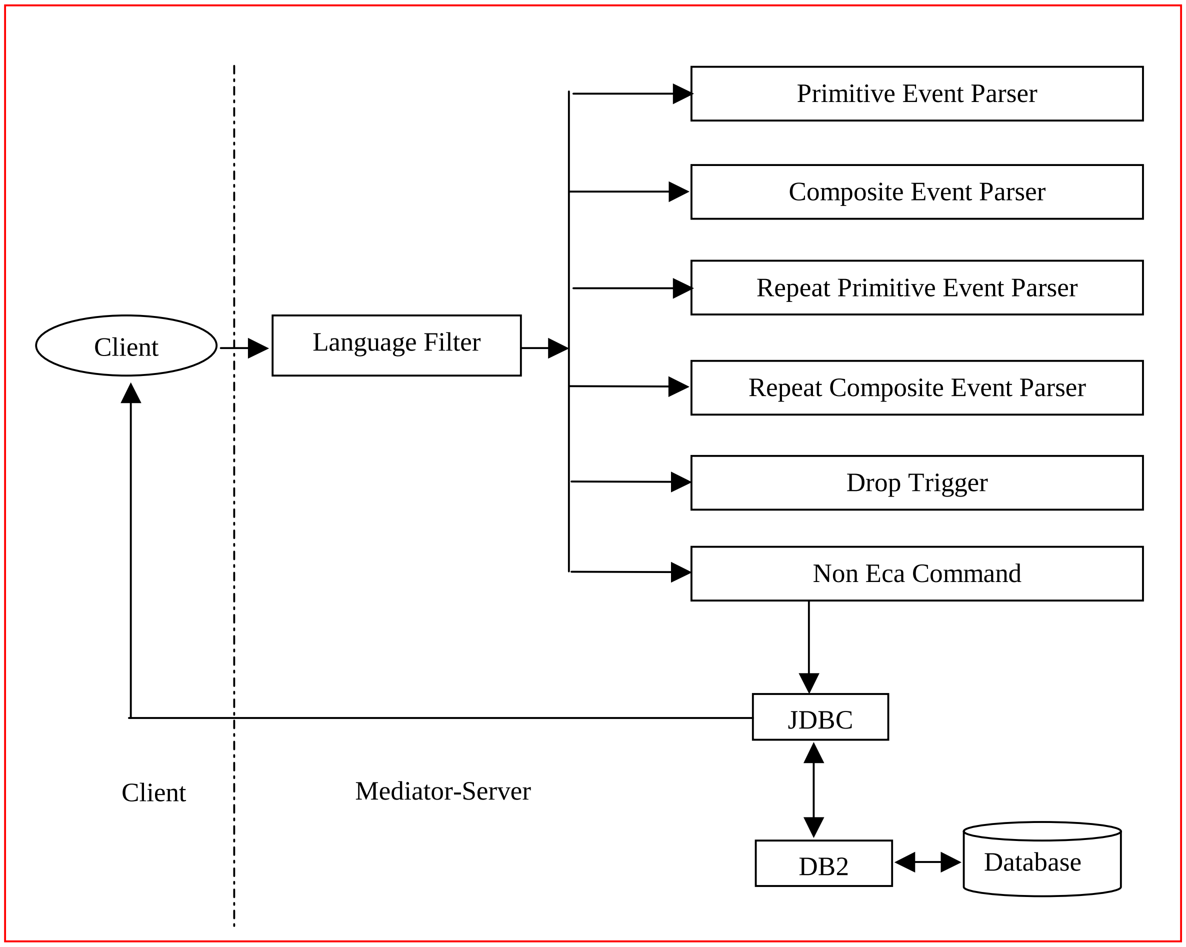


Figure 4.5. Language Filter.

4.5.3 ECA Parser

The ECA Parser parses the ECA command sent by the Language Filter. The ECA command sent to the ECA Parser can be a primitive event trigger, repeat primitive event trigger or composite event trigger, or repeat composite event trigger. Hence, the ECA Parser contains four functional sub-modules Primitive Event Parser, Repeat Primitive Event Parser, Composite Event Parser, and Repeat Composite Event Parser to parse different types of ECA commands. The ECA parser scans the ECA command for errors. If there are no errors, the ECA parser creates SQL commands for Mediator-Server actions and also generates event API and registers the event with the Java LED.

4.5.4 Persistence Manager

One of the main requirements of an active database system is to persist the events and rules. When a client specifies events (primitive or composite) and rules, the ECA Parser creates event API and registers the events with the Java LED. The relationship between the events is maintained by means of an event graph in the Java LED. This event graph will be lost if the client logs out or the session ends. Java LED does not have a mechanism to persist the event graph. Hence, there is a need to persist the events and rules so that the event graph can be recreated when the system restarts. The Mediator-Server persists the events and rules through the Persistence Manager. The Persistence Manager persists the events and rules by using the underlying RDBMS. Once the events and rules are persisted, they can be restored by calling the same API that was used for registering the events with the Java LED. The events and rules are restored when a client session restarts again. Further there is a need to store the primitive event parameters so that they can be used when composite events are detected. This storage of primitive event parameters for future usage is also done by the Persistence Manager. The Persistence Manager uses the tables 4.1, 4.2, 4.3 to perform the above discussed tasks.

SysPrimitiveEvent: This table is used to store primitive event information and is shown in table 4.1.

Table 4.1. SysPrimitiveEvent

Db Name	User Name	Event Name	Table Name	Operation	BeAf operation	Time stamp	VNo	Column Names

Here,

DbName-is the name of the database to which the user connects.

Username-is the valid user name.

EventName-is the name of the primitive event.

TableName-represents the table on which the primitive event is defined.

Operation-is the triggering operation and is either insert or delete or update.

Beafoperation-is either before or after the triggering event.

Timestamp-is the timestamp of event creation.

VNo-is the version number of the occurrence of the event and it is an integer.

ColumnNames-contains the column names on which an update of columns operation is defined. If the primtive event trigger is defined for an insert or delete or just an update of table, columnNames value is ‘none’.

SysCompositeEvent: This table is used to store information about composite events and it is as shown in table 4.2.

Table 4.2. SysCompositeEvent

DbName	UserName	EventName	EventDescribe	Timestamp	Coupling	Context	Priority

Here,

DbNam*e* , UserName and Timestamp are the same as in the previous table 4.1.

EventName- is the name of the composite event.

EventDescribe-is the composite event expression in Snoop. For example, if the composite event is andWeather=addwTexas ^ addwCalif, then eventDescribe is addwTexas^ addwCalif .

Coupling, Context, Priority – represent the coupling, context and priority of the rule execution for a composite event.

SysEcaTrigger: This table is used to persist all the triggers defined in the system and it is as shown in table 4.3.

Table 4.3. SysEcaTrigger

dbName	userName	triggerName	triggerProc	timestamp	eventName

Here,

DbName, Username and Timestamp are the same as in the tables 4.1, 4.2.

TriggerName-is the name of trigger created for the event

EventName-is the name of either primitive or composite event

TriggerProc-is the name of the stored procedure created for the event. At present, this column value is null but can be used for future use. For example, the action part of the composite event can be specified as a stored procedure so that it executes as a single transaction.

4.5.5 Drop Trigger

A user can create events by specifying triggers. Similarly, a user can drop events by specifying a drop trigger statement. In order to maintain transparency, the drop trigger statement is similar to the drop trigger statement of DB2. A user can specify a drop trigger statement as shown below.

drop trigger triggerName

The Drop Trigger Module is as shown in figure 4.6. The Drop Trigger Module first checks if the drop trigger statement is for a simple trigger created on DB2. If so, it sends it directly to the DBMS through JDBC. But if the drop trigger statement is for a trigger defined on a primitive or composite event, then this involves deleting all the event information from the system tables and also dropping the triggers created by the Mediator-Server for the event. Finally, the event node has to be deleted from the Java LED so that there is no further detection of the event. However, this raises an important issue. If a drop trigger is defined on a primitive event, and if the primitive event is the constituent event of some composite event, then the primitive event node cannot be deleted until the composite event has been dropped. Java LED provides API to delete event nodes. When a primitive event node is deleted, Java LED does not check whether the event node is the constituent event of some composite event. Hence, if the primitive event node is deleted, the composite event will not be properly detected. In order to deal with this problem, we use a table SysDrop in which we enter information about all the composite events and their constituent events created by the user. Before dropping an event node, we check the SysDrop table to see if the event is a constituent event of some composite event. This table is as shown in table 4.4.

Table 4.4. SysDrop

ConsEventName	Context	CompEventName

Here,

ConsEventName is the constituent event name

Context is the context in which the composite event is detected

CompEventName is the name of the composite event

For example, for the following composite event, the SysDrop table will have the values as shown in table 4.5.

create trigger t_andW event andWeather=addwTexas^addwCalif:recent begin…

Table 4.5. SysDrop Values for Drop Trigger Example

ConsEventName	Context	CompEventName
addwTexas	recent	andWeather
addwCalif	recent	andWeather

Before dropping an ECA trigger, the Mediator checks the table SysDrop to see if the event is a constituent event of some composite event. If yes, the node cannot be deleted in the Java LED. The Drop trigger module has two functional sub-modules Drop Primitive Trigger and Drop Composite Trigger to drop events and triggers.

