MAKING SYBASE FULLY ACTIVE: SUPPORTING COMPOSITE EVENTS AND PRIORITIZED RULES

by

GANESH AMBALAVANAN GOPALAKRISHNAN

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

August 2002

To My Parents

ACKNOWLEDGMENTS

First and foremost, I would like to thank my supervising professor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on naval research project and providing me great guidance and support throughout this research work.

I would like to thank Dr. Alp Aslandogan and Dr. Mohan Kumar for serving on my committee.

I am grateful to Nellainayagam, Arvind and Raman 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 during this work.

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 sister for their endless love and constant support throughout my academic career.

July 11, 2002

ABSTRACT

MAKING SYBASE FULLY ACTIVE: SUPPORTING COMPOSITE EVENTS AND PRIORITIZED RULES

Publication No.__________

Ganesh Ambalavanan Gopalakrishnan

The University of Texas at Arlington, 2002

Supervising Professor: Sharma Chakravarthy

A database management system (DBMS) is a software system for reliably and efficiently creating, maintaining, and operating large information repositories that can be accessed concurrently by a large number of users. Commercial RDBMSs used to be passive earlier, but currently support limited active capability. Active database management system is one that extends the passive DBMS with the ability to react to set of pre-defined situations or conditions (induced by the changes applied to the database). Event Condition and Action (ECA) rules are typically used to enhance the native active capability present in the RDBMS.

The goal of this thesis is to generalize the mediated approach to support active capability. Introducing a generalized ECA agent between the Database Server and the client permits us to enhance the native active capability of RDBMSs. The architecture, design, and implementation of this approach are explained in this thesis. The system developed supports primitive and composite events in different contexts. The mediated approach used in this thesis is in contrast to the integrated approach for supporting active capability. This approach is useful when it is not possible to change or add code to the underlying system.

This approach provides an alternative way to support complete ECA paradigm using a mediated approach by using the “trigger” capability available in the underlying DBMS.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Active Database Management Systems (ADBMSs) are those that are capable of monitoring and reacting to specific situations of relevance to an application. In commercial RDBMSs queries are executed as and when the user or the client application requests them. That is, it uses a query driven mechanism. Commercial RDBMSs also support triggers that provide a limited form of situation monitoring. This research addresses the mechanisms for providing complete active capability for RDBMSs and uses Sybase for implementing our approach.

1.1 ECA paradigm

An example in banking scenario can be taken to emphasize the need for ADBMS. Consider a situation in which a customer has a savings account and the minimum balance in savings account is five hundred dollars. For every transaction of this customer the minimum balance has to be checked, and if it goes below five hundred dollars, the banker and the customer has to be notified to take appropriate action. In this example, every transaction is an event, checking of the balance is the condition and the notification is the action. This is collectively called as ECA paradigm. This whole information is typically stored in a database. The above said action can be done through triggers in commercial RDBMSs but with restrictions. In this work we address the limitations of Sybase triggers and provide a generalized way to enhance the active capability for RDBMSs.

1.2 Issues

The development of an active database needs to consider the following issues [14]:

- An active database system must provide all the usual functionality of a conventional passive database system. Meanwhile, it is desirable that the performance of conventional database tasks is not penalized by the fact that the database system is active.

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

1.3 Triggers

Database triggers, sometimes referred to as alerts or monitors, specify that certain actions should be invoked whenever certain conditions are detected. For example, an inventory control trigger might detect when stock is low and automatically reorder items. Active database systems support triggers and in fact are based on the trigger concept [14].

In the context of Sybase, trigger is a stored procedure that goes into effect when you insert or delete or update data in a table. The main advantage of triggers is that they are automatically executed when the associated event (insert, delete or update) happens. The limitation in Sybase is that it allows only one trigger per operation on a table. It does not support before and after triggers as well.

1.4 ADBMS

There are lots of Active Database Management Systems (ADBMS) that come under different categories. They are, (Extended) Relational Active Databases such as, Ariel [8], Postgres [4], Starbust [14]; Object Oriented active databases such as, HiPac [5], Sentinel [10], EXACT [7].

Active database systems enhance traditional database functionality with powerful rule processing (or "trigger") capabilities. Active database systems are significantly more powerful than their passive counterparts in the following aspects [14]:

- Active database systems can efficiently perform functions that in passive database systems must be encoded in application. . It is not the efficiency as much but the ability to do it outside of the application that is important.
- Active database systems can support applications that are beyond the scope of passive database systems (e.g., situation monitoring)
- Active database systems can perform tasks that require special-purpose subsystems in passive database systems.

Most of the work on ADBMS is based on the integrated approach that requires access to the underlying source code of the DBMS. Integrated approach cannot be used for commercial RDBMSs, as access to the underlying source code cannot be obtained. A mediator based ECA Agent was implemented by Lijuan Li [15] in order to enhance the active capability provided by Sybase. This work used the Gateway Open Server present in Sybase to provide ECA functionality. Hence this implementation was not portable to other commercial RDBMSs and it was platform dependent. The next step was a Generalized ECA Agent that was implemented by Zecong Song [19]. 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 [9].

With this as the background, in this research work we use the Generalized ECA Agent to overcome the above deficiencies in the previous work and use Sybase as the example RDBMS. 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 all 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; the multi-user/multi-database capability of the underlying DBMS is preserved.

This chapter gave an introduction about the active technology, active database systems and triggers. It also discussed about the ECA paradigm and the issues involved in the active database systems.

2. RELATED WORK

The work on active capability has been ongoing for about two decades. There have been a number of research projects that have addresses various aspects of active database research and prototype development. In this chapter, we present a summary of a few of these research prototypes.

Ariel

The Ariel system implements active capability in a RDBMS by having a built-in rule system. The rule system of Ariel is based on the production system model. The approach taken in Ariel has been to adopt as much as possible from previous work on main memory production systems such as OPS5, but make changes where necessary to tailor the performance of a production based system to a DBMS environment. The changes that have been made include a rule language extension to POSTQUEL [5] with a query language like syntax, a discrimination network for rule condition testing tailored to database environment and measures to integrate rule processing with set oriented database update commands.

Since Ariel is based on the relational data model, it provides a subset of the POSTQUEL query language of POSTGRES for specifying data definition commands, queries and updates. The rule language of Ariel is a production rule language with enhancements for defining rules with conditions that can contain relational selections and joins as well as specification of events and transitions. The syntax of a rule is similar to that of a query language. The general form is shown below.

```
Define rule rule-name [in rule set-name][priority priority-val]
[on event]
[if condition] then action
```

A unique rule name is required for each rule so that the user can refer to the rule later.

The on clause allows the specification of an event that will trigger the rule. Following types of events can be specified after an on clause:

append [to] relation-name

delete [from] relation name

replace [to] relation name [(attribute list)]

To summarize, following are the features of the Ariel system:

1. It is based on the production system model.
2. It is set oriented.
3. It is tightly integrated with the DBMS.
4. Provides condition-action binding based on shared tuple variables.
5. Supports event, transition, and ordinary conditions in a uniform way.
6. Uses a modified version of TREAT algorithm called A-TREAT algorithm for rule testing, which speeds up the entire process.

2.2 Sentinel

The Sentinel system [9,10,11] is a follow-on system to HIPAC [5]. 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 [10]. 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. Extension of the 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.

This system uses an integrated approach, which cannot be used for commercial RDBMSs. This is because in the integrated approach, the system will be developed as a part of the existing system. Hence needs access to source code. We cannot get the source of the commercial RDBMSs.

2.3 Ode

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

Ode provides two kinds of active facilities: “constraints” for maintaining database integrity and “triggers” for automatically performing actions depending upon the database state [6].

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

Ode trigger model is an event-action (E-A) model. When an event occurs, the associated action is executed. Ode supports primitive events and composite events. Primitive events are defined and composite events are constructed by applying operators to primitive events. The basic events that are supported are object state events. The event operators supported are prior, sequence, first, first after, happened, every, prefix and so on.

2.4 InfoBus

InfoBus [3] provides transparent data exchange between different components of a system. The components are built using Java Beans discussed above. The components that are interested to exchange information subscribe to an InfoBus channel as data producers and/or data consumers. As the names indicate, data is transferred from a data producer to a data consumer. The connecting components have to follow certain rules mandated by InfoBus in order to exchange data. Semantics of data flow are based on interpreting contents of the data that flows across the InfoBus interface. They are the components that establish the data flow between each other. A data consumer subscribes to the InfoBus interface by naming the data item that it is interested to receive.

Events are sent by the InfoBus to listeners for each component on the bus. Three types of events are defined:

1. InfoBusItemAvailableEvent—an event which is broadcast on behalf of a producer to let potential consumers know about the availability of a new data item through the InfoBus.

2. InfoBusItemRevokedEvent—an event that is broadcast on behalf of a producer to let consumers know that a previously available data item is no longer available.

3. InfoBusItemRequestedEvent—an event which is broadcast on behalf of a consumer to let producers know about the need for a particular data item that they may be able to supply.

Thus, it provides only data exchange between various components of a system based on subscription mechanism. It does not have the notion of composite events or rules. Its main purpose is to provide transparent and easy exchange of data between the Java Bean components of a system.

2.5 An Agent Based Approach

Lijuan Li implemented an Agent-Based approach to extending the native active capability of relational database systems [15] specifically for Sybase in C++. The implementation has an ECA Agent, which is a mediator between clients and Sybase SQL Server. Sybase Gateway Open Server was used in that approach to extend the active capability of Sybase. 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 Open Server (specific to Sybase) to provide ECA functionality. Also since it uses the C++ LED there are portability constraints to be considered—it can be used only for that version of Sybase on Unix Platform. Hence this approach is not a generalized approach that can be ported to other commercial RDBMSs such as DB2, Oracle.

Though this approach has many disadvantages and lack many issues, this formed the basis for this thesis to support multiple triggers and composite events in a generalized way that may help us to port the entire system to different platform and also to other RDBMSs.

2.6 A Generalized approach

A Generalized ECA Agent was implemented by Zecong Song [19] using IBM DB2 as test database and another developed by Kim [18] using Oracle as test database. These systems were follow-on systems to the Sybase ECA Agent. These systems were also based on the mediated approach. The functionality of the Generalized ECA Agent is similar to the Sybase ECA Agent. The 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 completely implemented in this prototype. Furthermore, dropping of events and rules were not supported.

This chapter discussed some of the earlier research on active databases. The discussion was on integrated approach such as Ariel and Sentinel. Also systems that use the mediator-based approach such as Sybase ECA Agent, Generalized ECA Agent were also discussed.

3. OVERVIEW

This chapter gives an overview of the concepts and systems used in this research work.

3.1 Primitive Event

An event is an occurrence of interest at a specific point in time. Primitive events are the elementary occurrences and are classified into database, temporal, and explicit events. Database events are associated with the manipulation of data such as the update, deletion, or insertion (on tables) and are executed over a period of time. Event modifiers (begin and end) were introduced to transform operations that take an interval into an event. In other words, the event modifiers (begin and end) are used to map the logical events at the conceptual level to physical events. The begin event modifier denotes the starting point of a database event and the end event modifier denotes the ending point. Temporal events correspond to absolute and relative events that are associated with time. The absolute temporal event is an event associated with an absolute value of time. For example, 4 P.M. on August 15, 1947 is an absolute event. The relative temporal event is an event corresponding to a specific point on the time line, which is an offset from another time point (specified either as absolute or as an event). Explicit (also termed abstract) events are explicitly defined in the application, but their occurrences are either detected outside of the application and conveyed to the application or the application explicitly raises those events.

3.2 Composite Events

A composite event is an event that is composed of primitive events and/or other composite events by applying Snoop [9] event operators such as OR, AND, SEQUENCE, NOT. In order words, the constituent events of the composite event can be primitive events and/or composite events.

3.3 Snoop Event Operators

The event operators are used to construct composite events. Each of these event operators and its semantics are described briefly in the following section. The upper case letter E, which represents an event type, is a function from the time domain on the Boolean values. The function [9] is given by

E (t) = True if an event type E occurs at time point t

False otherwise

Disjunction: OR (∇)

Disjunction of two events E_1 and E_2 denoted by $E_1 \nabla E_2$ is applied when either E_1 occurs or E_2 occurs. Formally,

$(E_1 \nabla E_2)(t) = E_1(t) \nabla E_2(t)$

Conjunction: AND (Δ)

Conjunction of two events E_1 and E_2, denoted by $E_1 \Delta E_2$ is applied when E_1 occurs and E_2 occurs in any arbitrary order. Formally,

$(E_1 \Delta E_2)(t) = (E_1(t1) \Delta E_2(t)) \nabla ((E_1(t) \Delta E_2(t1))$

and $t1 \leq t$

Sequence (;)

The sequence of two events E_1 and E_2, denoted by $E_1 ; E_2$ occurs when E_1 happens before E_2. The timestamp of occurrence of E_1 is less the timestamp of occurrence E_2. Formally,

$(E_1; E_1)(t) = E_1(t1) \Delta E_2(t)$ and $t1 < t$

Negation: NOT (¬)

The *not* operator, denoted by $\neg(E_2)[E_1, E_3]$ is applied when there is no occurrence of E_2 in the closed interval formed by E_1 and E_3. Formally,

$\neg(E_2)[E_1, E_3](t) = (E_1(t1) \Delta \sim E_2(t2) \Delta E_3(t))$

and $t1 \leq t2 \leq t$

Aperiodic Operators (A)

The non-cumulative aperiodic operator, denoted by A (E_1, E_2, E_3) is used to express the occurrence of an aperiodic event in the half-open interval formed by two arbitrary events. The A event occurs each time when E_2 occurs during the half-open interval defined $E_1$and E_3. The number of occurrence of A event is proportional to the occurrence of E_2 during the interval. Formally,

$A(E_1, E_2, E_3)(t) = E_1(t1) \Delta \sim E_3(t2) \Delta E_2(t))$

and $(t1 < t2 \leq t$ or $t1 \leq t2 < t)$

Aperiodic Star Operators (A*)

The cumulative aperiodic operator, denoted by A* (E_1, E_2, E_3) is similar to the non-cumulative aperiodic operator. It A* event is signaled only once when E_3 occurs rather than detected the event every time the event E_2 occurs. However, the occurrences of E_2 are accumulated each time when E_2 occurs during the half –open interval formed by E_1 and E_3. Formally,

$A^*(E_1, E_2, E_3)(t) = (E_1(t1) \Delta E_3(t))$ and $t1 < t$

Periodic Operator (P)

The periodic operator, denoted by P $(E_1, [t], E_3)$ is used to express a periodic event that repeats itself within a constant and finite amount of time. The event P is signaled for every amount of time t in the half-open interval $(E_1, E_3]$. Formally,

$P(E_1, [TI], E_3)(t) = (E_1(t1) \Delta \sim E_3(t2))$

and $t1 < t2$ and $t1 + x * TI = t$ for some $0 < x < t$ and $t2 \leq t$

where TI is a time specification.