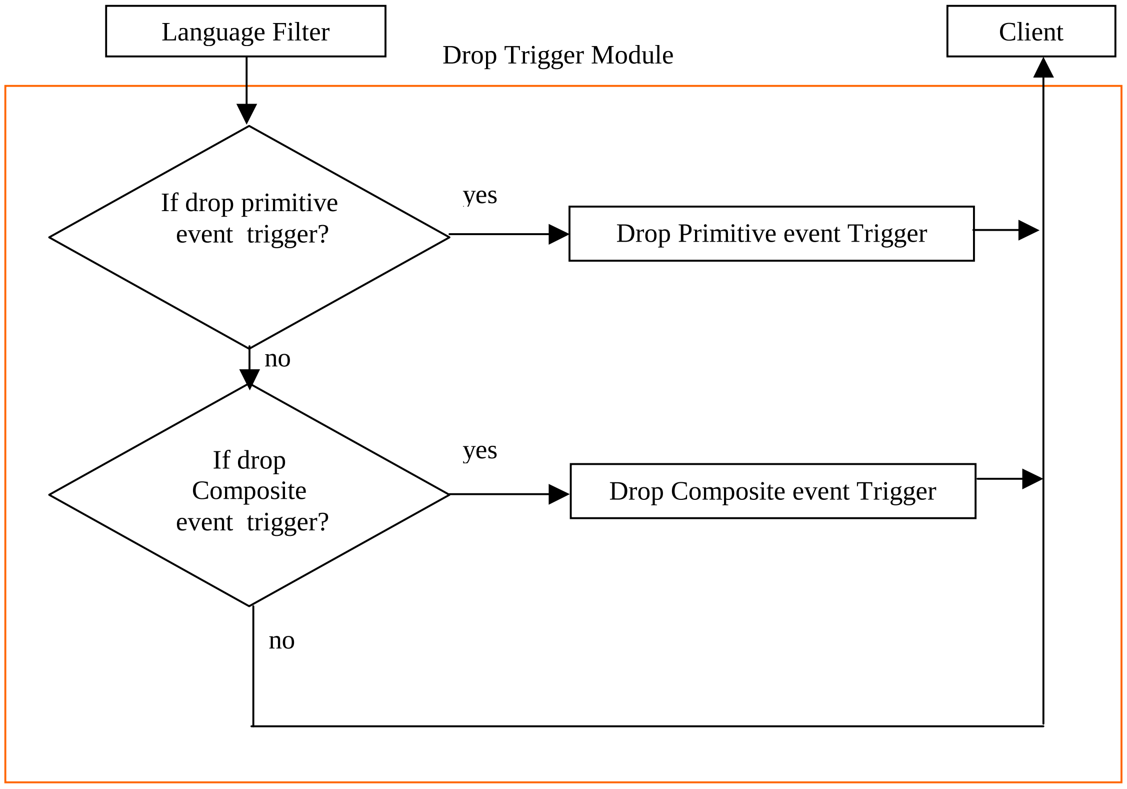

Language Filter
Client
Drop Trigger Module
If drop primitive event trigger?
yes
Drop Primitive event Trigger
no
If drop Composite event trigger?
yes
Drop Composite event Trigger
no

Figure 4.6. Drop Trigger Module.

4.6 Need for Notification

When a primitive event occurs, it occurs on the DB2 server side. For example, let us consider a primitive event 'addWeather' defined for an insert operation on Weather table. When an insert happens on the Weather table, the primitive event 'addWeather' occurs and the SQL statements in the trigger for the primitive event will be executed by the DB2 server. Now this raises the question, how to detect composite events using the Java LED if the primitive event occurs on the DB2 server side. When the user creates a primitive event trigger, the Mediator-Server registers the event with Java LED using the API provided. In order to detect a composite event, the constituent primitive events need to be detected by the Java LED. In the Sybase ECA Agent discussed in chapter 2, there is a build-in function, 'sybase-SendMessage(port, IP address,method)" which can be used to make a RPC call from the body of the trigger to raise the primitive event in the specified IP address. In our implementation JDBC is used for database connectivity. It is not possible to make a RPC call or RMI call from inside the body of trigger, to raise the event in the Java LED since JDBC does not allow it. In order to solve this problem, we use a table Notify in which we insert the event information, table name on which the event is defined and version number of the occurrence of the event. When a primitive event occurs, the body of the trigger associated with the primitive event contains SQL commands to insert the event information into the notify table. The Mediator-Server checks for the presence of event information in the Notify table. If an event is present, the Mediator-Server calls API for raising the event. Now the Java LED detects the primitive event.

Table 4.6. Notify

EventName	TableName	VNo

4.7 Composite Event Detection

Composite events are detected using Java LED. For example, let us consider the composite event defined by the trigger shown in example 4.2.

The composite event is

event andWeather = addWTexas ^ addWCalif

The event *andWeather* needs to be detected when the primitive events *addWTexas* and *addWCalif* occur. Here the primitive event *addWeather* denotes the event of inserting tuples into table WeatherTexas. WeatherTexas is used to store Weather information for the different cities in Texas and is as shown in table 4.7.

Table 4.7. WeatherTexas

city	time	temp	wspeed

Similarly the primitive event *addWCalif* denotes the event of inserting tuples into table WeatherCalif. WeahterCalif is used to store Weather information for the different cities in California and is as shown in table 4.8.

Table 4.8. WeatherCalif

city	time	temp	wspeed

Let the composite event *andWeather* be used to monitor the weather information in Texas and California. The composite event *andWeather* needs to be detected when the primitive events *addTexas* and *addCalifornia* are detected.

The issue now is how to detect the composite event using Java LED. It may be recalled that a composite event is registered with the Java LED by calling the composite event API. We also create a rule containing the parameter context, coupling mode, priority, condition method, action portion for the composite event. We already discussed that the primitive event is raised by the Mediator-Server and detected by the Java LED. If the primitive event detection satisfies the condition for composite event detection, the composite event is detected by the Java LED and the rule is executed.

4.7.1 Getting Parameter Context

Java LED supports the following parameter contexts for the detection of composite events: Recent, Chronicle, Continuous, Cumulative. The primitive events are database operations.

Now the issue is, if the primitive event occurs on the DB2 server side, how to get the primitive event parameters in the composite event action. When a primitive event occurs, the trigger associated with the event is fired. In DB2, the tuples that are inserted on a table are inserted into the transient table ‘NEW_TABLE’ and the tuples that are deleted from a table are inserted into the transient table ‘OLD_TABLE’. These tables are accessible only during the execution of the body of the trigger. The tuples need to be accessed outside the scope of the trigger so that the parameter context can be supported. Hence, we create tables R_inseted, R_deleted to store the tuples inserted, deleted on a relation R when a primitive event occurs. These tables are first created when the user defines a primitive event. Here ‘R’ is the name of the table on which the primitive event is defined. For the primitive event ‘addwTexas’ the tables Weather_inserted, Weather_deleted are created. These tables contain the parameters (tuples) for every occurrence of the primitive event. These tables contain an additional attribute ‘vNo’. This attribute is used to distinguish the tuples for different occurrences of the event. This attribute version number is obtained by doing a join between the R_inserted/R_deleted table and Version table. The Version table has been introduced to keep a global count of the occurrence of a primitive event.

Table 4.9. Version

vNo

Table 4.10. Weather_inserted/Weather_deleted

city	time	Temp	Wspeed	vNo

Now when the composite event is detected, the user should be able to access the parameters of the primitive event occurrences that resulted in the detection of the composite event. When the user specifies a composite event trigger, the user can access the parameters for a particular context by specifying the table name ‘R_inserted_tmp’ or ‘R_deleted_tmp’. Here ‘R’ represents the name of the relation on which the constituent events are defined. Just like the temporary tables ‘OLD_TABLE’, ‘NEW_TABLE’ in DB2 that are accessible in the body of the trigger, we create two tables ‘R_inserted_tmp’, ‘R_deleted_tmp’ which are accessible to the user in the action portion of the composite event trigger. While the tables, ‘R_inserted/R_deleted’ contain parameters for every occurrence of the primitive event; the tables ‘R_inserted_tmp/R_deleted_tmp’ need to contain only the parameters of the constituent events for that particular context of the composite event detection. For the WeatherTexas table shown in table 4.7 and the WeatherTexas_inserted_tmp /WeatherTexas_deleted_tmp is as shown in table 4.8.

Table 4.11. WeatherTexas_inserted_tmp/WeatherTexas_deleted_tmp

City	time	temp	Wspeed	VNo

We already discussed that the Mediator-Server notifies the primitive event occurrence to the Java LED. Before raising the primitive event, the tuple(s) whose insertion/deletion/modification that resulted in the primitive event occurrence need to be passed to the Java LED, so that they can be accessed in the composite event action portion.

Java LED provides API to insert parameters before raising the primitive event. But, if there are many tuples involved or if the tuples have many attributes, then passing these tuples will result in lot of overhead. Hence we need to pass minimum parameters for each occurrence of an event, so that these parameters can be used in the action portion of the composite event to access the actual tuples that were inserted/deleted/updated. We create a table SysContext for this purpose and is as shown in table 4.12.

Table 4.12. SysContext

Tablename	Context	VNo

Here, tableName is the name of the table on which the constituent event is defined. ‘context’ is the context in which the composite event is detected. vNo is the version number of the constituent event.

The Mediator-Server inserts parameters event name, table name and version number before raising a primitive event. When the event ‘addwTexas’ occurs for the first time (e_1^1), the parameters addwTexas, WeatherTexas, 1 are inserted into the primitive event node before raising it. The parameter list for this occurrence is propagated to the composite event node. Now if the same event ‘addwTexas’ occurs again (e_1^2), the parameter list containing the parameters event name (addwTexas), table name (WeatherTexas), version number (2) is propagated to the composite event node. This list replaces the previous parameter list because the composite event is being detected in the recent context. Now if the event ‘addwCalif’ occurs (e_2^1), the parameter list containing the parameters (addwCalif), table name (WeatherCalif), version number (3) is propagated to the composite event node. Now the composite event (e_1^2, e_2^1) is detected by the Java LED in the recent context and the action is invoked. The parameter lists can be accessed when the action is invoked. The parameter lists of the constituent events are converted to tuples containing table name, context and version number and are inserted into the table SysContext. The Syscontext table will now contain the following values:

Table 4.13. SysContext Values for AndWeather

Tablename	Context	VNo
WeatherTexas	recent	2
WeatherCalif	recent	3

By joining the tables SysContext and R_inserted/R_deleted with version number as the join attribute, the correct parameter context can be accessed and are inserted into tables R_inserted_tmp/R_deleted_tmp. The implementation details of composite events are discussed in chapter 6.

4.8 Need to Use Snoop Preprocessor

We use Java LED to detect composite events. Events and rules are created in the Java LED by calling the API provided. The user specifies the composite event expression as a part of the composite event trigger. It is easier for the user to specify this composite event in Snoop that has well defined event semantics rather than specifying the Java LED API. Since the user specifies the events in Snoop, we need to parse the Snoop expression and then generate the Java LED API. A Snoop Preprocessor has already been developed for this purpose. The Snoop Preprocessor takes a composite event expression, rule creation expression as input and generates the corresponding Java LED API. Since the Snoop Preprocessor with these features is available, we use the Snoop Preprocessor to parse the composite event expression.

4.9 Need to use JDBC

Java Database Connectivity (JDBC) provides database access for Java applications. JDBC is a generic SQL database access framework. JDBC allows writing code independent of a particular DBMS. By using JDBC, we have the option of porting this Mediator to different DBMSs (Oracle, Sybase etc.). Hence, JDBC has been used to provide database connectivity in this implementation.