Periodic Star Operator (P*)

The periodic star operator denoted by $P^*(E_1, [t], E_3)$ is a cumulative variant of P. The occurrence Similar to aperiodic star context, the P* event is signaled only once when E_3 occur. Formally,

$P^*(E_1, [TI], E_3)(t) = (E_1(t^1) \Delta E_3(t))$

and $t^1 \leq t$

Plus (+)

The plus operator denoted by E_1+ [T] is applied when T time units are elapsed after E_1 occurs.

3.4 Parameter Context

Four parameter contexts—recent, chronicle, continuous, and cumulative—were introduced to provide a mechanism for capturing meaningful application semantics and reduce the space and computation overhead for the detection of composite events using the semantics described above [9]. The contexts are defined by using the notions of initiator and terminator for events. An event that initiates the occurrence of a composite event is termed the initiator of the composite event. An event that completes the detection of a composite event is denoted as the terminator of the composite event. For example, a composite event (E1 Δ E2 Δ E3) has E1 as initiator and E3 as terminator.

Recent: In the recent context, only the most recent occurrence of the initiator (when there are multiple instances of the same event) for any event that has started the detection of that event is used. When the event occurs, all the occurrences of events, that are used in the parameter relation and cannot be initiators of that event in the future, are deleted. In this context, not all occurrences of a constituent event will be used in detecting a composite event. Furthermore, an initiator of an event will continue to initiate new event occurrences until a new initiator occurs. This recent context is suitable for events that happen at a fast rate and the multiple occurrences of the same event only refine the previous value.

Chronicle: In the chronicle context, the initiator-terminator pair is unique for an event occurrence. The oldest initiator is paired with the oldest terminator for each event. When event occurs, the occurrences of the events are deleted. The event occurrence can be used at most once for computing the parameters of the composite event. This context is useful when the different types of events have an established relationship between their occurrences.

Continuous: In the continuous context, each initiator of an event starts a separate detection of that event. A terminator event occurrence may detect one or more occurrences of the same event. The initiator and terminator are discarded after an event is detected. This context is fit for tracking trends of interest along a moving time window.

Cumulative: In the cumulative context, all occurrences of an event type are accumulated as instances of that event until the event is detected. When the event occurs, all the occurrences that are used for detecting are discarded.

3.5 Coupling Modes

In early systems such as POSTGRES [4], condition evaluation and action execution were done immediately after the event was detected. However, in some situations this is too restrictive. For integrity checks, condition evaluation and action execution need to be done at the end of a transaction before it commits. Coupling modes were introduced to specify a relative point in time where condition evaluation and action execution should take place after the event is detected, with the constraint that the action will be performed only when the condition is satisfied.

There are three coupling modes:

Immediate:

When an event is detected, the transaction is suspended, and the condition associated with the event is evaluated immediately. If the condition evaluates to true, the action is executed. The execution of the triggering transaction is continued when the condition evaluation and action execution are completed.

Deferred:

The triggering transaction is continued after an event is detected. Condition evaluation and action execution are done at the end of the triggering transaction before it commits.

Detached (or decoupled):

Condition evaluation and action execution are done in a separate transaction (or triggered transaction) from the triggering transaction. The detached mode can be classified into two types (totally independent and causally dependent). When two transactions are totally independent, the triggered transaction is executed regardless of whether the triggering transaction commits or aborts. On the other hands, the triggered transaction can commit only after the triggering transaction commits for the causally dependent mode.

3.6 Rule Priority

In addition to the parameter context and coupling mode associated with a rule, there is also a priority assigned to each rule. The default priority of a rule is a priority of 1. The priorities increase with the increase in numerical values. That is, 2 is a higher priority than 1, 3 is a higher priority than 2 and so on. Rules of the same priority are executed concurrently and rules of a higher priority are always executed before rules of a lower priority. It is possible that a rule raises events that in turn could fire more rules and so on. This results in a cascaded rule execution. Furthermore, rules can be specified either in the immediate coupling mode or the deferred coupling mode. Both the priority and coupling mode of a rule have to be taken into account for scheduling the rule for execution.

3.7 Java LED

We use the Java LED—Local Event Detector in our research work towards making Sybase fully active. It is used to detect the database events mapped to primitive events and composite events. LED is a system designed to provide support for events and rules in Java applications (whether it is a DBMS developed in Java or a general Java application) in a *seamless* manner. It is actually an event detector that was implemented to detect events in Java applications and executes rules defined on them. Primitive event detection as well as composite event detection in various parameter contexts and coupling modes has been implemented in LED. Also, the application developer has to explicitly put the raiseBeginEvent and raiseEndEvent calls inside the methods that have been defined as primitive events [17].

3.8 Java SPP

We use Java Snoop Pre Processor (Java SPP), often referred to as SPP in this thesis, to parse the SNOOP expression that has event operators connecting primitive or composite events. It takes a file with sjava as file extension as input and generates a java file and text file as outputs. The sjava file input contains the SNOOP expression. The output java file will contain the API for creating the composite event node and also the rule API. This conversion is done by SPP. The text file contains the names of the constituent events of the composite event defined. It is easier to specify events and rules at a higher level of abstraction. Hence to make definition easier and to avoid writing APIs we use SPP that translates event and rule definitions into appropriate APIs.

Snoop is an Event Definition Language (EDL). The Snoop Preprocessor:

1. Parses user defined events and rule specification expressions in Snoop.
2. Inserts appropriate Java code in application program.

Defining Events:

For Example,

An event is specified using the following format:

```
event begin ( sellStockBeginTest2 : VkTest : test2 ) void sellStock ( int dollar );
```

This event definition, after preprocessing, is translated to

```
EventHandle sellStockBeginTest2 =
myAgent.createPrimitiveEvent("sellStockBeginTest2","VkTest",EventModifier.BEGIN,
        "void sellStock(int dollar)",test2,DetectionMode.SYNCHRONOUS);
```

Defining Rules:

For Example the rule definition given as:

```
rule R5 [ sellStockBegin , VkTest.True , VkTest.displayPrice , 2 , DEFAULT , RECENT ];
```

is translated to:

```
myAgent.createRule("R5",sellStockBegin,"VkTest.True”,"VkTest.displayPrice",2,
        CouplingMode.DEFAULT ,ParamContext.RECENT];
```

3.9 Sybase Triggers

A **trigger** is a stored procedure that goes into effect when you insert, delete, or update data in a table. You can use triggers to perform a number of automatic actions, such as cascading changes through related tables, enforcing column restrictions, comparing the results of data modifications, and maintaining the referential integrity of data across a database [16].

Triggers can help maintain the referential integrity of your data by maintaining consistency among logically related data in different tables. You have **referential integrity** when the primary key values match the corresponding foreign key values.

The main advantage of triggers is that they are **automatic**. They work no matter what caused the data modification—a clerk's data entry or an application action. A trigger is specific to one or more of the data modification operations, **update**, **insert**, and **delete**. A trigger is executed once for each SQL statement [16].

Here is the complete **create trigger** syntax:

```
create trigger [ owner.] trigger_name
on [ owner.] table_name
for {insert , update , delete}
as SQL_statements
```

Or, using the **if update** clause:

```
create trigger [ owner.] trigger_name
on [ owner.] table_name
for {insert , update}
as
[if update ( column_name)
[{and | or} update ( column_name)]...]
```

SQL_statements

[if update (column_name)

[{and | or} update (column_name)]...

SQL_statements]...

The **create** clause creates the trigger and names it. A trigger's name must conform to the rules for identifiers. The **on** clause gives the name of the table that activates the trigger. This table is sometimes called the **trigger table**.

A trigger is created in the current database, although it can reference objects in other databases. The owner name that qualifies the trigger name must be the same as the one in the table. No one except the table owner can create a trigger on a table. If the table owner is given with the table name in the **create trigger** clause or the **on** clause, it must also be specified in the other clause.

SQL Statements That Are Not Allowed in Triggers

Since triggers execute as part of a transaction, the following statements are not allowed in a trigger [16]:

• All **create** commands, including **create database**, **create table**, **create index**, **create procedure**, **create default**, **create rule**, **create trigger**, and **create view**

• All **drop** commands

• **Alter table** and **alter database**

• **Truncate table**

• **Grant** and **revoke**

• **Update statistics**

• **Reconfigure**

• **Load database** and **load transaction**

• **Disk init**, **disk mirror**, **disk refit**, **disk reinit**, **disk remirror**, **disk unmirror**

• **Select into**

Trigger Restrictions:

Adaptive Server imposes the following limitations on triggers [16]:

• **A table can have a maximum of three triggers: one update trigger, one insert trigger, and one delete trigger.**

• Each trigger can apply to only one table. However, a single trigger can apply to all three user actions: **update**, **insert**, and **delete**.

• You cannot create a trigger on a view or on a temporary table, though triggers can reference views or temporary tables.

• The **writetext** statement will not activate insert or update triggers.

• Although a **truncate table** statement is, in effect, like a **delete** without a **where** clause, because it removes all rows, it cannot fire a trigger, because individual row deletions are not logged.

• You cannot create a trigger or build an index or a view on a temporary object (*@object*)

• You cannot create triggers on system tables. If you try to create a trigger on a system table, Adaptive Server returns an error message and cancels the trigger.

• You cannot use triggers that select from a *text* column or an *image* column of the inserted or deleted table.

• If Component Integration Services is enabled, triggers have limited usefulness on proxy tables because you cannot examine the rows being inserted, updated, or deleted (via the *inserted* and *deleted* tables). You can create a trigger on a proxy table, and it can be invoked. However, deleted or inserted data is not written to the transaction log for proxy tables because the **insert** is passed to the remote server. Hence, the inserted and deleted tables, which are actually views to the transaction log, contain no data for proxy tables.

We can see from the first restriction that there can be maximum of three triggers for a table. This poses a serious hurdle for the active capability in Sybase. So we start from here to support composite events and prioritized rules in Sybase.

3.10 Java Sockets

We use java as our programming language to implement the ECA agent to enhance the active capability in Sybase. For connecting to server from client we use java sockets. Hence we give an overview of Java sockets here [2].

In client-server applications, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

TCP provides a reliable, point-to-point communication channel that client-server application on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection with each another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

A **socket** is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes Socket and ServerSocket that implement the client side of the connection and the server side of the connection, respectively. Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

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

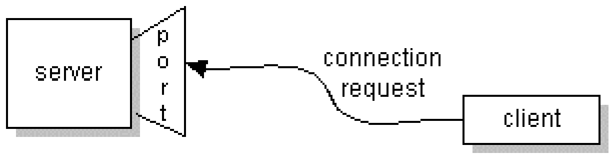

Figure 3.1. Connection.

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket (and consequently a different port number) so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

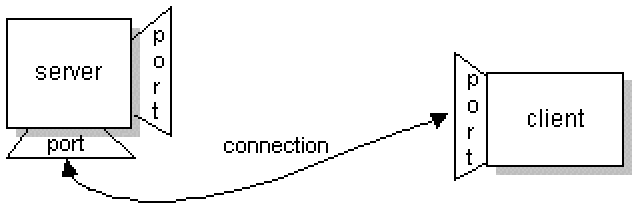

Figure 3.2. Server Listening.

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. Note that the socket on the client side is not bound to the port number used to rendezvous with the server. Rather, the client is assigned a port number local to the machine on which the client is running. The client and server can now communicate.

3.11 JDBC

We use the industry standard JDBC to connect to Sybase ASE server from the ECA Agent. JDBC technology is an API that lets you access virtually any tabular data source from the Java programming language. It provides cross-DBMS connectivity to a wide range of SQL databases, and now, with the new JDBC API, it also provides access to other tabular data sources, such as spreadsheets or flat files [1].

The JDBC API allows developers to take advantage of the Java platform's "Write Once, Run Anywhere" capabilities for industrial strength, cross-platform applications that require access to enterprise data. With a JDBC technology-enabled driver, a developer can easily connect all corporate data even in a heterogeneous environment.

Advantages of JDBC Technology

1. Leverage Existing Enterprise Data: With JDBC technology, businesses are not locked in any proprietary architecture, and can continue to use their installed databases and access information easily—even if it is stored on different database management systems.

2. Simplified Enterprise Development: The combination of the Java API and the JDBC API makes application development easy and economical. JDBC hides the complexity of many data access tasks, doing most of the "heavy lifting" for the programmer behind the scenes. The JDBC API is simple to learn, easy to deploy, and inexpensive to maintain.

3. Zero Configurations for Network Computers: With the JDBC API, no configuration is required on the client side. With a driver written in the Java programming language, all the information needed to make a connection is completely defined by the

JDBC URL or by a DataSource object registered with a Java Naming and Directory Interface™ (JNDI) naming service. Zero configurations for clients support the network-computing paradigm and centralize software maintenance.

Sybase provides a way to connect to its ASE Server from any java application through jConnect—The Sybase JDBC driver. jConnect is Sybase's high-performance JDBC driver. jConnect is both a:

Net-protocol/all—Java driver within a three-tier environment, and a

Native-protocol/all—Java driver within a two-tier environment.