4.10 Mediator-Server Work Flow

The Mediator is responsible for creating ECA rules, event notification and restoration of ECA rules when the system restarts. The workflow of the Mediator-Server is as follows:

- The first step the Mediator-Server does is to get the input sent by the client. The input can be an ECA statement or a simple SQL statement. The ServeOneClient module in the Mediator receives the input sent by the client.
- The ServeOneClient then checks for the presence of system tables. The system tables are SysEcaTrigger, SysPrimitiveEvent, SysCompositeEvent, SysContext, Version, and Notify. This is done by checking for the presence of Notify table in the DBMS. If Notify table is present, it means that the other system tables are present. If Notify is not present, the ServeOneClient creates all the system tables in the DBMS with Notify as the last table.
- The ServeOneClient checks for the presence of a unique directory for the given user. All the files created on behalf of a user are stored within the user directory. If the user directory is not present, the ServeOneClient creates the user directory.
- The ServeOneClient restores the ECA events and rules each time the system is restarted. The restoration of rules is done through the Persistence Manager. The ServeOneClient also resets the version number in the Version table to ‘0’. Since the Version

table keeps track of the version number of the occurrence of a primitive event and it is global, it needs to be reset when the system is restarted.

- After doing the above steps, the ServeOneClient then sends the input statement to the Language Filter. From the Language Filter, the input command is sent to the different modules of the Mediator-Server or directly to the DB2 server through JDBC.
- The ServeOneClient checks the Notify table to see if any event has occurred. If it has occurred, the ServeOneClient then raises the event and the Java LED detects the event. If some composite event is defined, the Java LED detects the composite event and executes the action.

Summary:

This chapter discussed the design issues of the mediator-based approach. The chapter also gave a brief overview of the event specification language Snoop. It also explained the reasons to use the Snoop Preprocessor and Java LED to support composite events. The chapter also dealt with the various functional modules of the Mediator-Server and the need to persist the events and rules. The issues involved in detecting composite events and supporting parameter context have also been discussed.

CHAPTER 5

IMPLEMENTATION OF PRIMITIVE EVENTS

This chapter discusses how the Mediator implements Primitive Events. This chapter also discusses implementation of triggers on existing events (termed Repeat Primitive Event Triggers) and also dropping primitive event triggers. Extending the native trigger syntax of DB2 by adding the keyword event followed by the event name specifies a primitive event. In order to explain the implementation of primitive events, the following three demo tables are considered.

Table 5.1. Weather

city	Time	temp	wspeed

The Weather table records the temperature and wind speed for a specific city at a particular instant of time.

Table 5.2. Temperature

city	Htem	ltem	avgtem

The Temperature table records the maximum temperature, minimum temperature and average temperature for each city over a period of time.

Table 5.3. Wspeed

city	hspeed	Lspeed	avgspeed

The WSpeed table records the high wind speed, low wind speed and average wind speed for each city over a period of time.

Let a primitive event *addWeather* be defined as shown in figure 5.1. Whenever tuples are inserted into table Weather, the trigger *t_addw* will be fired and the event *addWeather* occurs. This trigger inserts the maximum temperature, minimum temperature, and average temperature into the Temperature table for every tuple inserted into the Weather table.

The ServeOneClient module in the Mediator-Server first sends the input statement from the client to the Language filter. Since the example shown in figure 5.1, is a primitive event trigger, the Language Filter sends it to the Primitive Event Parser module in the ECA Parser for parsing the trigger statement and generating the primitive event.

```
create trigger t_addw  after insert on weather event addWeather
            referencing new as newrow
            for each row mode DB2Sql
            begin atomic
            delete from temperature where city=newrow.city;
            insert into temperature values(newrow.city,newrow.tem,newrow.tem,newrow.tem);
            update temperature set htem=(select max(tem)
                                            from weather where city=newrow.city)
                                          where city=newrow.city;
            update temperature set ltem=(select min(tem)
                                          from weather where city=newrow.city)
                                         where city = newrow.city;
            update temperature set avgtem=(select avg(tem)

                                                          from weather where city=newrow.city)
```

Figure 5.1. Primitive event trigger example.

5.1 Primitive Event Parser

The primitive event parser is as shown in figure 5.2. This parser checks for the presence of the keyword *event* after the operation and the table. If the keyword *event* is present, the parser takes the word following the keyword *event* as the event name. In the example being considered, the keyword *event* followed by the event name is present after the operation and the table.

create trigger t_addw after insert on weather event addWeather

The parser then checks if the primitive event trigger has already been defined in the system, because no two triggers can have the same name in a DBMS. The table SysEcaTrigger has information about every ECA trigger created in the system by that user

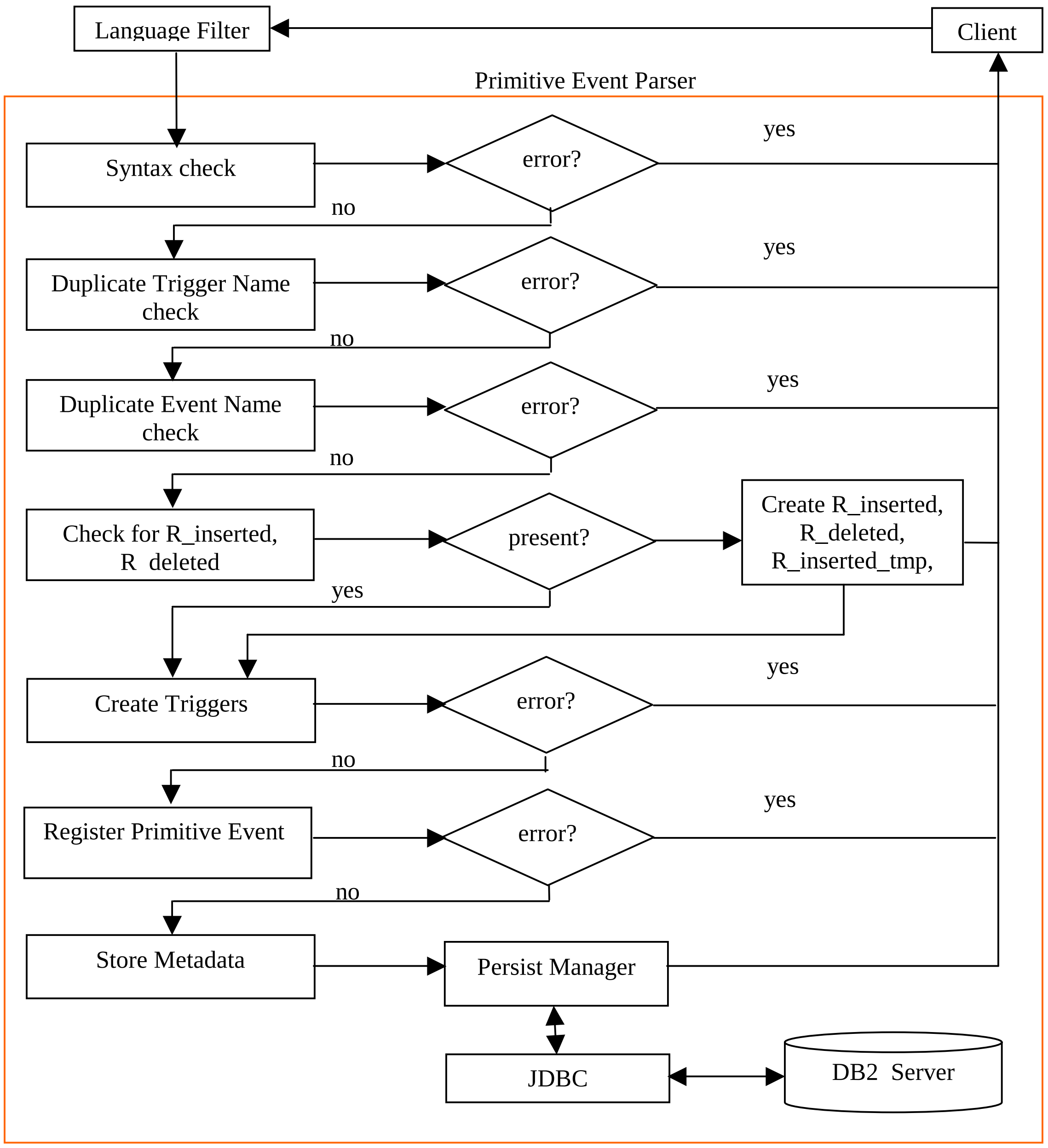


Figure 5.2. Primitive Event Parser.

for that database. By checking the SysEcaTrigger table, the parser finds out if a trigger with the same dbName.username.eventname is already present. If present, the parser returns an error. We create a primitive event node in the Java LED for every primitive event trigger defined by the user so that it can be used in the detection of composite events. Hence the parser has to make sure that no two primitive events have the same name. The table SysPrimitiveEvent has information about every primitive event defined in the system. By checking the SysPrimitiveEvent table, the parser finds out if the primitive event has already been defined in the system. If the name has already been defined in the system, the parser returns an error to the client. In the example being considered, the parser checks if the event name *addWeather* is already present in the SysPrimitiveEvent table. If the event name is not present, the parser proceeds further.

5.1.1 Creation of Temporary Tables

We have already discussed in the design issues, that in order to support composite events and parameter context, we create the tables *R_inserted/R_deleted* and *R_inserted_tmp/ R_deleted_tmp* to store the tuples inserted/deleted in table ‘R’. Here ‘R’ is the table on which the primitive event is defined. For the ‘Weather’ table shown in table 5.1, the tables *Weather_inserted/Weather_deleted* and *Weather_inserted_tmp/ Weather_deleted_tmp* are as shown by tables 5.4 and 5.5.

Table 5.4. Weather_inserted/Weather_deleted

city	time	temp	Wspeed	vNo

Table 5.5. Weather_inserted_tmp/Weather_deleted_tmp

city	time	Temp	wspeed	VNo

5.1.2 Creation of Triggers

When a primitive event occurs, a series of actions need to be performed before raising the primitive event. The actions to be done are:

Update Version Number in SysPrimitiveEvent Table:

This is done so that we can keep track, the number of times that particular event occurred. For the example shown in figure 5.1, if the primitive event *addWeather* occurs three times, the version number of event *addWeather* has to be incremented by one for each of the three times.

update SysPrimitiveEvent set vNo=vNo+1 where eventName= 'addWeather';

Update Version Number in Version Table:

It may be recalled that the version number is used to distinguish between different occurrences of an event. Hence, each time a primitive event occurs the version number in table Version has to be incremented by one so that each occurrence of an event has a unique version number. The initial value of the version number is '0'.

update version set vNo=vNo+1;

Insert into 'R_inserted'/ 'R_deleted' tables:

As mentioned in the design issues, the primitive event parameters need to be collected so that they can be used later for composite event detection to support parameter context. The example being considered is a primitive event defined for an insert operation. The inserted tuples will be present in the transient table 'NEW_TABLE'. Since this table vanishes once the trigger action is complete, the tuples from ‘NEW_TABLE’ are inserted into *Weather_inserted* so that they can be used for parameter context.

*insert into weather_inserted select * from newtable, Version;*

Here ‘newtable’ is the variable representing the transient table ‘NEW_TABLE’ containing all the inserted tuples for that particular insert operation in DB2. If the event *delWeather* is defined for a delete operation, the deleted tuples will be inserted into *Weather_deleted* table.

*insert into weather_deleted select * from oldtable, Version;*

Here ‘oldtable’ is variable representing the transient table ‘OLD_TABLE’ containing all the deleted tuples for that particular delete operation in DB2. Further, if we have an event *updWeather* defined for an update operation, then the old tuple before the update operation has to be inserted in the *Weather_deleted* table and the new tuple after the update operation is inserted in the *Weather_inserted* table. This is done because an update operation is a delete followed by an insert operation.

*insert into Weather_deleted select * from oldtable, Version;*

*insert into Weather_inserted select * from newtable, Version;*

Insert into Notify:

The Mediator application raises the primitive event after checking for the occurrence of the event (insert/delete/update operation) on the DBMS server side. This is done by checking for the presence of the event in the Notify table. Hence the event information tuple(eventName, tableName, vNo) has to be inserted into the Notify table. For the example being considered, we insert the following tuple into the notify table.

insert into Notify select eventName, tableName, Version.vNo

from SysPrimitiveEvent, Version where eventName= ‘addWeather’;

When an insert/delete/update operation for which the primitive event is defined occurs, all the above-mentioned actions need to be performed and also the user defined action portion of the primitive event trigger has to be executed. The solution is to create a trigger for this purpose. But a user can create additional actions (triggers) for the primitive event by specifying a repeat primitive event trigger. There can be multiple repeat primitive event triggers defined on the same event. When the event occurs, all the triggers (primitive event trigger, repeat primitive event triggers) associated with the event need to be executed. If the actions such as parameter collection, version number update, and inserting event notification information are clubbed together with the user-defined actions, then they will be executed for every trigger that is defined for the event. This will result in multiple parameter collection, version number updates and event notifications.

In order to avoid this, there is a need to demarcate the user-defined action from system actions for a primitive event trigger. Hence the parser creates two triggers 'triggerName' and 'triggerName01' for each type of primitive event defined. Here 'triggerName' is the name of the primitive event trigger specified by the user and it contains

```
create trigger t_addw after insert on weather
        referencing new as newrow
        for each row mode DB2Sql
        begin atomic
        delete from temperature where city=newrow.city;
        insert into temperature values(newrow.city,newrow.tem,newrow.tem,newrow.tem);
        update temperature set htem=(select max(tem)
                        from weather where city=newrow.city)
                        where city=newrow.city;
        update temperature set ltem=(select min(tem)
                        from weather where city=newrow.city)
                        where city = newrow.city;
        update temperature set avgtem=(select avg(tem)
                from weather where city=newrow.city)
                where city=newrow.city;end
```

Figure 5.3. t_addw.

the action portion of the primitive event. For the example being considered, this trigger created is ‘t_addw’ and is shown in figure 5.3. This trigger is similar to the primitive event trigger shown in figure 5.1, but the keyword *event* followed by event name has been removed.

The second trigger ‘triggerName01’ created by the parser contains the SQL statements to do Mediator-Server actions and is as shown in figure 5.4. Out of the two triggers created by the parser, the trigger ‘t_addw’ is a row level trigger, whereas the trigger ‘t_addw01’ is a statement level trigger. The trigger ‘t_addw’ contains action portion of the primitive event trigger. Since the primitive event ‘addWeather’ is defined on an insert ope-

```
create trigger t_addw01 after insert on weather
Referencing new_table as newtable
for each statement mode db2sql
begin atomic
update SYSPRIMITIVEEVENT set VNO=VNO+1 where eventname=’addw’;
update version set vno=vno+1;
insert into weather_inserted select * from newtable,version;
insert into notify select eventname,tablename,version.vno from sysprimitiveevent, version where
eventname=addw;
end
```

Figure 5.4. t_addw01.

ration and the action has to be fired for each row that is inserted, the trigger ‘t_addw’ also has to be a row level trigger. If the primitive event trigger is a statement level trigger, then the trigger ‘t_addw’ should also be a statement level trigger.

But the other trigger ‘t_addw01’ is a statement level trigger irrespective of whether the primitive event trigger is a row level trigger or statement level trigger. The trigger ‘t_addw01’ contains the SQL commands for parameter collection, version number update and event notification that need to be executed when the insert operation occurs. These SQL statements should be executed only once for each occurrence of the event.

For example, let the user input the following statement:

insert into Weather values (‘ arlington’, current timestamp, 80, 23),

(‘austin’,current timestamp,77,22);

Let the trigger ‘t_addw01’ created by the parser be a row level trigger. The above input is a single occurrence of the event *addWeather*, in which the client inserts 2 rows into the Weather table. Since it is a single occurrence of the event, the version number has to be incremented by only one in the Version table. A row level trigger will fire for each row that is inserted. Since there are two rows that are inserted, the version number will be incremented twice for a single occurrence of the event *addWeather*. Also parameter collection and insertion of event notification information in the notify table will occur twice. Hence the trigger ‘t_addw01’ needs to be a statement level trigger and should be executed only once for each occurrence of the event.

5.1.3 Generate Primitive Event API

Any primitive event can be registered with the Java LED by calling the primitive event API of Java LED. For the event *addWeather* being considered, the API for registering the event is as shown in figure 5.5.

In order to generalize the generation of the primitive event API, the event handle name in Java LED is kept the same as the event name. The ECA Agent instance name in the Java LED is given by ‘userName_databaseName’. If ‘nellai’ is the user and ‘ecadb2’ is the database name, then the ECA Agent instance name is ‘nellai_ecadb2’. By creating an LED with this name, all events for this user under this database name is detected by that event detector. In figure 5.5, ‘Led’ denotes the name of the Java class in which the event will be raised. “void c_addWeather()” is the method signature for the event. The user dynamically defines the primitive events. Hence the parser has to register the events dynamically. The parser creates a java file ‘c_eventName.java’ containing the primitive event API .

```
EventHandle addWeather =null;
ECAAgent myAgent = ECAAgent.initializeECAAgent("nellai_ecadb2");
addWeather = myAgent.createPrimitiveEvent("addWeather","Led", EventModifier.BEGIN, "void
c_addw()",  DetectionMode.SYNCHRONOUS);
```

Figure 5.5. Primitive Event API.

This java file needs to be compiled dynamically. The code to compile this java file dynamically is as shown in figure 5.6.

```
String cmd = "javac " + "metadata\\" + dir + "\\" + className + ".java" ;
  java.lang.Runtime rt = Runtime.getRuntime();
  try
  {
    Process pro = rt.exec(cmd);// execute the command
    System.out.println(" Compiling.for primitive event :"+cmd);

    int a = pro.waitFor();  // wait until the current process terminate,
                            // so that the command completed
   System.out.println(" Compile Finished.");
 }
  catch(Exception e)
  {
    System.out.println(e.toString());
  }
```

Figure 5.6. Code to compile c_addWeather.java.

After the file is compiled, primitive event API needs to be called for the event to be registered with the Java LED. The compiled class 'c_addWeather.class' needs to be loaded into memory. The ClassLoader class in Java is used for this purpose. The 'getSystemClassLoader()' method in the ClassLoader class returns the system class loader. The method 'loadClass(className)' is used to load the class

ClassLoader cl= ClassLoader.getSystemClassLoader();

Class cla=cl.loadClass(''c_addWeather'');

Now the method ‘call_addWeather()’ in the loaded class ‘c_addWeather.class’ needs to be invoked to register the event. This is done as follows in the application,

```
Class[] paramTypes = null;              // list of formal parameters
String methodName = “call_addWeather”;   //method name
Object obj = null;                  //for a static method, object argument is null
Object[] params = null;                 // list of actual parameters
try
{
//get the particular method with  the given method name and formal parameter types
java.lang.reflect.Method meth = cla.getMethod(methodName, paramTypes);
//invoke the method
meth.invoke(obj,params)
}
catch (exception e){}
```

The underlying method “call_addWeather” is a static method, hence the specified object argument is null. Since the formal parameters for the method is null, the actual parameters passed to the method when invoked is also null. Once the method “call_addWeather()” is invoked the primitive event is registered with the Java LED.

5.1.4 Store metadata

Once the event has been registered, the parser persists the information needed to reconstruct the primitive event using the Persistence Manager. For the example under consideration, the following tuples are inserted by the Persistent Manager into the tables SysPrimitiveEvent and SysEcaTrigger.

Table 5.6. SysPrimitiveEvent Values for Example

Db Name	User Name	Event Name	Table Name	operation	Beaf operation	Time stamp	vNo	Column Names
ecadb2	nellai	Add Weather	Wea-ther	insert	after	Current timestamp	0	none

Table 5.7. SysEcaTrigger Values for Example

dbName	userName	triggerName	triggerProc	timestamp	eventName
ecadb2	nellai	t_addw	None	Current timestamp	addWeather
ecadb2	nellai	t_addw01	none	Current timestamp	addWeather

5.2 Detection of Primitive Event

The primitive event is a database operation. In the example shown in figure 5.1, the primitive event *addWeather* is defined on the database operation insert. When a tuple is inserted into the table 'Weather', the primitive event *addWeather* needs to be detected by the Java LED. It may be recalled that when the insert operation occurs, the triggers 't_addw' and 't_addw01' get fired. The trigger 't_addw01' inserts the event name, table name, and version number into the table Notify.

Table 5.8. Notify Values for Example.

eventName	tableName	VNo
addWeather	weather	1

The Mediator checks for the presence of the event tuple in the Notify table. If an event tuple is present, the Mediator takes the following steps to raise the event.

Led raiseEvent = new Led();

raiseEvent.PrimEvent(eventName,tableName,vNo);

“Led” is the name of the class in the application that is used for raising the primitive event. It contains the API to raise the primitive event. The “Led” class is as shown in figure 5.7.

```
package server;

import sentinel.led.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

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

EventHandle[] eventHandler = ECAAgent.getEventHandles(eventname);
ECAAgent.insert(eventHandler,"eventname",eventname);
ECAAgent.insert(eventHandler,"tablename",tablename);
ECAAgent.insert(eventHandler,"vno",vno);

ECAAgent.raiseBeginEvent(eventHandler, this);
System.out.println("***** raiseBeginEvent ***** " + eventname);
}

//ECA_Condition
public static boolean True(ListOfParameterLists parameterLists)
{

System.out.println("***** From Condition ***** ");
return true;
}
}
```

Figure 5.7. Led Class.