3.12 Terminology used

The whole system designed is referred to as **ECA Agent System—**Event Condition and Action—as the whole process of enhancing active technology is based on ECA paradigm. The server is referred to as ECA Server and the client that serves as an interface for the user is referred to as ECA client.

The insert, delete and update operations are treated as primitive events. The trigger defined is referred to as **Primitive Trigger** or Primitive Event Trigger. If more triggers are defined on the same table for same operation then it is referred to as **Repeat Trigger** and the event is referred to as Repeat Primitive event. The trigger defined on composite event with SNOOP expression is referred as **Composite Trigger** or Composite Event Trigger.

This chapter gave an overview of all the components used on this thesis like java sockets, JDBC. It also gave background information about JavaLED, SPP, SNOOP, Sybase triggers and event operators.

4. DESIGN ISSUES

This chapter discusses the design issues of the ECA Agent System.

4.1 General Architecture

In Sybase, a table can have a maximum of three triggers: one update trigger, one insert trigger and one delete trigger. Our aim is to make Sybase fully active. First step towards doing this is to support multiple triggers for a table. Then we have to support composite events. To achieve these goals, we have to have control over the input commands given by the user. Also, Event—Condition—Action based approach is best suited for achieving our goal.

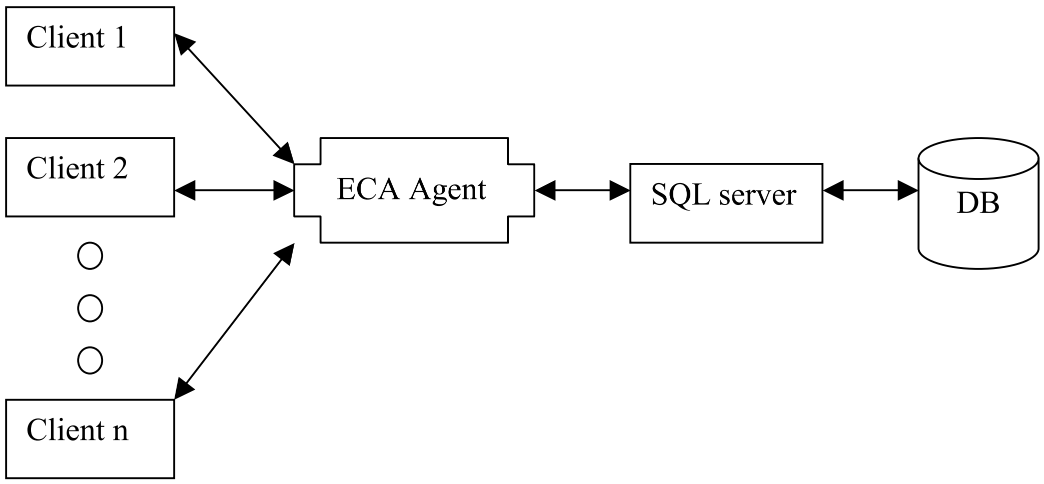

Figure 4.1. General Architecture.

As shown in figure 4.3, the user gives the input command in the ECA Client. This is routed to the ECA Server. The ECA server sends it to the corresponding database server—Sybase server, in our case. This is a generalized approach in the sense it can connect to any

SQL server (Oracle and DB2 have been used in other projects). The reason for providing the ECA Agent System is to gain control over the inputs thus facilitating us to achieve our goal.

4.2 Issues:

Now the ECA server being introduced between the client and the Sybase server should take the role of the server and do many functions that the Sybase server does. These are the issues that need to be addressed:

1. Multi user/Multi Database connectivity
2. Full Active capability
    a. Multiple rules on the same primitive event
    b. Composite events
    c. Parameter contexts
3. Persistence of events
    a. Persistence of triggers across sessions
    b. Persistence of meta data

4.2.1 Multi user/ Multi Database connectivity:

The underlying RDBMS—the Sybase server allows a single user to connect to different databases and at the same time allows many users to connect to the same database. In our case, ECA Server has to provide this functionality.

We use JavaLED to detect events. The JavaLED maintains event graphs to register and notify events. There is a facility in JavaLED to create multiple instances of the event detector and create event graphs in each one of them. In our case, for a specific user—database combination we create an instance of the named event detector instance as shown in figure 4.2. Each instance of the event detector will have the event graph corresponding to the events created by that user for that database. The following features from JavaLED are used for this purpose: [17]:

1. Multiple event graphs .
2. The API methods invoked on a specific instance of the ECA Agent class.
3. Multiple event graphs created by creating the events and rules on different ECA Agent instances.
4. An event created in more than one event graph by invoking the same API on different instances.
5. Logical grouping of events and rules within the same application.

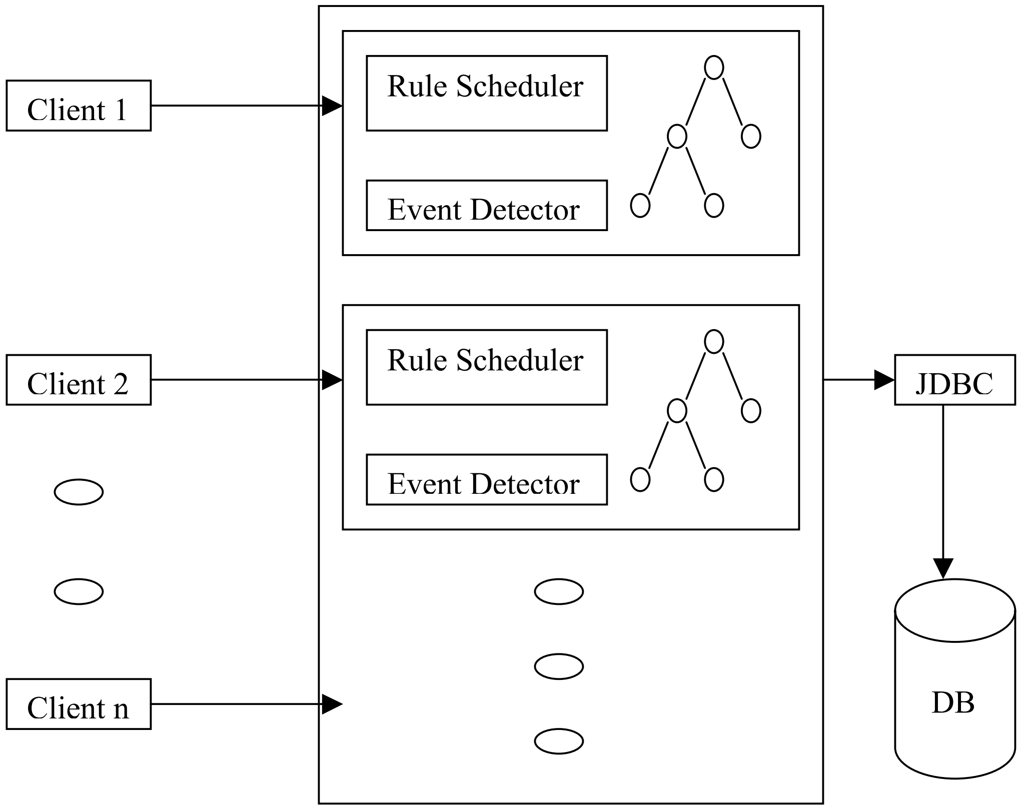

Figure 4.2. Muliple ECA Agents.

We create a file with .java as extension that contain APIs to create the events in the JavaLED. These files have to be created and kept in separate directory for each user—DB combination. Also the event information is stored in the underlying RDBMS by creating different relational tables. These tables should have the username and the database name to identify the events and triggers created by different users.

4.2.2 Full Active Capability:

Most commercial RDBMSs support only triggers on insert/delete/update operations on a table. They do not support the notion of triggers on composite events. In this research work, 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—here the event is called primitive event. We allow users to define additional action on the same event by reusing the event name and giving a different trigger name thus supporting multiple triggers, which the underlying RDBMS (Sybase) does not support. The 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 RDBMS. So the syntax is more similar to the trigger syntax in Sybase. The ECA Server should also handle the special case of "if update" clause triggers in Sybase. This is explained in detail in section 6.2.

4.2.3 Persistence of Events:

The event graph created by JavaLED is stored in main memory. Hence, when a session ends the event graph is not persisted. But in a typical database scenario every event and the corresponding transaction has to be persisted over sessions. In this case, the ECA Agent server should take steps to persist these events. For this we have two choices:

1. Using Files.

2. Using RDBMS.

If we decide on using files, separate care has to be taken by ECA server for all file operations such as inserting event information, retrieving the event information when necessary, deleting the event information, naming the files and handling them. But if we use the underlying RDBMS, this information is taken care by itself when we create relational tables. We can use the SQL query mechanism to insert, delete or retrieve the event information, which is relatively easy than the file handling. Hence we use Sybase to store the event information and use it to draw the event graph again when a session ends and new session starts.

2 4.3 Architecture of the ECA Server

The ECA server has many modules as shown in figure 4.2. In general, the client input goes to the ServeOneClient module of the ECA Server. It routes it to Language filter, which checks if it is an extended trigger command. If so it is sent to ECA Parser, else to the JDBC Call module. The ECA Parser parses the command and sends it to the Persistence Manager. The Persistence Manager persists the trigger and event details. The drop trigger module handles the dropping of events. Each module is explained in detail in this chapter.

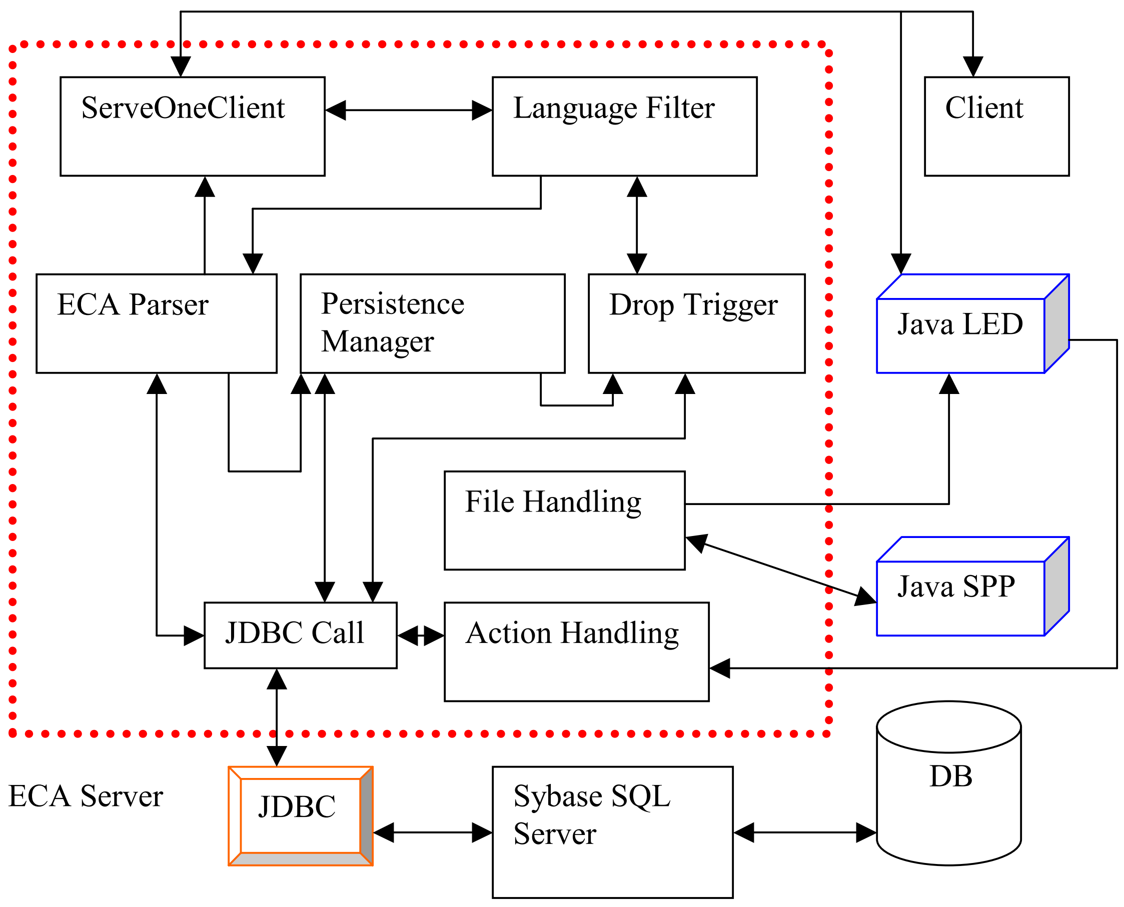

Figure 4.3. Detailed Architecture.

4.3.1 ServeOneClient

This module is responsible for making the client connection, getting the input, connecting to the respective database. When a client tries to connect to the ECA server, the ServeOneClient module makes connection. The input given to the ECA server comes to this module, which routes it to Language Filter.

4.3.2 Language Filter

This module checks the command sent to it from ServeOneClient. If it is an ECA command, it sends it to ECA Parser, else it is sent to the JDBC Call module. It also sends the command to specific sub module under ECA Parser module as it has many sub modules. Also the drop trigger command has to be sent to the proper sub module in Drop Trigger module.