In figure 5.7, the method getEventHandles(eventname) gets all the event handles for the registered event. The event handle is used to insert the parameter’s event name, table name and version number. This is stored as a list in the Java LED for that particular event. The data structure that is used to store this list is a Parameter List [15]. Calling the method "raiseBeginEvent" of the ECAAgent class in Java LED raises the event. Once the event is raised by the application, the Java LED detects the primitive event.

It may be noted that no rule is specified on the primitive event. When a tuple is inserted on the table 'Weather', the triggers 't_addw', 't_addw01' are fired. The trigger 't_addw' contains the action portion of the user defined primitive event trigger. Since the trigger action is fired for every occurrence of the event, the need for condition check and then to fire action does not arise for primitive event detection. But in the case of composite events rules are defined on the composite event and are discussed in detail in chapter 6.

5.3 Triggers on Update of Columns

In DB2, triggers can be defined for update on table and also for update of columns on a table. Let us consider the situation where there are two triggers defined,

Trigger 1- after update of table

Trigger 2- after update of columns x, x+1 on table

Case1: update of table for some column which is neither 'x' nor 'x+1'

Trigger 1 gets fired

Case2: update of column 'x' or x+1on table

Trigger 1 gets fired

Trigger 2 gets fired

Case3: update of columns x and x+1

Trigger 1 gets fired

Trigger 2 gets fired

From case 2 and case 3 it is apparent that, 'Trigger 2'which is defined for update of columns on a table, is fired as long as at least one of the columns for which it is defined is updated. The other trigger ‘Trigger 1’ is fired for an update of any column on the table. Trigger 2 will not fire if the update is on a column other than the columns it is defined on. Since both triggers are not fired for any update, ‘update on table’ and ‘update of columns x, x+1 on table’ need to be considered as two different events.

As a result the user (client) can define primitive event triggers for update of columns on a table, update on table. If there is a primitive event defined for update of columns on a table, then the attribute ‘columnNames’ in table SysPrimitivEvent will contain those column names [16] when the values are persisted. The column names are used in processing Repeat Primitive Event Triggers.

5.4 Before Triggers

In DB2, before triggers are always row triggers. Before triggers are used to ‘condition’ data values before they are entered into the database by an INSERT or UPDATE statement. Before triggers in DB2 cannot access the transition tables ‘OLD_TABLE’, ‘NEW_TABLE’.

The primitive event is defined by extending the native trigger syntax of DB2. If a primitive event is defined by extending the before trigger syntax for an insert/delete/update operation, the primitive event parser creates two triggers ‘triggerName’ and ‘triggerName01’. The trigger ‘t_triggerName01’, which contains SQL statements to modify the Mediator system tables, is always a statement level trigger. This trigger needs to access the tables ‘OLD_TABLE’, ‘NEW_TABLE’, so that tuples can be inserted into ‘R_inserted/R_deleted’ tables. But this is not possible. Hence primitive event definition using before trigger syntax is not allowed for the construction of composite events.

5.5 Repeat Primitive Event Triggers

DB2 supports multiple triggers for an operation on a table. This provision of DB2 is used by the Mediator to allow specification of additional actions (triggers) for an already existing primitive event. A repeat primitive event trigger can be specified as shown in figure 5.8. The syntax is slightly different from the primitive event trigger syntax in that the user does not have to specify the operation and table name for the trigger.

```
create trigger t1_addw event addWeather
referencing new as newrow for each row mode DB2Sql

//sql statements
.
.
;end
```

Figure 5.8. Repeat Primitive Event example.

The trigger 't1_addw' uses the event that is already defined in the system. The primitive event trigger shown in figure 5.1 defines the event *addWeather*. The action portion for the first primitive event is the trigger 't_addw' shown in figure 5.3. By specifying the repeat primitive event trigger shown in figure 5.8, the user specifies additional actions for the same event. When the user inputs a repeat primitive event trigger, the Language Filter sends the input to the Repeat Primitive Event Parser. The Repeat Primitive Event Parser is as shown in figure 5.9.

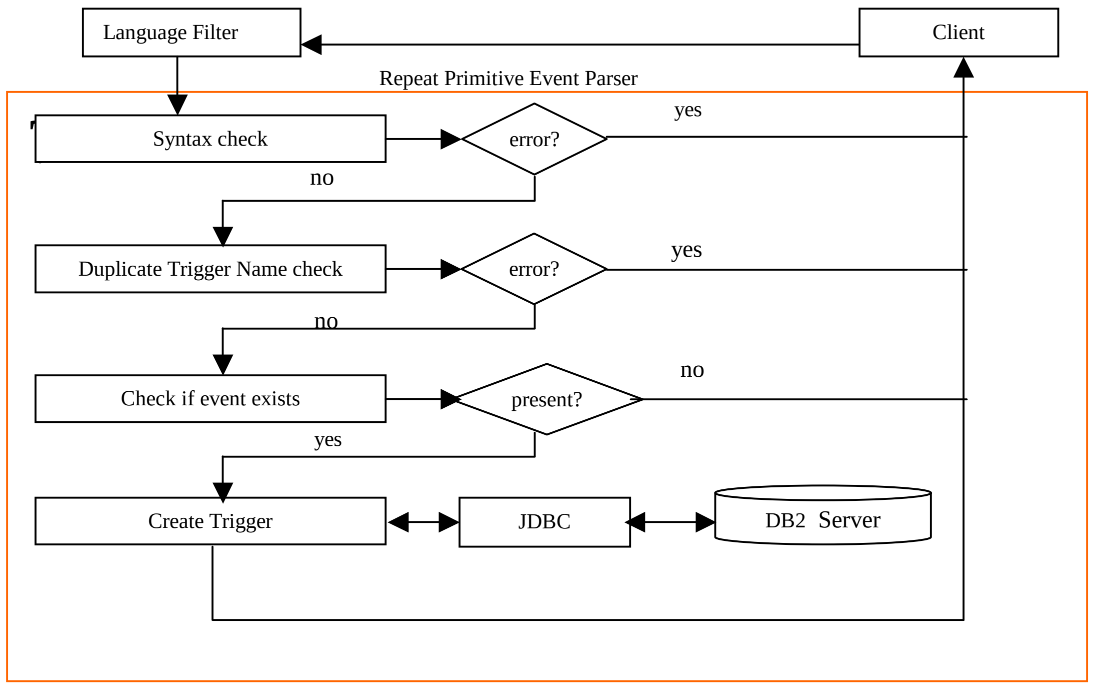


Figure 5.9. Repeat Primitive Event Parser.

5.5.1 Repeat Primitive Event Parser

Like the Primitive Event Parser, this parser does syntax checking, duplicate trigger name checking. The parser also checks if the event name *addWeather* that is re-used is already defined in the System. Checking the SysPrimitiveEvent table for the presence of the event does this. If the event is not present, the parser returns an error to the client. If there are no errors, the parser creates a trigger ‘t1_addw’ that contains the action portion of the repeat primitive event trigger. The parser gets the operation (insert/delete/update) and table name from SysPrimitiveEvent table for the event *addWeather* and generates the trigger ‘t1_addw’ as shown in figure 5.10. If the operation is ‘update of columns’ then the parser also gets the column names from the SysPrimitiveEvent table.

```
create trigger t1_addw after insert on weather
referencing new as newrow for each row mode DB2Sql
 begin atomic
//sql statements
.
.

;end
```

Figure 5.10. Trigger t1_addw.

5.5.2 Store metadata

The parser persists the repeat primitive event through the Persistence Manager. This persistent information is useful in dropping events. Persistence Manager inserts the following tuple into the SysEcaTrigger table for the repeat primitive event trigger shown in table 5.9.

Table 5.9. SysEcaTrigger Values for Repeat Primitive Event Example

dbName	userName	triggerName	triggerProc	Timestamp	eventName
ecadb2	nellai	t_addw	none	Current timestamp	addWeather
ecadb2	nellai	t_addw01	none	Current timestamp	addWeather
ecadb2	nellai	t1_addw	none	Current timestamp	addWeather

5.6 Drop Primitive Trigger

The Drop Trigger Module checks if the drop trigger statement is for a trigger defined on primitive event. It does this by checking the trigger information in the SysEcaTrigger table. If yes, it sends it to the Drop Primitive Event Trigger (DPT) sub-module.

Let the user input the following drop trigger statement.

drop trigger t_addw;

This statement is to drop the trigger 't_addw' defined on the primitive event *addWeather*. The DPT sub-module does a series of actions before it can drop the trigger. The actions are:

Check if constituent event of some composite event:

As we have discussed earlier, a primitive event cannot be dropped if it is the constituent event of a composite event. The table SysDrop has this information. The DPT module checks if the primitive event is present as a constituent event in the SysDrop table. If present in the SysDrop table, the trigger cannot be dropped. The error message is sent back to the client. If not present in the SysDrop table, the DPT module proceeds further.

Check for repeat primitive event trigger:

We have discussed earlier, that when the user defines a primitive event, the Mediator-Server creates two triggers, one for the Mediator-Server actions and the other for the user defined action portion. So when a primitive event has to be dropped, both these triggers have to be dropped. But there is a possibility that the user created a repeat primitive event trigger. In such a case, only the trigger containing the user defined action portion needs to be dropped. The other trigger containing the Mediator-Server actions should not be dropped since the repeat primitive event trigger uses it. The DPT module checks if there are repeat primitive event triggers for the event *addWeather*. Table 'SysEcaTrigger' contains the trigger names of all the triggers created by the Mediator application. If there are more than two triggers for the event in the table ‘SysEcaTrigger’, it means that there is a repeat primitive event trigger defined for the event *addWeather*. Hence the tuple for trigger ‘t_addw’ is deleted from table SysEcaTrigger but ‘t_addw01’ is not deleted. Also the primitive event tuple in SysPrimitiveEvent table is not delted, since the event is used by the repeat primitive event trigger.

```
delete from SysEcaTrigger where username= ‘nellai’ and triggerName= ‘t_addw’;
```

The trigger is then dropped from the DBMS

```
drop trigger ‘t_addw’
```

Delete Event node:

If there are no repeat primitive event triggers defined on the event *addWeather*, then the triggers ‘t_addw’, ‘t_addw01’ are deleted from the table SysEcaTrigger and the event tuple is deleted from SysPrimitive Event .

```
 delete from SysEcaTrigger where username= ‘nellai’ and eventName= ‘addWeather’;
 delete from SysPrimitiveEvent where  username = ‘nellai’ and eventName = ‘addWeather’;
```

The trigger ‘t_addw’ and also the trigger ‘t_addw01’ is then dropped from the DBMS.

```
  drop trigger ‘t_addw’;
  drop trigger ‘t_addw01’;
```

The ‘c_eventName.java’ and ‘c_eventName.class’ files used to register the primitive event are deleted from the user directory .

```
 String dir = username + "_" + database ;
 String fileName1 = ".\\metadata\\" + dir + "\\c_" + eventName +".java" ;
String fileName2 = ".\\metadata\\" + dir + "\\c_" + eventName +".class" ;
File obj=new File(fileName1);
 obj.delete();
 obj=new File(fileName2);
```

obj.delete();

Since the primitive event no longer needs to be detected, the event node is deleted from the Java LED by calling the following API

ECAAgent myAgent=ECAAgent.getAgentInstance("nellai_ecadb2");

myAgent.deleteEvent(eventname);

Summary:

This chapter discussed the implementation of primitive events. The issues like creation of separate triggers to perform Mediator-Server actions such as parameter collection, version number update, event notification and user-defined actions have been discussed. The chapter also discussed the implementation details of registering the primitive event with Java LED and raising the event so that Java LED detects it. The implementation details of repeat primitive event triggers and dropping of primitive events were also discussed.

CHAPTER 6

IMPLEMENTATION OF COMPOSITE EVENTS

Composite events are not supported in DB2 except for the update of columns on a table. If a trigger is created for the update of multiple columns on a table, the trigger will executed even if one of the columns is updated. This is analogous to OR operator in ECA paradigm. In order to enhance active capability of DB2, a wider spectrum of composite events need to be supported. In this thesis we support the composite events defined by Snoop using Java LED. This chapter discusses the implementation issues of composite events and also deals with dropping of composite events.

The general syntax of the composite event trigger is as shown in figure 4.3. For example, let us consider the composite event shown in figure 6.1. The Language Filter sends the trigger statement to the Composite Event Parser. This parser is as shown in figure 6.2. This parser does a series of steps before generating the API to register the event with Java LED.

```
create trigger t_andW  event andWeather = addwCalif ^ addwTexas:
recent     immediate    1
begin
           inset into WeatherNational select * from  WeatherCalif_inserted_tmp;
           insert into WeahterNational select * from WeatherTexas_inserted_tmp;
           insert into notifyUser values('andWeather occurred');
end
```

Figure 6.1. Composite event andWeather.

6.1 Composite Event Parser

Like the primitive event parser, this parser does syntax checking, duplicate trigger name checking and also duplicate event name checking. If there are no errors, the parser parses out the string containing parameter context, coupling mode and priority between the token ':' and keyword 'BEGIN' so that they can be used to generate the API to register the event with Java LED. If the parameter context, coupling mode and priority are not specified, the parser considers the default values. The default values are,

Context – Recent

Coupling mode – Immediate

Priority – 1

If the composite event has to be registered with Java LED, then the composite event API provided by the Java LED needs to be called. We have discussed earlier in the Design chapter the advantages of using Snoop Preprocessor to parse the composite event expression. Hence the Composite Event Parser sends the composite event definition string `event andWeather = addwCalif^addwTexas` and the rule definition string `rule Rule_andWeather[andWeather,c_andWeather.True,c_andWeather.andWeatherAction,1,IMMEDIATE,Recent]` to the Snoop Preprocessor. If there are no errors, the Snoop Preprocessor generates the composite event API.

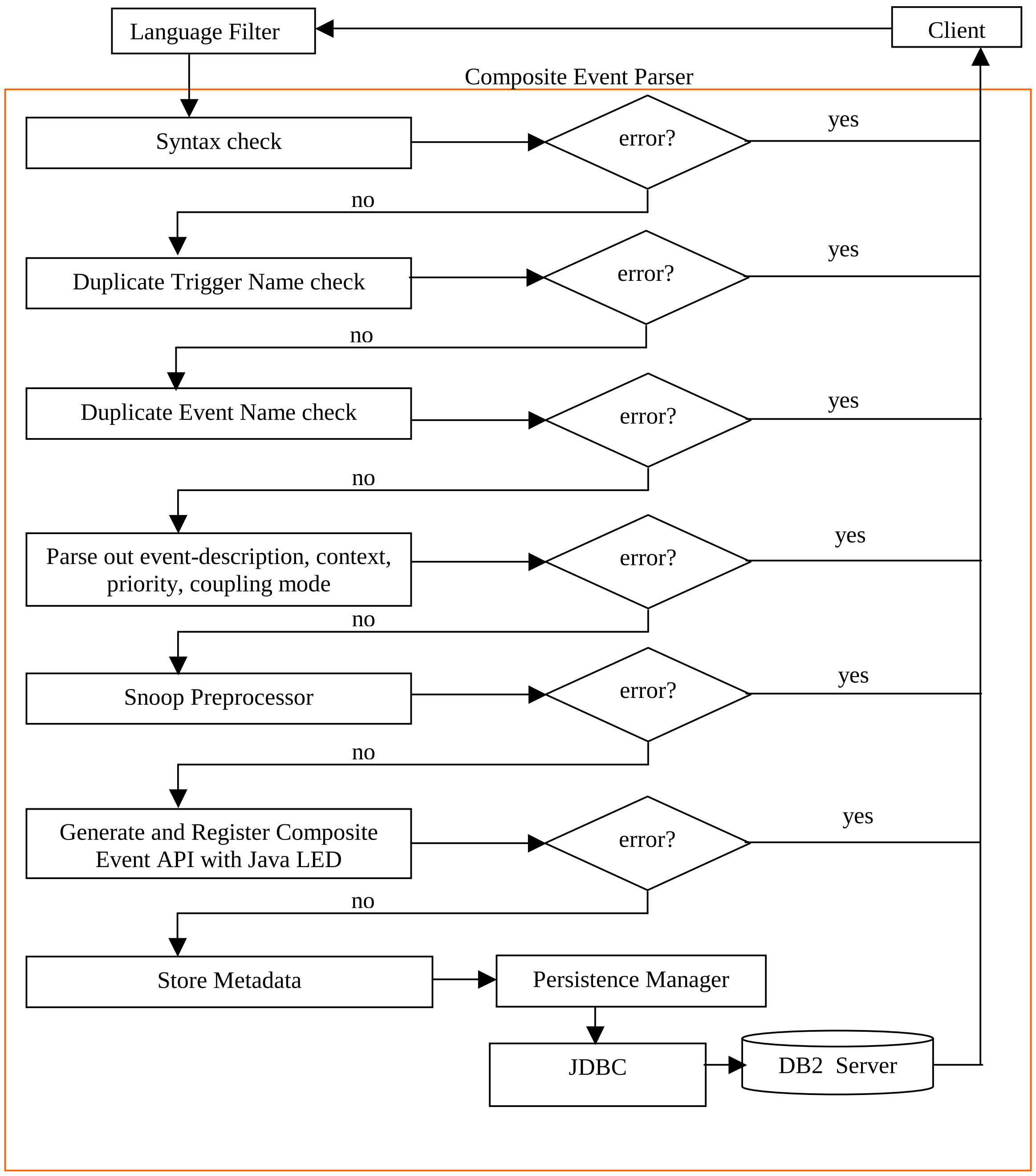


Figure 6.2. Composite Event Parser.

6.1.1 Interface with Snoop Preprocessor

The input to the Snoop Preprocessor is a file named *c_eventName.sjava* containing the event definition string and rule definition string expressed in Snoop. The processor scans the input file for the event definition string, rule definition string and generates the corresponding composite event API. For the example shown in figure 6.1, the parser generates a file *c_andWeather.sjava* that contains the event definition string and rule definition string and is shown in figure 6.3.

```
 ECAAgent myAgent = ECAAgent.initializeECAAgent("nellai_ecadb2");
 event andWeather = addwCalif^addwTexas;
rule
Rule_andWeather[andWeather,c_andWeather.True,c_andWeather.andWeatherAction,1,IMMEDIATE,R
ecent];
}
```

Figure 6.3. c_andWeather.sjava

In the figure, the shaded text represents the event definition string and rule definition string, which the Snoop Preprocessor will process and then generate the composite event API if there are no errors. The Snoop Preprocessor generates two output files *c_eventName.txt* and *c_eventName.java*. The file *c_eventName.txt* contains a list of the constituent events of the composite event that is used to check if the constituent event names are defined in the system. A composite event can be registered and detected only if the constituent events are defined in the system. If the constituent events are not defined in the system, then a composite event cannot be registered. Also the constituent event names are used to find out the table names on which the constituent events are defined so that parameter context can be supported in the action portion of the composite event. The parser first checks the table SysPrimitiveEvent for the presence of the events. There is a possibility that a constituent event can be a composite event. Hence, if a constituent event is not a primitive event, the parser checks if it is a composite event by checking the SysCompositeEvent table. For the example shown in figure 6.1, the file *c_andWeather.txt* is as shown in figure 6.4.

```
addwCalif
addwTexas
```

Figure 6.4. c_andWeather.txt.