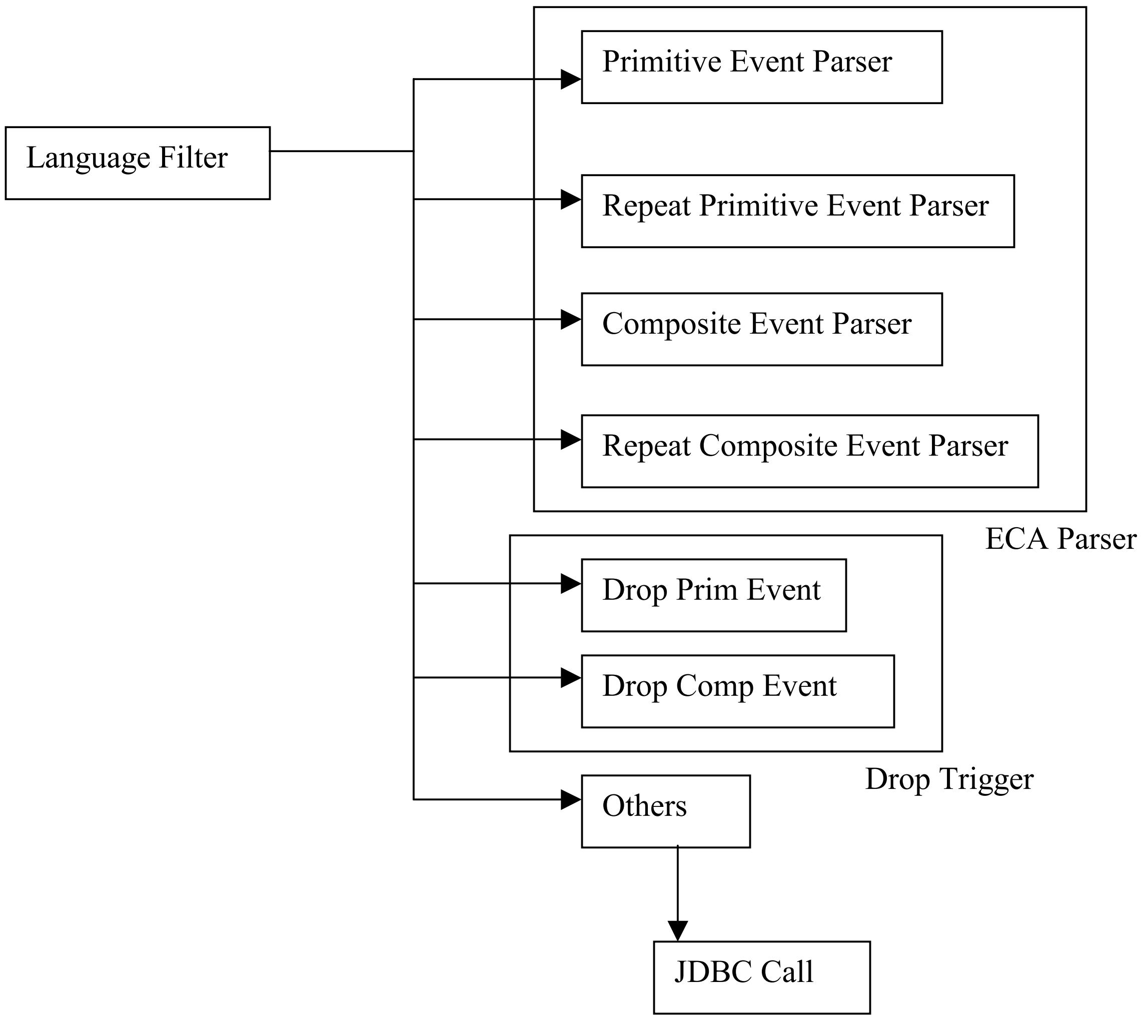

Figure 4.4. Language Filter.

4.3.3 ECA Parser

It has the following sub modules to which the Language Filter sends the create trigger ECA Command

1. Primitive Event Parser
2. Repeat Primitive Event Parser
3. Composite Event Parser
4. Repeat Composite Event Parser

4.3.3.1 Primitive Event Parser

This is a sub module under the ECA Parser. This sub module is exclusively used for handling the primitive events. When the primitive event creation command comes from the Language Filter, this module checks for correctness of syntax. If the syntax is correct, it checks for duplicate trigger name and event name. If these are correct, then the code for registering the event with Java LED is sent to File Handling module. Also, the trigger command is sent to the persistence manager for creating the trigger. After the successful completion of registration arrives from the File Handling module, the information is sent to the Persistence Manager for persisting the event information in the RDBMS. If there is an error at any stage, it is sent back to ServeOneClient, which forwards it to client.

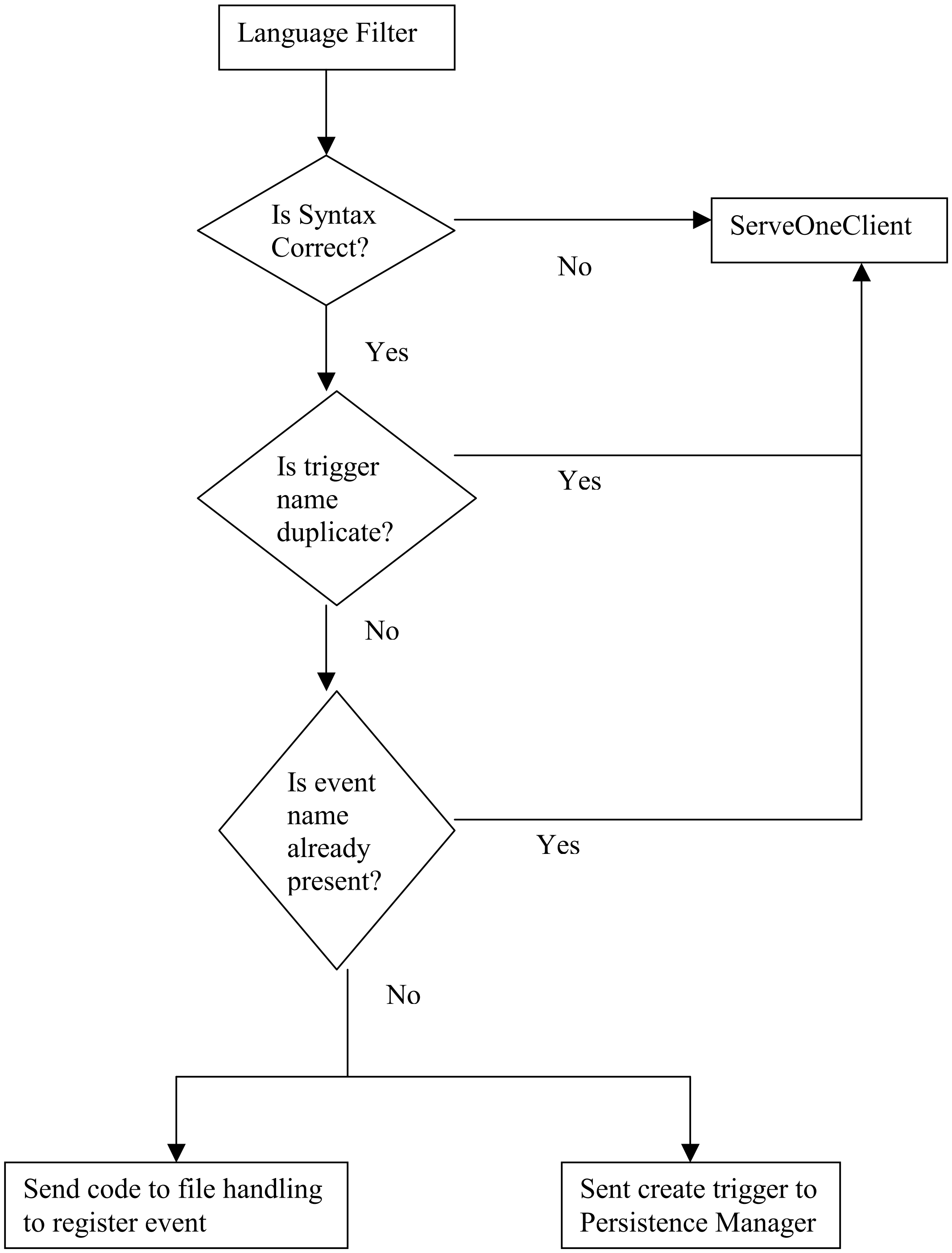
Language Filter
Is Syntax Correct?
No
ServeOneClient
Yes
Is trigger name duplicate?
Yes
No
Is event name already present?
Yes
No
Send code to file handling to register event
Sent create trigger to Persistence Manager

Figure 4.5. Primitive Event Parser.

4.3.3.2 Repeat Primitive Event Parser

This is also a sub module under the ECA Parser module. This parser module takes care of parsing the repeat primitive event creation command.

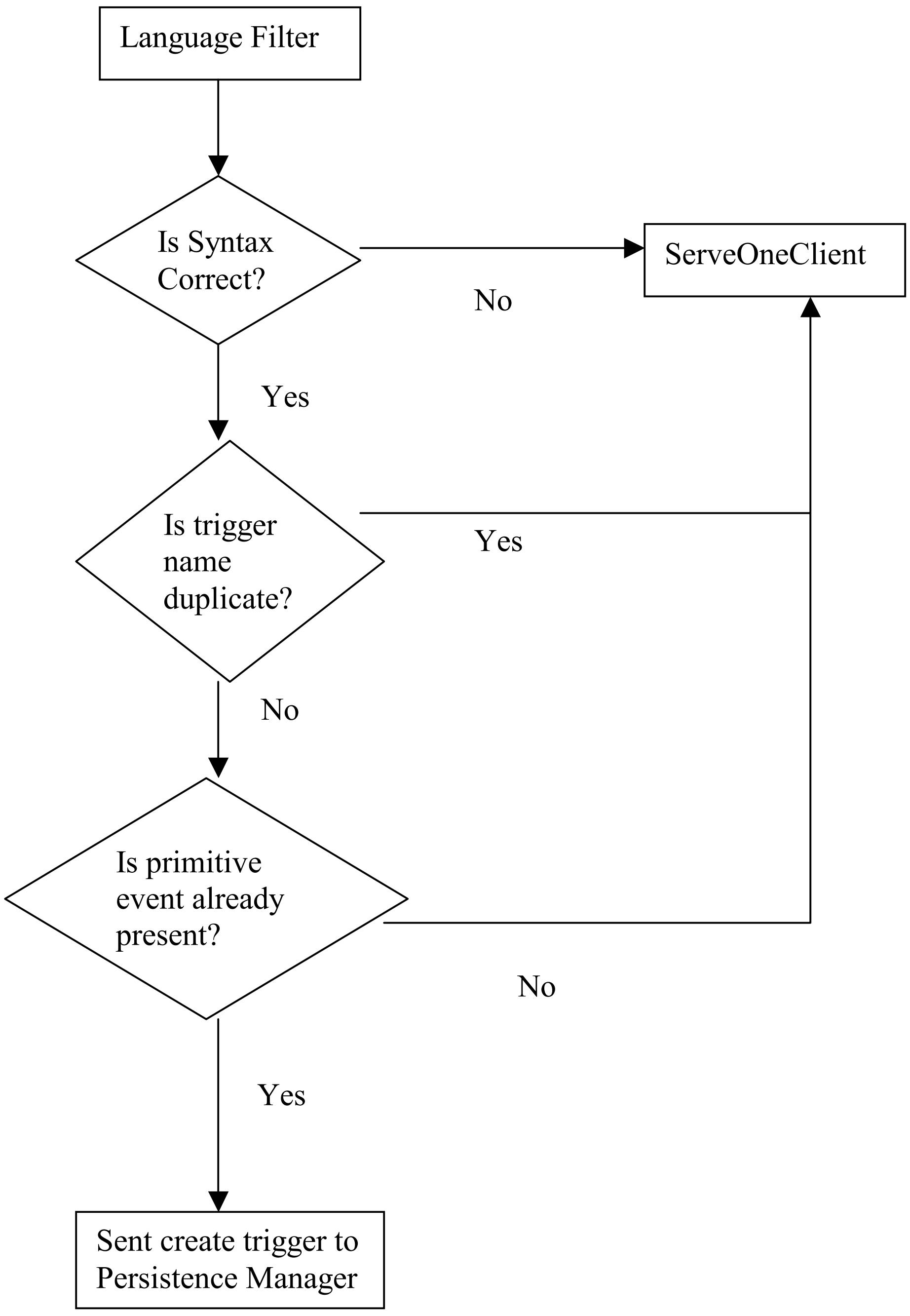
Language Filter
Is Syntax Correct?
No
ServeOneClient
Yes
Is trigger name duplicate?
Yes
No
Is primitive event already present?
No
Yes
Sent create trigger to Persistence Manager

Figure 4.6. Repeat Primitive Event Parser.

This is similar to primitive event parser except for some steps. After checking for the syntax and trigger name duplication, the parser has to check if the primitive event is defined. If it is present then the create command and the information for persisting the information about the event and trigger is sent to the Persistence Manager. Here there is no need to generate the file as the event is already registered with the LED and the java file is already present.

4.3.3.3 Composite Event Parser

This module is also a sub module under ECA Parser. This module is responsible for handling create composite command. As part of parsing, syntax, trigger and event names are checked for correctness. Then, the code is sent to the Java Snoop Pre Processor (JavaSPP), which generates (as a side effect of parsing and API generation) the constituent event list of the composite event to be registered. Using that, this module checks for the presence of constituent events. If the constituent events are not present then an error is sent to ServeOneClient. If constituent events are present, the code is sent to the File Handling module for registering the composite event. Upon successful registration, the information is sent to the Persistence Manager for making the event information persistent in the RDBMS.