The second file *c_eventName.java* generated by the Snoop Preprocessor contains the composite event API and rule API to register the event and rule with Java LED. For the example shown in figure 6.1, the file *c_andWeather.java*, is as shown in figure 6.5.

```
.
.
.


ECAAgent myAgent = ECAAgent.initializeECAAgent ( "nellai_ecadb2" ) ;

// event andWeather = addwCalif ^ addwTexas;
EventHandle andWeather = myAgent.createCompositeEvent(EventType.AND, "andWeather"

,c_addwCalif.addwCalif,c_addwTexas.addwTexas);


        //rule Rule_andWeather [

//andWeather,c_andWeather.True,c_andWeather.andWeatherAction,1,IMMEDIATE,RECEN

T];

myAgent.createRule("Rule_andWeather",andWeather, "c_andWeather.True",

"c_andWeather.andWeatherAction", 1, CouplingMode.IMMEDIATE,

ParamContext.RECENT);

}
}
```

Figure 6.5. c_andWeather.java generated by Snoop Preprocessor

6.1.2 Generate the Action Method

As already mentioned in the design issues chapter, when a composite event is detected the parameter lists of the constituent events need to be accessed and the tuples containing tablename, context and version number for each of the constituent events need to be inserted into table SysContext. Further, the table SysContext may contain tuples to get parameters from an earlier detection of the composite event in a different context. Hence the SysContext table needs to be cleared of old tuples before inserting tuples in the present detection. The table ‘R_inserted’/ ‘R_deleted’ has all the tuples that were inserted/deleted for the different occurrences of the constituent events. In order to get the parameter context, we do a join of the tables SysContext and R_inserted/R_deleted and insert the results into the table ‘R_inserted_tmp’/ ‘R_deleted_tmp’. The sequence of actions that need to be performed in the composite event action portion is as shown below.

```
  delete from SysContext;
 for each constituent event
{
      get the parameter list from Java LED;
      insert into SysContext values(tableName, context, vNo);
}
for(each of the constituent events)
{
      delete from R_inserted_tmp;
      insert into R_inserted_tmp select * from R_inserted, SysContext where
      SysContext.context=‘eventContext' and SysContext.tableName=‘R' and
      R_inserted.vNo=SysContext.vNo";

      delete from R_deleted_tmp;
      insert into R_deleted_tmp select * from R_deleted, SysContext where
      SysContext.context=‘eventContext’' and SysContext.tableName=‘R’' and
      R_deleted.vNo=SysContext.vNo";
}
{
/user defined action
}
```

Java LED provides API to access the parameter lists of the constituent events. The parser generates an action method that contains the Java LED API and SQL statements to perform the above-mentioned actions. The generic name of the action method generated by the parser is eventName+ ‘Action’. For the example shown in figure 6.1, the action method is *andWeatherAction.*

6.1.3 Register the Composite Event

The user dynamically defines the composite events; hence the parser has to register the events dynamically. The parser extends the file *c_eventName.java* to contain the action method. Now the file *c_eventName.java* contains the composite event API, condition method and action method. The file *c_andWeather.java* for the event *andWeather* is given in Appendix. This java file needs to be compiled dynamically. The code to compile this java file is the same as shown in the implementation of primitive events. After the file is compiled, API needs to be called for the event to be registered with the Java LED. The compiled class *c_andWeather.class* is loaded into the memory and the method *call_andWeather()* in the loaded class ‘c_andWeather.class’ is invoked to register the event. Unlike the primitive event implementation, the parameters for the method “call_andWeather” are not null. The SQL statements in the action portion of the composite event are executed through JDBC calls. Hence, the database name, url, username and password are passed to the method “call_andWeather” before invoking it.

```
/*list of formal parameters*/
   Class[] paramTypes = new Class[4];
    paramTypes[0] = (new String()).getClass();
    paramTypes[1] = (new String()).getClass();
    paramTypes[2] = (new String()).getClass();
    paramTypes[3] = (new String()).getClass();
/*list of actual parameters*/
    Object[] params = new Object[4];
    params[0] = rdbms;
```

```
params[1] = url;
params[2] = username;
params[3] = password;
```

Once the method “call_addWeather()” is invoked the composite event ‘andWeather’ is registered with the Java LED.

6.1.4 Difficulty in Supporting Stored Procedure in the User Defined Action:

The composite event trigger syntax shown in figure 4.3 allows the user to specify the user-defined actions of the composite event as a stored procedure. At present the user defined actions are SQL statements and are executed as separate JDBC calls. By specifying a stored procedure, the user has the option of executing the actions as a single transaction. In DB2 6.1, a stored procedure consists of two parts, the stored procedure itself which runs on the server machine, and the client program that runs on the client machine. A stored procedure can be written in any of the programming languages supported by DB2 6.1:C, C++, Java. The stored procedure must be compiled and installed on the DB2 server machine. A stored procedure cannot be created from a DB2 client. The stored procedure can only be invoked from the DB2 client by means of a CALL statement. In our implementation the Mediator-Server sits between the server and the client(user) and services the user request. If the Mediator-Server is running on a different machine other than the DB2 server machine, it will not be possible for the Mediator-Server to write a stored procedure and compile it on the DB2 server. This is because the Mediator-Server will not have administrator privileges to install code on the DB2 server machine. But in Oracle and Sybase, a SQL stored procedure can be created from the RDBMS client. Hence a stored procedure for the user defined action portion of a composite event is not supported in this implementation for DB2 6.1 while they are supported for similar implementations in Oracle and Sybase.

6.1.5 Store Metadata

Once the event has been registered, the parser persists the composite event through the Persistence Manager. For the composite event trigger shown in figure 6.1, following tuples are inserted by the Persistence Manager into the SysCompositeEvent and SysEcaTrigger tables.

Table 6.1. SysCompositeEvent Values for Example

dbName	userName	eventName	eventDescribe	timestamp	Coupling	context	priority
ecadb2	nellai	andWeather	addwTexas^ addwCalifornia	Current timestamp	immediate	recent	1

Table 6.2. SysEcaTrigger Values for Example

DbName	userName	triggerName	triggerProc	timestamp	eventName
ecadb2	nellai	t_andW	none	Current timestamp	andWeather

6.2 Drop Composite Trigger

The Drop Composite Trigger(DCT) sub-module in the Drop Trigger module takes care of dropping the composite event from the system. For example, let us consider the drop trigger statement shown below.

drop trigger t_andW;

Like the DPT sub-module, the DCT sub-module checks if the composite event is the constituent event of some other composite event. This is done by checking the SysDrop table. If it is a constituent event, then the event cannot be dropped. If the composite event is not a constituent event, then the DCT sub-module deletes the event information from the system tables (SysEcaTrigger, SysCompositeEvent).

delete from SysEcaTrigger where username= ‘nellai’ and triggerName= ‘t_andW’;

delete from SysCompositeEvent where username = ‘nellai’ and eventName = ‘andWeather’;

When a composite event is defined, the Composite Event Parser enters information about the composite event and it’s constituent events in the table SysDrop. For the composite event *andWeather*, the SysDrop table will have the values as shown in table 6.3.

Table 6.3. SysDrop Values for Example

consEventName	context	compEventName
addwTexas	recent	andWeather
addwCalif	recent	andWeather

The primitive events *addwTexas* and *addwCalif* can be dropped only if the composite event *andWeather* is dropped. Hence when a composite event is dropped, the composite event information tuple is deleted from the table SysDrop, so that the constituent events can be dropped if the user decides to drop them.

delete from SysDrop where compEventName= ‘andWeather’;

Finally, the *c_eventName.java ,c_eventName.class, c_eventName.sjava, c_eventName.txt* files used to register the composite event are deleted from the user directory .

```
String dir = username + "_" + database ;
String fileName1 = ".\\metadata\\" + dir + "\\c_" + eventName +".java" ;
String fileName2 = ".\\metadata\\" + dir + "\\c_" + eventName +".class" ;
String fileName3 = ".\\metadata\\" + dir + "\\c_" + eventName +".sjava" ;
String fileName4 = ".\\metadata\\" + dir + "\\c_" + eventName +".txt" ;
File obj=new File(fileName1);
```

```
obj.delete();
obj=new File(fileName2);
obj.delete();
obj=new File(fileName3);
obj.delete();
obj=new File(fileName4);
obj.delete();
```

Since the composite event no longer needs to be detected, the event node is deleted from the Java LED. Before deleting the event node, the rule associated with the event node is deleted. This is done by calling the following API.

```
ECAAgent myAgent=ECAAgent.getAgentInstance(“nellai_ecadb2”);
myAgent.deleteRule(ruleName);
myAgent.deleteEvent(eventname);
```

Summary:

This chapter discussed the implementation of composite events. The implementation details of interfacing with the Snoop Preprocessor to parse the composite event expression and to generate the rule API were discussed. The chapter also discussed the implementation details of registering the composite event with Java LED. Furthermore, implementation details of supporting parameter context were also discussed.

CHAPTER 7

CONCLUSIONS

In this thesis, we have explained the design and implementation of a Mediator-based approach to support ECA rules in DB2. The Mediator provides ECA functionality whose level of abstraction is useful for many real world application domains and business rules. The creation of events (primitive, composite) and rules by specifying triggers has also been explained. The persistence of the events and rules using the underlying RDBMS has also been explained. The contributions of this thesis can be summarized as follows:

- Support primitive events.
- Support repeat primitive events.
- Support dropping of primitive events.
- Support most of the composite events as expressed in the event specification language Snoop. The composite events supported are And, Or, Sequence, Not, Aperiodic, Aperiodic*.
- Support dropping of composite events and rules.
- Persist the events and rules using the underlying RDBMS.
- Provide transparency to the user.
- Preserve the multi-user/multi-database feature of the underlying RDBMS.

7.1 Future Work

- The action portion of the composite event contains SQL statements that are used to support parameter context. The action portion also contains SQL statements that perform the user-defined action for the composite event. The SQL statements are executed as individual transactions by making JDBC calls. When two or more composite events containing rules with identical priorities are detected, the execution semantics of the SQL statements of the respective action portions are not clearly defined. This results in errors in getting the correct parameter context. Hence there is need to define execution semantics for two or more events with rules of same priority.

- In this thesis the Mediator is external to underlying RDBMS. Hence the active capability that can be provided is dependent on the Mediator architecture. The Mediator architecture can be extended to connect to multiple RDBMS platforms. A single Mediator can be used as the bridge between any RDBMS client and RDBMS server as shown in figure 7.1.

- The Mediator interfaces with the Java Local Event Detector (Java LED) to detect events. The actions for the events that are detected are sent to the respective clients that created the events. There are some real world application domains where the actions need to be sent to clients that are just monitor clients over a distributed environment. These monitor clients subscribe to events created by other clients. This involves changes to the Mediator architecture. There is also need to interface with the Java Global Event Detector (Java GED) to detect events over a distributed environment.

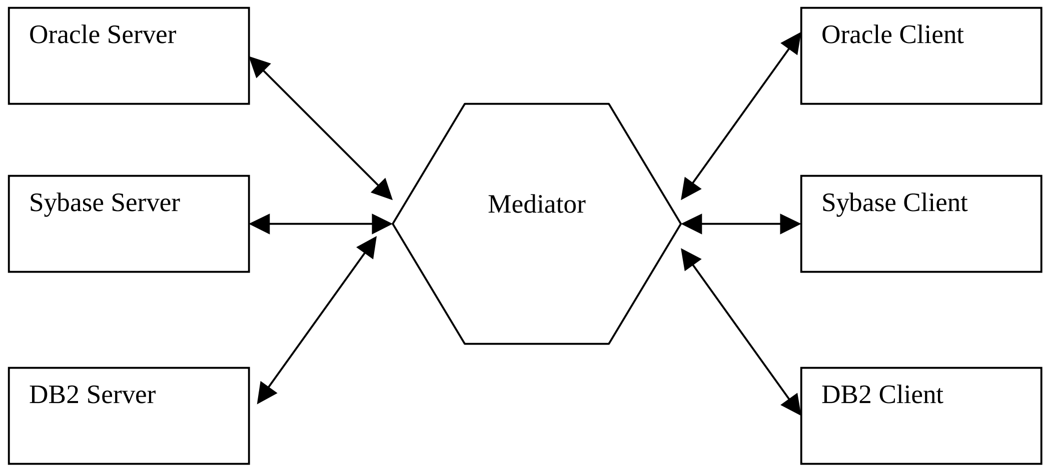


Figure 7.1. Multi-RDBMS.