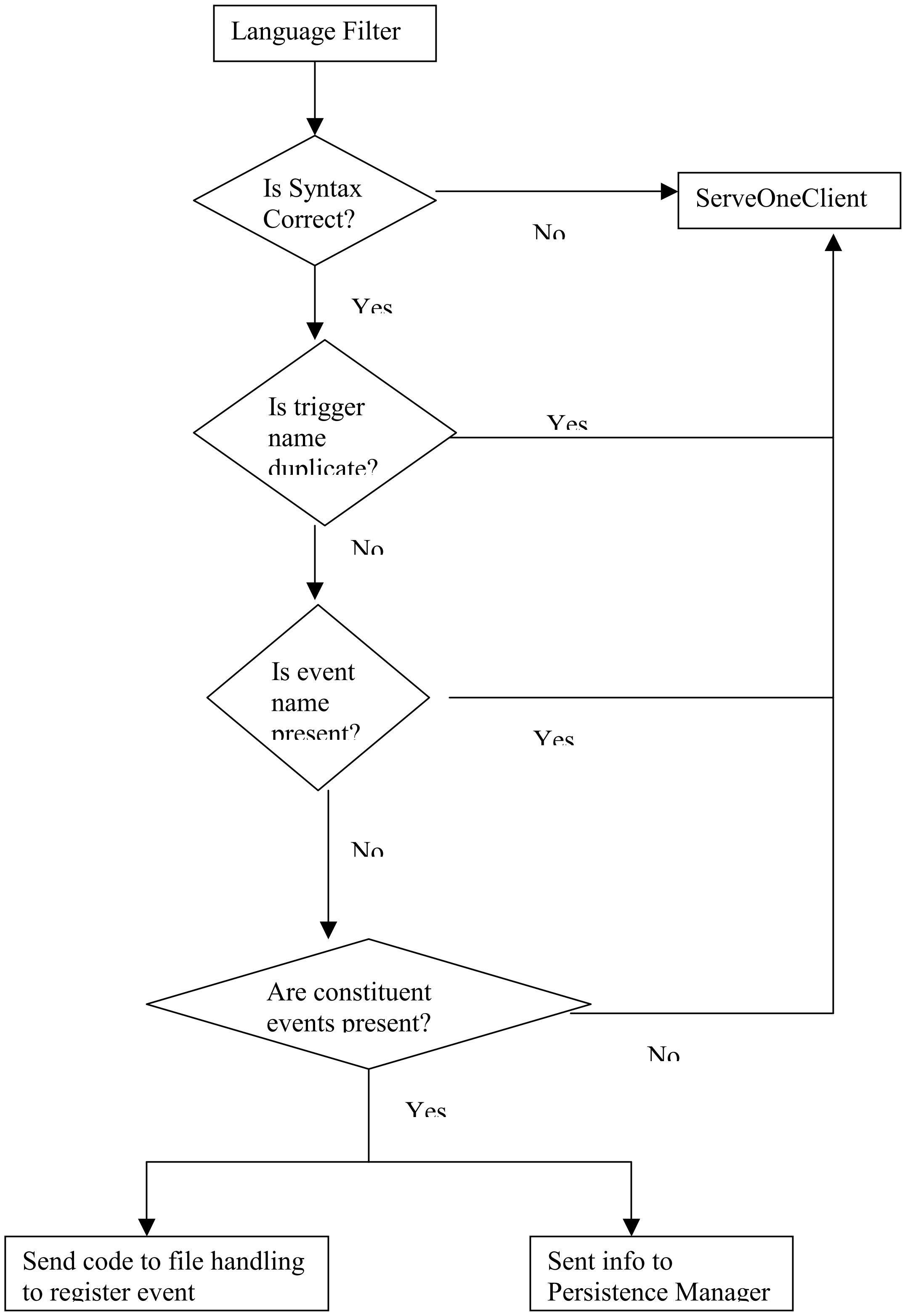
Language Filter
Is Syntax Correct?
No
ServeOneClient
Yes
Is trigger name duplicate?
Yes
No
Is event name present?
Yes
No
Are constituent events present?
No
Yes
Send code to file handling to register event
Sent info to Persistence Manager

Figure 4.7. Composite Event. Parser.

4.3.4 Persistence Manager

The create trigger command comes from ECA Parser to this module. In order to support multiple triggers, we have to keep track of the triggers already created by the user. The best way to do this is to persist the information about them in the RDBMS by creating system tables. Also action portion can be converted into a stored procedure and can be stored in the database itself. The execute call for this can be given in the trigger. This makes the handling and dropping of multiple triggers easier.

The table SYSECATRIGGER keeps track of the trigger information. In all the system tables created, DBNAME refers to the database name; USERNAME refers to the user connected to that database. EVENTNAME refers to the event name associated with the trigger.

Table 4.1. SYSECATRIGGER

DBNAME	USERNAME	TRIGGERNAME	TRIGGERPROC	TIMESTAMP	EVENTNAME	TRTYPE

TRIGGERNAME – name of the user defined trigger

TRIGGERPROC – name of the stored procedure created for that trigger

TIMESTAMP – timestamp at which the trigger is created

TRTYPE – trigger type {normal | if}

The LED we are using does not persist the information about the triggers and the events. So it is the responsibility of the ECA Agent to persist the event information. For this we use two system tables: SYSPRIMITIVEEVENT used to persist the primitive events and SYSCOMPOSITEVENT used to persist the composite events.

Table 4.2. SYSPRIMITIVEEVENT

DBNAME	USERNAME	EVENTNAME	TABLENAME	OPERATION	BEAFOPERATION	TIMESTAMP	VNO

Table 4.3. SYSCOMPOSITEVENT

DBNAME	USERNAME	EVENTNAME	EVENTDESCRIBE	TIMESTAMP	COUPLING	CONTEXT	PRIORITY

TABLENAME – the table on which primitive event is defined

OPERATION – {insert | delete | update}

BEAFOPERATION – {before | after}

TIMESTAMP – the timestamp at which the event is created

VNO – version number to denote the occurrence

EVENTDESCRIBE – description of the composite event

COUPLING –{immediate | deferred | detached}

CONTEXT – {recent | chronicle | continuous | cumulative}

PRIORITY – a positive integer

The version number in the SYSPRIMITIVEEVENT table is obtained from the VERSION table shown below.

Table 4.4. VERSION

VNO

4.3.5 Drop Trigger

This module is responsible for dropping an event and the associated trigger. It has two sub modules—DropPrimTrigger and DropCompTrigger. It is important that the primitive event should not be dropped if a composite event is subscribed to it. In order to keep track of this information we use the SYSDROP table.

Table 4.5. SYSDROP

CONSEVENTNAME	CONTEXT	COMPEVENTNAME

CONSEVENTNAME – constituent event name

CONTEXT – the context associated with the composite event

COMPEVENTNAME—composite event name

4.3.5.1 Drop Primitive Event Trigger

DropPrimTrigger is a sub module under drop trigger module that is used for dropping the primitive event and its associated trigger.

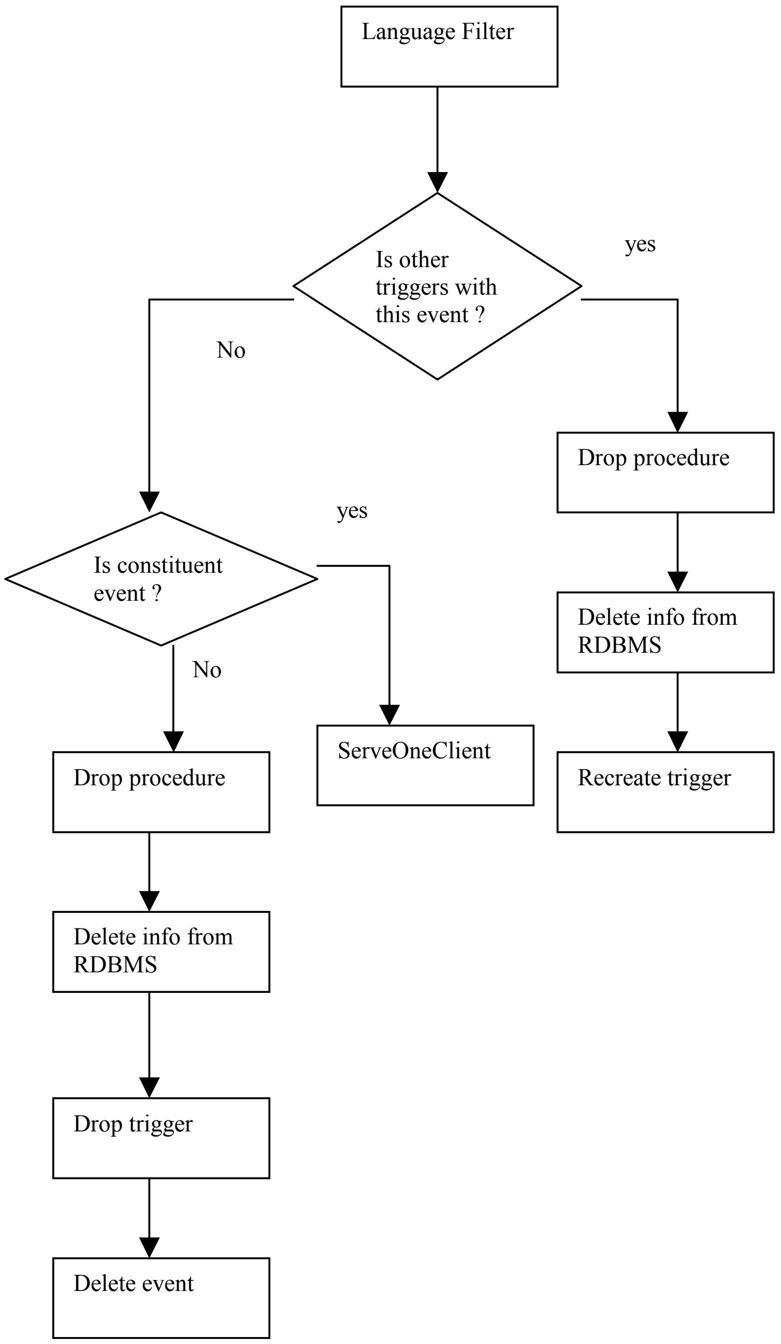

Figure 4.8. Drop Primitive Event.

The first step is to check if there are any other triggers associated with this event. If so the event should not be deleted but only the trigger and the stored procedure associated with it has to be dropped. The corresponding trigger information in the SYSECATRIGGER has to be deleted. Then the trigger has to be recreated with the action of the other triggers associated with the event.

On the other hand if the event associated with the user given trigger has only one trigger defined on it, then this module checks if it is a constituent event of any other composite event. If so an error is sent to the ServeOneClient module. If not, the event has to be deleted from event graph in Java LED. The event and trigger information has to be deleted from the system tables. Finally, the trigger and the stored procedure have to be dropped.

4.3.5.2 Drop Composite event trigger

DropCompTrigger is a sub module under the drop trigger module that is used for dropping triggers associated with composite event.

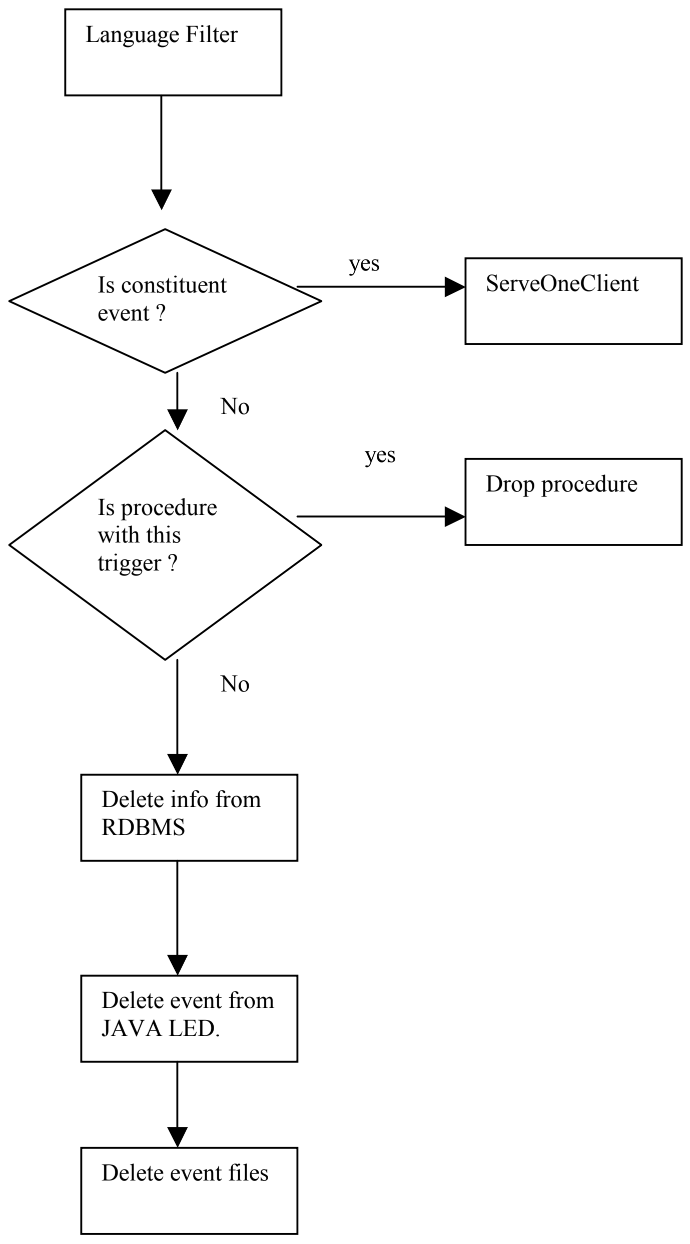

Figure 4.9. Drop Composite Event.

The Language Filter sends the drop trigger command associated with the composite event to this module. The first step is to check if it is a constituent event of any other composite event. If so an error is sent back to ServeOneClient. Then the association of stored procedure with this event is checked. If it is present, then it is dropped. The trigger information in the system tables is deleted and the node is deleted from the Java LED by calling the corresponding API.

4.3.6 Other Modules

File Handling module is responsible for all the file handlings in the ECA server. The functions performed by this module are: creating sjava file for processing by the JavaSPP. The files are compiled, and executed invoking the corresponding method and registering events and rules with LED. The Action Handling module takes care of the action portion to be performed after raising the events. This module also handles the user given action portion. JDBC call module is responsible for all JDBC calls. Every call to the JDBC from every module in the ECA Server goes through this module. This has methods to get the result set for the queries sent such as select.

4.4 ECA Agent Client

The design of the ECA agent client is done as a thin client. That is, it performs as minimum functions as possible and does not have the logic to process the input. It provides the GUI for getting the information such as IP address of ECA server, RDBMS, the username, password and the database to be connected and another GUI for displaying the results.

4.5 Extended Trigger Syntax

The trigger syntax provided by Sybase has to be extended to support multiple triggers, primitive events and composite events. Keeping the following facts in mind, the design of the extension of the trigger syntax is done:

1. The user works in the SQL Environment.
2. The user works on Sybase SQL Server.
3. Minimal extension should be made to the existing trigger syntax.

4.5.1 Sybase Trigger Syntax

Here is the complete **create trigger** syntax:

Create trigger [owner.] trigger_name

On [owner.] table_name

For {insert , update , delete}

as SQL_statements

Or, using the **if update** clause:

create trigger [owner.] trigger_name

on [owner.] table_name

for {insert , update}

as

[if update (column_name)

[{and | or} update (column_name)]...]

SQL_statements

[if update (column_name)

[{and | or} update (column_name)]...

SQL_statements]...

The **create** clause creates the trigger and names it. A trigger’s name must conform to the rules for identifiers. The **on** clause gives the name of the table that activates the trigger. This table is sometimes called the **trigger table** [16].

4.5.2 Primitive Event Trigger

The syntax for defining primitive event trigger is given below.

```
Create trigger trigger_name
On [ owner.] table_name
For {insert , update , delete}
event event_name
as SQL_statements
Or, using the if update clause:
create trigger [ owner.] trigger_name
on [ owner.] table_name
for {insert , update}
event event_name
as
[if update ( column_name){and | or} update ( column_name)]...]
Begin
SQL_statements
End
[if update ( column_name){and | or} update ( column_name)]….]
Begin
SQL_statements
End…..
```

The only extension made is that the user has to specify a unique event name along with the keyword '***event***'. This is done to support multiple triggers by naming a primitive event.

4.5.3 Repeat Primitive event trigger

The syntax for defining repeat primitive event trigger is given below.

```
Create trigger trigger_name event event_name
as SQL_statements
Or, using the if update clause:
create trigger trigger_name event event_name
as
[if update ( column_name)
[{and | or} update ( column_name)]...]
Begin
SQL_statements
End
[if update ( column_name)
[{and | or} update ( column_name)]...]
Begin
SQL_statements
End…..
```

Here the user need not give the table name and operation as in a regular trigger, as the ECA Parser takes care of that. The table and operation associated with the event name (already defined) is used for this purpose.

4.5.4 Composite event Trigger

The syntax for defining composite event trigger is given below.

Create trigger trigger_name ***event event_name= snoop expression:***

[coupling mode] [parameter context] [priority]

Begin [sp]

SQL_statements

End

Snoop expression—the event expression using the syntax of the event definition language SNOOP.

Coupling mode—Immediate, deferred, detached.

Parameter context—Recent, continuous, cumulative, chronicle.

Priority—any positive integer.

Sp—stored procedure if the user wants the SQL statements to be performed as a transaction.

4.6 Need for Notification

We use JavaLED to detect the composite events and execute the rules according to the priority given. The events insert/delete/update operations are executed by the server on the database side and hence the trigger is also executed by the server. But as we are using LED to raise the primitive event and thus raise the subscribed composite event we need a mechanism to notify LED of the occurrence of primitive events. In the Sybase ECA Agent discussed in chapter 2, there is a built-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. You cannot make an RPC call to the ECA server through JDBC. For this reason, we insert the event information in a table and return; in the ECA server a check is made on this table to raise the event. There is a decoupling between the trigger execution and raising of the event in two address spaces which are connected by the JDBC. This table is NOTIFY [21].

Table 4.6. NOTIFY

EVENTNAME	TABLENAME	VNO

4.7 Getting Parameters

Primitive events are mapped to database operations—insert, delete or update. We use LED to detect composite events. If the primitive event occurs on the Sybase SQL server, it is not possible to get the parameters of the primitive event in the composite event action, which occurs on LED. When a primitive event occurs, the trigger associated with the event is fired. In Sybase, at the time of trigger execution, if the table on which the trigger is defined is modified, then the tuples that are inserted on that table are inserted into the 'Inserted' table and the tuples that are deleted from that table are inserted into the 'Deleted' table. These tables are accessible only during the execution of the body of the trigger and that too only from inside 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_inserted, 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. 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 the VERSION table. This VERSION table has been introduced to keep a global count of the occurrence of a primitive event and is stored in VNO attribute of it.

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. 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. This is analogous to the Inserted and Deleted tables that can be accessed from inside the Sybase triggers.

Java LED provides API to insert parameters before raising the primitive event. We pass minimum number of 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.

Table 4.7. SYSCONTEXT

TABLENAME	CONTEXT	VNO

TABLENAME—The name of the table on which the constituent event is defined.

CONTEXT—The context on which the composite event is defined.

VNO—The version number of the constituent event.

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.

This chapter gave the overall design issues of the ECA Agent system. It also discussed in detailed about the functions of each and every module in the ECA Agent system. Also the chapter gave the syntax proposed for triggers to support primitive and composite events. And the final discussion was on notification to LED and getting the parameters for composite events.

5. IMPLEMENTATION

This chapter explains in detail the problems involved and solutions proposed for implementing ECA Agent system.

5.1 ECA Agent System

It is implemented in Java so that it is portable to other platforms. The connection to the Sybase database server is through the industry standard—JDBC. The subsections explain this in clear.

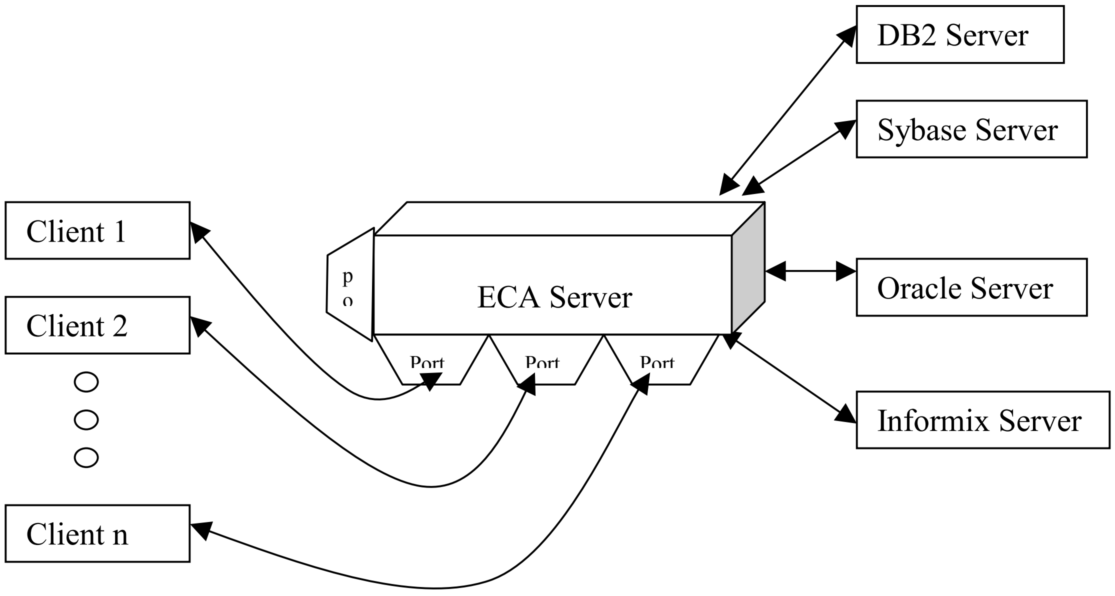

Figure 5.1. Server Client connection.

5.1.1 Connection

When the ECA Agent server is up, it waits on the specified port on the server socket for the client connection. Whenever a client connection is accepted, a dedicated socket is assigned to it by java. We spawn a new thread to serve each client.

5.1.2 System Tables

The dedicated thread loads the JDBC driver and connects to the Sybase database server. The next step is to check if the user is new. If the user is new, the system tables are created using transact SQL facility in Sybase. If the user is not new, the events defined by that user are restored and the information is sent to java LED to build the event graph. The system table notify is cleared and the version number in the version table is reset to zero.

5.1.3 Restoration of events

The SYSPRIMITIVEEVENT table is used to retrieve the event names. With the event names, the class name is found as c_eventname. Now the class loader class in java is used to load the class, and the method in it containing the API to register the primitive event is invoked. In the same way, using the SYSCOMPOSITEVENT table composite events and rules are restored.

5.1.4 Registering events with Java LED

The issue of registering the event with Java LED is not straightforward because the users create the events at runtime. In the database scenario we associate every database event with a primitive event in java LED—so to register them with LED we have to create the API and execute it. So we create a java file for this purpose. We use Runtime Class in java to compile the java file. Then the class file is loaded and the method inside the class file is invoked to register the event.

5.2 ECA Agent Client

This is a thin client with minimal functionality in it. When the client is brought up, the user has to provide some information—SQL Server to connect to: as one of Sybase or Oracle or DB2, the database name to connect to, the IP address/name of the machine in which SQL server is running, the IP address/name of the machine in which ECA server is running, the user name and password to connect to the SQL server. This information is sent through the socket to the server, which is listening on a server socket. Once this information is authenticated, the client provides an interactive interface to talk to the server. This interface has options to send SQL queries, select an arbitrary SQL file with commands and some sample trigger and SQL commands.

5.3 MULTI USER/MULTI DATABASE

The design and implementation done so for is for a single user connecting to a single database at a time. Since a database supports multiple users connecting to one or more databases at the same time, we need to support the same. Also, databases preserve the data and triggers created by a user from one session to another. This entails that the ECA agent server do the same for primitive and composite events.

5.3.1 Problem Statement

Let us consider a scenario in which there are two users: user-1 and user-2. Suppose user-1 creates a primitive event addw. If the user-2 also creates an event with the same name addw, then the java file, which we associate for every event will have the same name. Hence, the event representation of user-1 will be replaced by user-2. Let us consider another scenario, in which user-1 tries to delete the event created by him, this will delete the java file associated with it. If the other user also uses this event then he will lose this event. Hence we have to address this problem.

5.3.2 Solution

In order to overcome the problem described above, we have to separate the files created by each user. But the same user can create the event with the same name in two different databases and can use them at different times. Hence, we create a metadata directory, as this is more similar to the metadata stored in databases, and place files created by each user—database combination in separate directory. It is a unique combination in all cases.

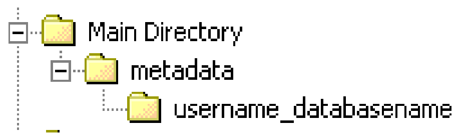

Figure 5.2. Multi user / Multi database.

5.3.3 Implementation Issues

The directory name is going to change every time a user logs into the system. This is because there may be many numbers of users connecting to the same database. First thing we do is to check if the directory is already present for the same user – database combination, if not we create a new directory. Creating the java files in a specific directory is not a problem as java provides a –d option while compilation. But compiling and loading is a serious problem.

The compilation of the java file needs to be changed to include the class path. If it moves to another directory, it cannot access the files in the ECA server. The need for accessing the ECA server files is in places where we use the JDBC calls. Hence, we have made all the files in the ECA server into a package. Now the java files associated with the events can access the files in the package by just importing that package.

Loading the class from a specific directory other than the current working directory is not possible. Hence a simple solution is to have a batch file, pass the directory name and file name as parameters to the batch file and copy the class file to the current working directory. Then load the file using Class Loader class in java and invoke the method to register with java LED. After creating the node, delete the copy in the current working directory. The File Handling module of ECA server does all these stuff.

The next problem is that when we use Java SPP, the output files comes to the current working directory. Hence we have to move the input sjava file, the outputs java file and text file from the SPP to the user—database specific directory. The File Handling module does this. It uses another batch file, which takes file name and destination directory as inputs and moves the file to that directory.

We execute the batch files by using Runtime class in java which has a execute method that does the execution as if it is done at command prompt. The problem addressed above is very serious because everything happens during runtime. We do not know the file name, the directory name, and the database name until the user connects to it. Hence we use Runtime class, batch files and execute methods.

5.3.4 System Tables

We create system tables to store information about the primitive and composite events. We also create a separate table to store the trigger information apart from what Sybase does to support multiple triggers. Each table created has a separate user name, database name attribute to make handling easy. Also, the list of tables created by ECA Agent is created once for every user—database combination. Thus the information of the event and trigger creation will be specific for that user connected to the specific database and will be stored in the corresponding table.

5.4 Persistence of Events across sessions

We use JavaLED to detect composite events. Whenever the user creates an event it is registered as a node in JavaLED. These nodes form an event graph in LED. This is in main memory. So whenever a session ends the event information is also lost. But in a typical database scenario, the user has to keep this information across the session. Hence we enter the information about the trigger and events in separate table created by us. Then whenever the user starts his new session, the java files associated with the nodes are invoked to rebuild the event graph.

This chapter gives an outline on the implementation about the ECA server and ECA client. It also gave a note on the compiling and loading the java files associated with each composite event. This chapter also gave the problem of multi user/multi database connectivity and then gave the solution provided. The next two chapters give a detailed explanation about the issues while creating, raising and handling of primitive and composite events.

6. PRIMITIVE EVENTS

This chapter deals with creation and dropping of primitive and repeat primitive events.

6.1 Primitive Triggers

As has been indicated earlier, we associate each database event—insert or delete or update (along with the table name) with a primitive event in LED.

The Syntax of the primitive event trigger is:

```
Create trigger trigger_name
On [ owner.] table_name
For {insert , update , delete}
event event_name
as SQL_statements
```

For Example,

```
Create Trigger T_ADDW on WEATHER for insert
event addw
as
begin
        insert into info values (‘sharma’,’professor’)
end
```

In the above example, the insert operation of the WEATHER table is associated with a LED event—addw.

Table 6.6. WEATHER

City	Time	Tem	Wspeed

Table 6.7. INFO

NAME	DESIGNATION

The WEATHER table has city and time as its primary key. On an insert operation in this table, the trigger is executed and the values are inserted in the INFO table. The above example is used through out this chapter.