REFERENCES

1. Widom, J. and C. St, *Introduction to Active Database Systems*, in *Active Database Systems - Triggers and Rules For Advanced Database Processing*. 1996, Morgan Kaufman Publishers Inc. p. 1--42.

2. Chakravarthy, U.S., *{Rule management and Evaluation: An Active DBMS Perspective}.* Special issue of ACM Sigmod Record on rule processing in databases, 1989. **18**(3): p. 20--28.

3. Hanson, E.N., *Rule Condition Testing and Action Execution in {A}riel*, in *Proc 1992 ACM-SIGMOD Conf. on Management of Data*. 1992: San Diego, California.

4. Chakravarthy, S., et al., *Design of Sentinel: An Object-Oriented {DBMS} with Event-Based Rules.* Information and Software Technology, 1994. **36**(9): p. 559--568.

5. Widom, J., R.J. Cochrane, and B.G. Lindsay, *Implementing Set-Oriented Production Rules as an Extension to Starburst*, in *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*. 1991: Atlantic City.

6. Stonebraker, M. and G. Kemnitz, *The Postgres Next-Generation Database Management System.* Communications of the ACM, 1991. **34**(10): p. 78--92.

7. Stella, G. and K.R. Dittrich, *SAMOS: An Active Object-Oriented Database System.* IEEE Quarterly Bulletin on Data Engineering, 1992. **15**(1-4): p. 23--26.

8. Li, L. and S. Chakravarthy. *An Agent-Based Approach to Extending the Native Active Capability of Relational Database Systems*. in *ICDE*. 1999. Australia: IEEE.

9. Song, Z., *A Generalized Approach For Extending The Active Capability Of RDBMSs*, in *Database Systems R&D Center, CISE Department*. 2000, University of Florida: Gainesville.

10. Krishnaprasad, V., *Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation*, in *MS Thesis*. 1994, Database Systems R&D Center, CIS Department, University of Florida: Gainesville.

11. Hyesun, L., *Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing*. 1996, University of Florida: Gainesville.

12. Chakravarthy, S., et al., *HiPAC: A research project in active, time-constrained database management*. 1989, Tech. Report (89-02), Xerox Advanced Information Technology: Cambridge.

13. CHAMBERLIN, D., *A Complete Guide to DB2, Universal Database*. 1998: Morgan Kaufmann.

14. Chakravarthy, S. and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases*. Data and Knowledge Engineering, 1994. **14**(10): p. 1--26.

15. Dasari, R., *Design and Implementation of a Local Event Detector in Java*, in *CISE*. 1999, University of Florida: Gainesville.

16. Mysore Ganesha Rao, Y., *An Agent based approach for extending the Trigger capability of Oracle*, in *ITLAB, CSE department*. 2002, The University of Texas at Arlington: Arlington.

APPENDIX

FILE GENERATED BY THE COMPOSITE EVENT PARSER TO REGISTER THE COMPOSITE EVENT WITH JAVA LED

```
import Server.src.* ;
import sentinel.led.* ;
import java.util.Vector ;
import java.util.Hashtable ;
import java.util.Enumeration ;
public class c_andWeather
{
static String rdbms = "" ;
static String url = "" ;
static String username = "" ;
static String password = "" ;
public static EventHandle andWeather = null ;
public static void call_andWeather( String Prdbms , String lru , String Pusername , String Ppassword )
{
rdbms = Prdbms ;
url = lru ;
username = Pusername ;
password = Ppassword ;
ECAAgent myAgent = ECAAgent.initializeECAAgent ( "testlogin_test1" ) ;

// event andw = addwc ^ addwt;
EventHandle andw = myAgent.createCompositeEvent(EventType.AND, "andw"
,c_addwTexas.addwTexas ,c_addwCalif.addwCalif);


// rule Rule_andw [ andw , c_andw.True , c_andw.andwtestlogin , 1 , IMMEDIATE , RECENT ];
myAgent.createRule("Rule_andw",andWeather, "c_andWeather.True",
"c_andWeather.andWeatherAction", 1, CouplingMode.IMMEDIATE, ParamContext.RECENT);
}
```

….contd.

c_andWeather.java

```
public static boolean True ( ListOfParameterLists parameterLists )
{
System.out.println ( "*****FromCondition*****" ) ;
return true ;
}

 public static void andwtestlogin(ListOfParameterLists paramLists)
 {
 Enumeration en = paramLists.elements();
Vector sysContext = new Vector(10,2);
 String ctc = "delete from SYSCONTEXT";
 Jdbc ctrigger = new Jdbc(rdbms,url,username,password,ctc);
 ctrigger.ExecuteSqlUpdate("delete SysContext ");
 int c = 0;
  while (en.hasMoreElements()) {
     ParameterList pl = ((ParameterList)en.nextElement());
     try
        {
           System.out.println("inside .....");
           System.out.println(++c);
          String eventname = pl.getObject("eventname").toString();
          String tablename = pl.getObject("tablename").toString();
           int vno = ((Integer)pl.getObject("vno")).intValue();
           sysContext.addElement(eventname);
           sysContext.addElement(tablename);
           sysContext.addElement(new Integer(vno));
           ctc = "insert into SYSCONTEXT values('" + tablename + "','RECENT'," + vno + ")";
           ctrigger = new Jdbc(rdbms,url,username,password,ctc);
           ctrigger.ExecuteSqlUpdate("insert into table SYSCONTEXT ");
        }
        catch(TypeMismatchException e)
         {
             System.out.println("Error in get Object in checking Parameter");
         }
         catch(ParameterNotFoundException ee) {System.out.println("Error Parameter");}
    }
 String info = "";
```

…..contd.

c_andWeather.java

```
  for (int i=0;i<sysContext.size();i++)
     info = info + " " + sysContext.elementAt(i).toString();

  String spc0 = "delete from WEATHERCALIF_inserted_tmp";
 Jdbc storedProCom0 = new Jdbc(rdbms,url,username,password, spc0);
 storedProCom0.ExecuteSqlUpdate("delete from WEATHERCALIF_inserted_tmp");
 spc0 = "insert into WEATHERCALIF_inserted_tmp select * from WEATHERCALIF_inserted,
SYSCONTEXT where SYSCONTEXT.CONTEXT='RECENT' and
SYSCONTEXT.TABLENAME='WEATHERCALIF' and
WEATHERCALIF_inserted.VNO=SYSCONTEXT.VNO";
storedProCom0 = new Jdbc(rdbms,url,username,password, spc0);
 storedProCom0.ExecuteSqlUpdate("insert into WEATHERCALIF_inserted_tmp");

spc0 = "delete from WEATHERCALIF_deleted_tmp";
storedProCom0 = new Jdbc(rdbms,url,username,password, spc0);
storedProCom0.ExecuteSqlUpdate("delete from WEATHERCALIF_deleted_tmp");
spc0 = "insert into WEATHERCALIF_deleted_tmp select * from WEATHERCALIF_deleted,
SYSCONTEXT where SYSCONTEXT.CONTEXT='RECENT' and
SYSCONTEXT.TABLENAME='WEATHERCALIF' and
WEATHERCALIF_deleted.VNO=SYSCONTEXT.VNO";
storedProCom0 = new Jdbc(rdbms,url,username,password, spc0);
storedProCom0.ExecuteSqlUpdate("insert into WEATHERCALIF_deleted_tmp");

String spc1 = "delete from WEATHERTEXAS_inserted_tmp";
Jdbc storedProCom1 = new Jdbc(rdbms,url,username,password, spc1);
storedProCom1.ExecuteSqlUpdate("delete from WEATHERTEXAS_inserted_tmp");
spc1 = "insert into WEATHERTEXAS_inserted_tmp select * from WEATHERTEXAS_inserted,
SYSCONTEXT where SYSCONTEXT.CONTEXT='RECENT' and
SYSCONTEXT.TABLENAME='WEATHERTEXAS' and
WEATHERTEXAS_inserted.VNO=SYSCONTEXT.VNO";
 storedProCom1 = new Jdbc(rdbms,url,username,password, spc1);
 storedProCom1.ExecuteSqlUpdate("insert into WEATHERTEXAS_inserted_tmp");

spc1 = "delete from WEATHERTEXAS_deleted_tmp";
storedProCom1 = new Jdbc(rdbms,url,username,password, spc1);
storedProCom1.ExecuteSqlUpdate("delete from WEATHERTEXAS_deleted_tmp");
spc1 = "insert into WEATHERTEXAS_deleted_tmp select * from
WEATHERTEXAS_deleted, SYSCONTEXT where SYSCONTEXT.CONTEXT='RECENT' and
SYSCONTEXT.TABLENAME='WEATHERTEXAS' and
WEATHERTEXAS_deleted.VNO=SYSCONTEXT.VNO";
storedProCom1 = new Jdbc(rdbms,url,username,password, spc1);
storedProCom1.ExecuteSqlUpdate("insert into WEATHERTEXAS_deleted_tmp");
…..contd.
```

c_andWeather.java

```
// user defined action function
     String spc = "insert into WEATHERNATIONAL select city,time,tem,wspeed from
WEATHERTEXAS_inserted_tmp;insert into WEATHERNATIONAL select city,time,tem,wspeed from
WEATHERCALIF_inserted_tmp";
     Jdbc storedProCom = new Jdbc(rdbms,url,username,password, spc);
     storedProCom.ExecuteSqlBatch( " Action Portion ");
     //System.out.println("testing....");
     String spcn = "insert into NOTIFYCOM values('andw','" + info + "')";
     Jdbc storedProComn = new Jdbc(rdbms,url,username,password, spcn);
     storedProComn.ExecuteSqlBatch( " update notifyCom ");
     System.out.println ("****From Composite Event andw Action of Rule****");
    }}
```

c_andWeather.java