The explanation of the primitive events is split into two phases:

1. Creation Phase
2. Execution Phase

6.1.1 Creation Phase

When the trigger in the example is given as input on the client side, it goes to ServeOneClient module of the ECA-Server, which sends it to Language Filter.

The **language filter** parses the input command into tokens and recognizes it as a primitive event command and sends it to the Primitive Event parser of the ECA parser. The recognition is done by the presence of event keyword followed by event name.

The Primitive Event parser, sub-module of the ECA parser parses the trigger command and gets the event name and trigger name. The presence of the same name is checked in the system tables. If there is duplication, an error message is sent to client via ServeOneClient module.

The next step is to check the presence of R_Inserted and R_Deleted tables. In our example, the presence of WEATHER_INSERTED and WEATHER_DELETED are checked. If they are not present, they are created.

Table 6.8. WEATHER_INSERTED

City	Time	Tem	Wspeed	VNO

The version number (VNO) is added to weather table attributes and this table is created by using the **select … into** facility of Sybase.

These tables are created with a prospective view of handling composite event triggers. This will act as inserted and deleted tables in Sybase that can be accessed from with in the triggers. Then the trigger command is send to create trigger module of the persistent manager. It waits for any error, if there is any error it is sent back to the client. If there is no error, the parser generates code to create java file and gives it to the file-handling module. The code has API to create primitive event node in LED.

This info is sent again to GeneratePersistCode module of Persistent Manager to persist the code. The ECA parser sends the code as a string to create the java file to file handling module. This module creates the java file with the name c_eventname. Then it is compiled as descried in chapter 5. The class file is loaded and the method containing the API is invoked and the event is registered with Java LED.

The first time the control comes to **persistent manager** module from the ECA parser is for creating triggers. Once the trigger is created successfully, persistent java code is created and stored by this module. **CreateTrigger module** is a sub module under persistent manager. This module is responsible for creating the trigger that is acceptable by the Sybase SQL server and also in a way that it supports multiple triggers. When the input comes to this module, it takes off the user given portion of event and event name. Then the action portion is converted into a stored procedure—with the convention of triggername_proc. Also to maintain the order and occurrence of events we add other DML statements with the user given portion.

We use stored procedures to make it easy to handle the multiple triggers. But the problem is that the user cannot access the inserted and deleted tables from the stored procedure. So we have to change “inserted” and “deleted” to “R_inserted_temp” and “R_deleted_temp”— where R is the relation name— here WEATHER. These tables are created to provide user transparency.

For our example, the stored procedure will be:

```
Create procedure T_ADDW_PROC  as
Begin
        Insert into info (‘sharma’,’professor’)
End
```

The trigger given to the Sybase will be of the form:

```
Create trigger T_ADDW on WEATHER for Insert as
Begin
```

```
update SYSPRIMITIVEEVENT set VNO=VNO+1 where EVENTNAME='addw' update VERSION set VNO=VNO+1
insert into WEATHER_inserted select inserted.city, inserted.time, inserted.tem, inserted.wspeed, VERSION.VNO from inserted, VERSION
insert into NOTIFY select EVENTNAME, TABLENAME, VERSION.VNO from SYSPRIMITIVEEVENT, VERSION where EVENTNAME='addw'
delete from WEATHER_inserted_temp
insert into WEATHER_inserted_temp select inserted.city, inserted.time, inserted.tem, inserted.wspeed from inserted
delete from WEATHER_deleted_temp
insert into WEATHER_deleted_temp select deleted.city, deleted.time, deleted.tem, deleted.wspeed from deleted
exec T_ADDW_PROC
End
```

In order to execute the user given action, we have to execute the stored procedure with an exec call. The explanation for the DML statements is given in section 6.1.2. The result of the creation of trigger and stored procedure is sent to Persistent Manager.

After the ECA parser gets a signal from Persistent Manager about the creation of trigger, the control is transferred to **Generate Persist Code Module**. This module is responsible for persisting the event information. This is done so that the events can be restored when system is restarted. For this purpose, the event and trigger information is inserted into the tables created in database. For our example the tables will have the following information after the trigger processing is completed.

Table 6.9. SYSECATRIGGER

Dbname	Username	Trname	Trigproc	Timestamp	Event	Trtype
Ecadb	Logineca	T_addw	T_addw_proc	Current	Addw	Normal

Table 6.10. SYSPRIMITIVEEVENT

Dbname	Username	Event	Table	Op	Beaf op	Timestamp	vno
Ecadb	Logineca	Addw	weather	Insert	None	Current	0

6.1.2 Execution phase

This section explains the changes that happen when the user gives a DML statement and the trigger that is associated with the event is executed. For example, consider

Insert into WEATHER values (‘arlington’,’getdate()’,72,10)

If the user gives the above insert statement, it goes to the ECA server, which routes it to Sybase server via the JDBC call module. This insert statement will raise the already registered trigger T_ADDW.

Trigger Execution

When trigger T_ADDW is triggered it executes the DML statements inside it. Now we explain every statement in the trigger with our example.

1. To keep track of the number of occurrences of the primitive event (addw) – we update the version number in the SYSPRIMITIVEEVENT table.

Update SYSPRIMITIVEEVENT set VNO=VNO+1 where EVENTNAME='addw'

Table 6.11. SYSPRIMITIVEEVENT

Dbname	Username	Event	Table	Op	Beaf op	Timestamp	vno
Ecadb	Logineca	Addw	weather	Insert	None	Current	**1**

2. We have a global version number in the VERSION table for dealing with the composite events. Hence we update that by:

Update VERSION set VNO=VNO+1

Table 6.12. VERSION

VNO
1

3. We insert tuples into R_inserted and R_deleted tables. For the current example it is inserted from WEATHER and VERSION to weather_inserted. This is done so that the user can access the parameters of the primitive events while interested in composite events.

Insert into WEATHER_inserted select inserted.city, inserted.time, inserted.tem, inserted.wspeed, VERSION.VNO from inserted, VERSION

4. The occurrence of the event in Sybase has to be notified to the JavaLED. This is done by thread running in the ECA Server. The thread checks for the presence of any tuple in the NOTIFY table. For this we insert the corresponding values in the NOTIFY table as part of the trigger.

Insert into NOTIFY select EVENTNAME, TABLENAME, VERSION.VNO from SYSPRIMITIVEEVENT, VERSION where EVENTNAME='addw'

Table 6.13. NOTIFY

Event name	Tablename	VNO
Addw	WEATHER	1

5. As discussed earlier in chapter 4, to give access to the user to inserted and deleted table from stored procedure, we use R_inserted_temp / R_deleted_temp tables.

Delete from WEATHER_inserted_temp

Insert into WEATHER_inserted_temp select inserted.city, inserted.time, inserted.tem, inserted.wspeed from inserted

Delete from WEATHER_deleted_temp

Insert into WEATHER_deleted_temp select deleted.city, deleted.time, deleted.tem, deleted.wspeed from deleted

6. The stored procedure is executed by the exec call. This causes the insertion of the user action portion into the INFO table.

exec T_ADDW_PROC

Table 6.14. INFO

Name	Designation
Sharma	Professor

JavaLED

This section explains the use of JavaLED. When the tuple is inserted into NOTIFY, the thread in the ECA server gets this information. Then it gets the event handle for the event and inserts the information as parameters for the event node. Then it calls the raiseBeginEvent method for that event through the API provided by the JavaLED. Now the LED will raise the event —addw in our example.

After raising this event the result of this insert and the raise event notification is send to the client via ServeOneClient module of the ECA server.

6.2 Repeat Triggers

When we define a trigger on an already existing event, we call the trigger as repeat trigger and the event as repeat primitive event. For example,

Create trigger t1_addw event addw as

Begin

Insert into INFO values (‘Aravind’,’GTA’)

End

The explanation is similar to the primitive triggers except for a few changes.

6.2.1 Creation phase

When this create trigger command comes to Language Filter it recognizes it as the repeat trigger as there is no table name following the event name and hence sends it to Repeat Primitive Event Parser module of the ECA parser.

The Repeat Primitive Event Parser checks if the primitive event is already present by checking the event names in the SYSPRIMITIVEEVENT. If present, it also gets the operation and table name for that event. This info along with the trigger is send to Create Trigger Repeat of the Persistent Manager. It waits for the result. If there is no error, it sends the command to **Generate Persist Code Repeat** of the Persistent Manager.

Create Trigger Command Repeat module is a sub-module under the Persistent manager that gets the create trigger command, information about the table name and operation from the ECA parser. Now this module gets the names of the stored procedure associated with the event (addw in our example). After this, as in the Primitive trigger case, the user action portion is wrapped as a stored procedure.

For our example the trigger will look like this,

```
Create trigger T_ADDW on WEATHER for Insert as
Begin
update SYSPRIMITIVEEVENT set VNO=VNO+1 where EVENTNAME='addw' update
VERSION set VNO=VNO+1
insert into WEATHER_inserted select inserted.city, inserted.time, inserted.tem,
inserted.wspeed, VERSION.VNO from inserted, VERSION
insert into NOTIFY select EVENTNAME, TABLENAME, VERSION.VNO from
SYSPRIMITIVEEVENT, VERSION where EVENTNAME='addw'
delete from WEATHER_inserted_temp
insert into WEATHER_inserted_temp select inserted.city, inserted.time, inserted.tem,
inserted.wspeed from inserted
delete from WEATHER_deleted_temp
insert into WEATHER_deleted_temp select deleted.city, deleted.time, deleted.tem,
deleted.wspeed from deleted
exec T_ADDW_PROC
exec T1_ADDW_PROC
End
```

This trigger is sent to Sybase and the result is send to ECA parser.

As the event name is already present in the SYSPRIMITIVEEVENT table, this module inserts trigger information into the SYSECATRIGGER table.

For our example the table will look like,

Table 6.15. SYSECATRIGGER

Dbname	Username	Trname	Trigproc	Timestamp	Event	Trtype
Ecadb	Logineca	T_addw	T_addw_proc	Current	Addw	Normal
Ecadb	Logineca	T1_addw	T1_addw_proc	Current	Addw	Normal

The result of the successful creation of the trigger is send to the client. In this case, there is no need to create a java file as the event is already registered with the event.

6.3 Drop Trigger

This module in the ECA server is responsible for dropping triggers and the events associated with it. When the drop trigger command comes to the ServeOneClient module, it sends the command to the Language Filter. The Language filter sends the input command to Drop Primitive Trigger module of the Drop Trigger module after checking the presence of the event in the SYSPRIMITIVEEVENT table.

6.3.1 Dropping Primitive Trigger

The first check is made in the SYSDROP table to see if it is a constituent event of any composite event. If so, an error message is to the client. If not, the number of rows having the same event name is checked. If it is greater than one it is a repeat trigger or else it is a primitive trigger. The next step is to drop the stored procedure associated with the user given drop trigger. Then the drop trigger command as given by the user is given to the Sybase SQL server through the JDBC. The tuples storing the information about this trigger is deleted from the SYSECATRIGGER and SYSPRIMITIVEEVENT tables. Then the files associated with this event are also deleted. To delete the node permanently from the Java LED, the Delete Event API of the LED is used.

6.3.2 Drop Repeat Trigger

If the SYSECATRIGGER has more tuples corresponding to the event name associated with the trigger, then, it is considered as dropping the repeat trigger. As discussed in chapter 4, the trigger has to be re-created now. Re-creation of the trigger is a complicated process, as we do not have the contents of the trigger stored. In order to get the trigger information we get the id of the trigger from the sysobjects—system table of Sybase, with this id we get the text (the actual contents of trigger) from the syscomments—system table of the Sybase. We parse the text into tokens, we remove the exec call corresponding to the drop trigger issued and then we re create the trigger with a different name.

The next step is to drop the stored procedure associated with it and then, remove the corresponding information in the SYSECATRIGGER table. The event still persists with other triggers associated with it; hence we do not delete the tuple containing the information in the SYSPRIMITIVEEVENT table.

6.4 If update clause

This is a special case of trigger support provided by Sybase. Even this trigger can be associated with a primitive event. But the handling has to be done in a different way. This is because we convert the action portion into a stored procedure and hence it cannot include **if update** statements.

The Syntax using the if update clause

```
Create trigger [ owner.] trigger_name event event_name
on [ owner.] table_name
for {insert , update}
as
[if update ( column_name)]
[{and | or} update ( column_name)]...]
Begin
SQL_statements
End
[if update ( column_name)]
[{and | or} update ( column_name)]
Begin
SQL_statements
End ……
```

We convert the user given action portion after the keyword ‘as’ in the trigger portion into a stored procedure to support multiple triggers. If we do the same in this case, the ‘if update clause’ also goes into the stored procedure. But the ‘if update clause’ is supported only through the trigger and not inside stored procedure. Hence the problem becomes serious for ECA Agent system.

The important fact to be noted here is that the primitive event is going to be mapped to the insert or update database event. The ‘if update clause’ and the following SQL statements are just transact SQL statements that are specified be along with the trigger. So the solution to this problem is similar to normal primitive event triggers. We ask the users to give all the SQL statements inside Begin and End. When we get the create trigger command with ‘if update clause’, we associate it with insert or update operation as given in the input and then parse the input completely. We take each SQL statements with the Begin and End keywords and make it into a stored procedure. The rest is same as that of the normal primitive trigger. Similarly while dropping we check the SYSECATRIGGER if it is if update clause trigger. If so, we get the corresponding stored procedures and drop them separately.

This chapter gave a detailed explanation of the primitive event implementation. It also described about the repeat triggers and the special case ‘if update clause’ and finally a detailed explanation about the dropping of the events, primitive and repeat triggers.

7. COMPOSITE EVENTS

The composite events are supported by the ECA Agent system. It is supported transparently in that the body associated with the trigger is executed at the server as if it were a trigger body and the trigger statement extensions to specify composite events are minimal. This chapter deals with the creation and dropping of the composite events.

The syntax of a trigger on a composite event is as follows:

Create trigger trigger_name *event event_name= snoop expression:*

[coupling mode] [parameter context] [priority]

Begin [sp]

SQL_statements

End

We introduce an example and explain the composite events in two phases as we did for primitive events. Example:

```
Create trigger event orw = addw | delw: Immediate 1
Begin
        Insert into INFO values (‘Nellai’,’GRA’);
End
```

7.1 Creation Phase

When this input trigger comes to the ECA—Server, it is to the Language Filter for further processing. The Language Filter recognizes the input as create trigger command and scans it. After finding the snoop expression it sends it to the Composite Event Parser of the ECA Parser.

7.1.1 Composite Event Parser

This parser first checks for the duplicate event name and trigger name by checking in the tables having that information. The next step is to check for the presence of coupling mode, parameter context and priority. In this example, the context is not present hence the default context—RECENT is taken. Similarly, 'IMMEDIATE' is the default coupling mode and the default priority is one.

The parser then creates a sjava file with the event expression and rule sample using the File Handling module and gives it to Java SPP. The SPP takes the sjava file— "c_orw.sjava" in this case—and returns a java file (c_orw.java) and a text file (c_orw.txt) as outputs. The SPP actually converts the snoop expression into necessary JavaLED API calls and also creates the rule API with the rule sample provided.

In our example,

Event orw = addw ^ delw;

is converted into:

EventHandle orw = myAgent.createCompositeEvent(EventType.OR, "orw" ,c_addw.addw ,c_delw.delw);

And,

rule Rule_andw [orw , c_orw.True , c_orw.orwdemoeca , 1 , IMMEDIATE , RECENT];

is converted into :

myAgent.createRule("Rule_orw",orw, "c_orw.True", "c_orw.orwdemoeca", 1, CouplingMode.IMMEDIATE, ParamContext.RECENT);

The remaining commands in the sjava file are returned as it is in the java file. The text file contains the names of the constituent events. In our example, it will have addw and delw.

Now the ECA parser reads the text file and checks for the existence of those events in the SYSPRIMITIVEEVENT and SYSCOMPOSITEEVENT table. If they do not exist the error is to the ECA client. If not, the tuples necessary for insertion into the SYSCONTEXT, R_inserted_tmp and R_deleted_tmp are added to the java file. After this the user action portion is also entered into the java file. The control is given to the File Handling module.

Also to handle the drop trigger on a composite event correctly, we need to keep track of constituent events to check whether they are also used as constituent event by another composite event. In order to check for this, we add a tuple to the SYSDROP table, once we check the existence of the constituent events from the text file.

Table 7.16. SYSDROP

CONSEVENTNAME	CONTEXT	COMPEVENTNAME
Addw	RECENT	Orw
Delw	RECENT	Orw

7.1.2 File Handling Module

This module creates the complete java file with the user action portion and compiles it to create class file using Runtime Class in java. It is similar to executing the javac command in the command line. It loads the resultant class file using the Class Loader class and then invokes the method that has the API to create the composite event node and the rule in the Java LED.

7.1.3 Persistent Manager

After the class file is generated, information about the events has to be persisted in the tables created. For our example, assuming the addw and delw events are already created, the tables will look as follows:

Table 7.17. SYSECATRIGGER

Dbname	Username	Trname	Trigproc	Timestamp	Event	Trtype
Ecadb	Logineca	T_addw	T_addw_proc	Current	Addw	Normal
Ecadb	Logineca	T_delw	T_delw_proc	Current	Delw	Normal
Ecadb	Logineca	T_orw	None	Current	Orw	Normal

Table 7.18. SYSCOMPOSITEEVENT

Dbname	Username	Event	EventDesc	Timestamp	Context	Coupling	priority
Ecadb	Logineca	Orw	Addw|delw	Current	Recent	Immediate	1

7.2 Execution Phase

This is similar to the execution phase of the primitive trigger. Consider the example:

Insert into WEATHER values (‘dallas’, getdate (), 86,8)

When this command is given to Sybase through our ECA Server, it executes the primitive event trigger for event addw and does the steps explained in chapter 6. Then the control goes to the LED, it raises the event addw, thus raising the orw composite event. This will execute the action portion, which is a java method. This has a call to the user defined action portion. Hence the action will insert the tuple in INFO table.

Table 7.19. INFO

Name	Designation
Sharma	Professor
Nellai	GRA

The method also has other manipulations as follows.

1. First thing is to delete the contents of the SYSCONTEXT table and then insert the new information.

Table 7.20. SYSCONTEXT

Context	Tablename	VNO
Recent	WEATHER	1

2. It deletes the contents of R_inserted_tmp and R_deleted_tmp and then inserts the new values with the SYSCONTEXT information.

3. It also inserts information in NOTIFYCOM table to print the result out to the client

Table 7.21. NOTIFYCOM

Event name	Info
Orw	Orw recent 1

This result along with the notification of the raise of primitive and composite event is to the client.

7.3 Java file

The java file contains the API to create the composite event node in the LED and the rule associated with it.

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

EventHandle orw = myAgent.createCompositeEvent(EventType.OR, "orw" ,c_addw.addw ,c_delw.delw);

myAgent.createRule("Rule_orw",orw, "c_orw.True", "c_orw.orwsybaseeca", 1, CouplingMode.IMMEDIATE, ParamContext.RECENT);
```

Figure 7.1. Code For Java LED.

1. We initialize the ECA Agent with username_databasename and get a named ECA Agent instance.
2. We use that instance and create the rule and the node in the Java LED.

3. The Create node API has event type, event name, left event and right event names as parameters

4. The rule is the one that is executed when the event is raised and the given condition evaluates to be true—it has a rule name, event handle, condition name, action name, priority, coupling mode and parameter context.

Here the condition method is also created inside the file. In this case there is no condition, hence the method will always return true. The action method is what the user gives along with our DML statements explained above.

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

Figure 7.2. Condition Method.

7.4 Stored Procedures

There is an option in the syntax of create composite event trigger to make all the actions as a stored procedure. To do this the user has to specify the option ***sp*** after the ***begin*** statement in the trigger syntax.

The priority in the composite event syntax can be any positive integer with 1 being the lowest priority. This number is given for the rules associated with every node in the Java LED. The LED executes the rules on the basis of the priority using a thread. So if they were of different priorities they would be executed in a proper order (in the order of higher to lower priority) one after the other. If several triggers have the same priority, LED will execute them concurrently. They may give unexpected results, as we do not have any control over the order of thread execution. Hence to get control over the execution and get the expected results, we give the option of making all the SQL statements in the action of the composite event into a single transaction. The easy way to do this is to specify the 'sp' option. If the user gives this option the ECA Agent will convert the action portion into a stored procedure. Everything else remains the same. The dropping of the composite event is taken care in the normal way as described in the next section. The Agent takes care of the stored procedure created and handles its drop. .

7.5 Drop Composite Event

When the user gives the drop command, it comes to the ECA server. The server routes it to the Language Filter module. The Language Filter sends it to the Drop Composite Trigger of the Drop Trigger module.

First it checks the SYSDROP table if it is a constituent event of some other composite event. If so it sends an error message, else deletes that information from the table. Then it checks if there is some procedure associated with it. If so, it gets the name of the procedure and drops it. Then it deletes the composite event information from the SYSCOMPOSITEVENT table and the trigger information from the SYSECATRIGGER table. The next step is to delete the files associated with it—java file, sjava file, class file and the text file. After this the node has to be deleted from the Java Led for this, the named ECA agent instance is obtained and the APIs provided are used to delete the rule and then the node.

The successful drop message is to the client from the ECA server via ServeOneClient module.

7.6 Composite events in JavaLED

Every time an event is defined and the API for registering the event is called, the JavaLED creates a node for that event with the specified unique name. This node can hold parameters when the insert parameter API is called. With each composite event, rules (or trigger body in this case) are associated along with other information such as parameter context, coupling mode and priority. The nodes associated with primitive and composite events form an event graph.

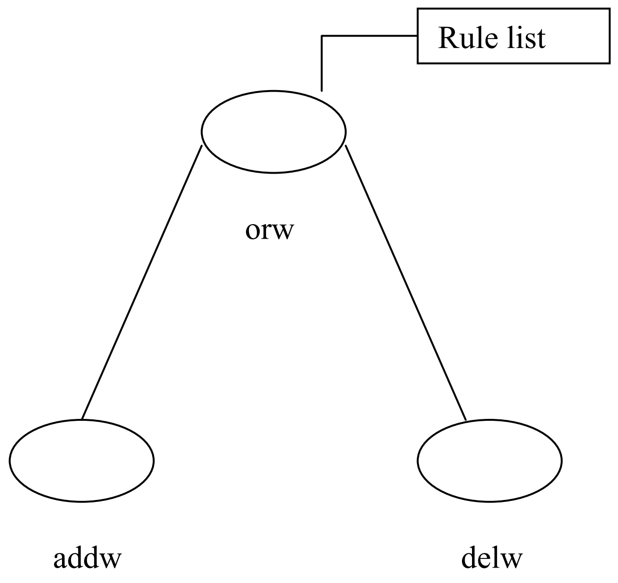

Figure 7.3. Event Graph.

This chapter explained in detail about the implementation of the composite event and gave a detailed explanation in two phases—while the input is given for creation and while the trigger is executed. The description also covered the stored procedure option in composite events and the dropping of the composite trigger and the corresponding event.

8. CONCLUSIONS

8.1 Conclusions

This thesis discusses a generalized approach to make Sybase fully active using ECA rules. The approach uses JDBC, the industry standard—for connecting to the Sybase SQL Server. Hence the same system can be used for supporting active capability by connecting to other RDBMSs such as Oracle, DB2 or Informix with minor modifications to the system. The ECA Agent system has made Sybase more powerful by adding active capability to it. It also provides user transparency. The important fact is that the enhancement has been achieved with out changing the native behavior of the underlying system. The syntax and semantics described are also similar to the native SQL syntax. The user continues to interact in SQL with an extended syntax only when additional capability not supported by Sybase is required. Otherwise, all applications are backward compatible and do not require any changes.

8.2 Contributions of this thesis

1. An agent based approach for enhancing the native active capability of Sybase.
2. Architecture, design and implementation of ECA agent system for Sybase.
3. Support for primitive, repeat primitive events and composite events.
4. Support for multiple triggers in Sybase.
5. Dropping multiple triggers and events.
6. Multi user—multi database connectivity for the ECA Agent system.
7. Persistence and retrieval of the events using the underlying RDBMS.

8.3 Future Work

The system can be extended in several ways:

1. Support for other coupling modes—Deferred and detached.
2. Global monitoring Environment—In the current scenario the events defined are specific to a user-database combination. This can be converted into a monitoring environment by allowing some clients to register as monitoring clients for specific situations that interests it. These clients can notify other normal clients if some changes occur on the interested events.
3. Making Sybase participate in a distributed application environment so that it can subscribe to or send events to other applications.
4. Sybase triggers are not SQL3 complaint. It lacks many features defined in SQL3 standard such as before or after triggers, statement or row level triggers. Hence the future work can be to extend it to make it support all those features.

REFERENCES

1. http://java.sun.com/docs/books/tutorial/jdbc/index.html, *JDBC Tutorial*.

2. http://java.sun.com/docs/books/tutorial/networking/sockets/, *Java Sockets*.

3. http://java.sun.com/products/javabeans/infobus/, *Infobus*.

4. Stonebraker, M., Hanson, E., and Hong, C. H. *The Design of the Postgres Rule System*. in *3rd International IEEE Conference on Data Science*. 1987.

5. Dayal, U., Blaustein, B., A. Buchmann, S. Chakravarthy, *The HiPAC project: Combining active databases and timing constraints*, in *ACM SIGMOD*. 1988. p. 51-70.

6. Gehani, N. and H.V. Jagadish, *Ode as an Active Database: Constraints and Triggers*, in *Proc. 17th Int'l Conf. on Very Large Data Bases*. 1991: Barcelona. p. 327-336.

7. O.Diaz, N.P.a.P.G. *Rule Management in object oriented databases: A uniform approach*. in *Seventeenth International Conference on Very Large Data Bases*. 1991. Barcelona, Spain.

8. Hanson, E.N. *Rule Condition Testing and Action Execution in Ariel* in *ACM SIGMOD 1992 International Conference on Management of Data*. 1992. San Diego, California.

9. Chakravarthy, S.a.M., D, *Snoop: An Expressive Event Specification Language for Active Databases*. 1993, University of Florida: Gainesville.

10. 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.

11. Krishnaprasad, V., *Event Detection for supporting active capability in an OODBMS: semantics, architecture and implementation*. 1994, University of Florida.

12. S.Chakravarthy, V.K., E. Anwar and S.K.Kim,. *Composite Events for Active Databases: Semantics, Contexts and Detection*. in *Proceedings International Conference on Very Large Databases*. 1994.

13. Lee, H., *Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing*. 1996, University of Florida.

14. Widom, J. and S. Ceri, *Active Database Systems: triggers and rules for advanced database processing*. 1996: Morgan Kaufmann Publishers Inc.

15. Li, L., *An agent-based approach to extending the native active capability of relational database systems*. 1998, University of Florida.

16. *Sybase® Adaptive Server™ Enterprise Transact-SQL® User's Guide*. 1999.

17. Dasari, R., *Events and rules for java: Design and implementation of a seamless approach*. 1999, University of Florida.

18. Kim, Y., *A generalized active agent system for extending the active capabilities of a RDBMS*. 2000, University of Florida.

19. Song, Z., *A generalized method to extending the active capability of relational database systems*. 2000, University of Florida.

20. 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.

21. Subramaniam, N., *A Mediator-based approach to support ECA rules in DB2*. 2002, The University of Texas at Arlington: Arlington.

BIOGRAPHICAL INFORMATION

Ganesh A Gopalakrishnan was born January 3, 1979 in Tirunelveli, India. He received his Bachelor of Science degree in Computer Science and Engineering from University of Madras, India in August 2000. In the Fall of 2000, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science and Engineering from The University of Texas at Arlington, in August 2002. His research interests include active databases, data mining and data warehousing.