AN AGENT BASED APPROACH FOR EXTENDING THE TRIGGER CAPABILTY OF ORACLE

By

Yogesh Aravind M.G.

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2002

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Sharma Chakravarthy, for his guidance and support, and for providing an opportunity to work on this project. I would also like to thank Dr. Mohan Kumar and Dr. David Kung for serving on my committee.

I would like to thank Nellainayagam and Ganesh Ambalavam for the valuable discussions we had during the course of my research work. I would also like to thank Pratyush Mishra for maintaining a well-administered research environment and for being helpful at times of need. I am thankful to my friends at ITLAB, Nishanth Vontela, Sreekath Thirunagari and Raman Adaikkavalan for their valuable tips and inputs during the various discussions.

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

Finally, I would like to thank my parents for their patience, their support and their encouragement without which this work would not have been possible.

Dedicated to my parents
and my family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

AN AGENT BASED APPROACH FOR EXTENDING THE TRIGGER CAPABILITY OF ORACLE

Publication No._________

Yogesh Aravind Mysore Ganesha Rao, M.S.

The University of Texas at Arlington, 2002

Supervising Professor: Sharma Chakravarthy

Relational DBMSs currently support the notion of triggers on insert, delete, and update operations. Triggers provide a primitive form of active capability to monitor changes to the database state and take (or trigger) actions without user/application intervention. Expressiveness of these triggers vary among relational DBMSs as there is no standardization.

ECA (or event-condition-action) rules have been proposed as an expressive way to support active capability. Triggers, currently supported by a relational DBMS, form a subset of generally accepted active capability. The events in ECA rules can be quite expressive and include composite events as well as parameter contexts and coupling modes. Snoop, developed as an event specification language provides a number of event operators.

In this thesis, we develop an agent-based approach for supporting full active capability using ECA rules for Oracle. This approach allows us to use the native trigger capability and augment the DBMS with triggers on composite events, parameter contexts, and coupling modes. The approach is general in that it can be used for other relational DBMSs as well. The advantage of this approach is that it transparently supports active capability and preserves compatibility with existing applications.

This thesis describes the architecture of the agent and the functionality supported by the resulting system. We use some of the components with minor changes (e.g., the local event detector and the pre-processor) that have been developed earlier for supporting active capability with Java applications. The agent has to deal with some of the Oracle features that are not common to other DBMSs. The multi-user and multi-database capability is preserved to allow multiple clients to use active capability using a single agent. This thesis also highlights some of the difficulties encountered with the use of JDBC as part of the agent.

CHAPTER 1

INTRODUCTION

Over the past two decades, DBMSs traditionally referred to as passive DBMSs, have evolved significantly to meet the increasing requirements of a wide variety of applications. In spite of these advances, they still fall short of expectations in terms of their need to react to certain specific situations without user or application intervention. Most of the new developments in database technology over the past few years represent real world situations that are to be monitored and provided with an appropriate reaction automatically; i.e., without user or application intervention. An Active DBMS (ADBMS) is one that can continuously monitor situations (specified to the DBMS) to initiate appropriate actions in response to database updates and the occurrences of particular events automatically. Frequently mentioned examples of applications that require DBMSs to have active capability are Computer Networking, Computer-Integrated manufacturing, Workflow and process control, hospital environment etc. In all these applications, as we mentioned earlier, an ADBMS keeps monitoring the situation continuously and provides the appropriate response.

Making a DBMS active is based on the idea of supporting ECA (Event Condition and Action) rules. Conditions are evaluated when an associated event occurs and if it evaluates to true, the action is executed. These ECA rules are stored as a part of the database. Active database systems, based on rule definition, event detection, and action execution not only behave as traditional passive databases, but they are also able to recognize specific situations (in the database and external to the database) and react to them.

Let us now digress for a moment to understand the concept of ECA rules, which we are going to elaborate upon in the forthcoming chapters. ECA rules, as the acronym implies, are composed of three main components. An Event, one or more Conditions and an Action. An Event is an indicator of a happening, which can be either primitive or composite; a primitive event corresponds to a change of state caused by a database operation (such as insert, delete or update) or a method invocation, while a composite event corresponds to a combination of these primitive events. The condition can be a simple or complex query based on the existing database states or a set of data objects. Actions specify the operations to be performed when an event has occurred and the condition evaluates to true. Once the ECA rules have been specified declaratively, it is the responsibility of the DBMS to monitor the situations and trigger the rules.

We will proceed now by taking a look at some of the recent research prototypes that have been developed, and the approaches that have been adopted to develop these systems. A number of research prototypes have been developed over the years: HIPAC [1], Ariel [2], Sentinel [3], Starburst [4], POSTGRES [5], SAMOS [6] etc. Most of these systems have been developed either from the scratch or integrated directly into the Kernel of the DBMS. The integrated approach provides us with the following advantages: [1]

- Does not require any changes to database applications.
- DBMS is responsible for optimizing the ECA rules.
- DBMS functionality is extended.
- Better modularity is achieved leading to easier maintenance.

The implementation of the integrated approach, however, presents us with a much larger problem i.e., the requirement to have access to the internals of a DBMS into which the active capability is integrated. This requires us to have access to the source code, making the cost of developing an integrated system very high. This is accompanied with the additional problem of increase in the integration time. Hence, most of the integrated systems are research prototypes.

There are also other alternative approaches to support Active capability, such as embedding situation check in the application code and polling. However these approaches have their own limitations. For example, the embedded (situation monitoring) approach requires extra code in all the applications. Since modularity is compromised, management of the applications and hence their maintenance becomes very difficult. Also, constraints and business rules are not clearly separated from the application [1]. We elaborate more on polling in Chapter 2.

Keeping these problems in our perspective, the challenge that lay in front of us was to work towards turning a commercial database system into a true Active database system without making any changes to the underlying system. Some   work has been done in this direction. Lijuan Li [7], in her thesis, has used the mediated approach to add a mediator to the Sybase SQL server. Preliminary works of Zecong Song [8] and Younghun Kim [9] have proved that we can, as a matter of fact design and develop a mediator agent that can be ported across platforms and made to work conveniently with any DBMS.

Although there are advantages related to the integrated approach for achieving active capability, the use of a mediator outside the SQL server provides the following benefits:

1. Transparency: The clients have no knowledge of the existence of the ECA Server.
2. System functionality: None of the existing system functionality would be lost.
3. Extensibility: A more distributed architecture can be designed.
4. Scalability: The architecture is scalable and with minor modifications can be tailored to meet the specifications of various applications.
5. Portability: Once the mediator has been developed, it can be ported across platforms and used with any RDBMS.

Problem Statement

In this thesis, we have used the mediated approach for implementing the active capability. We have used Oracle as the test DBMS. It has a very limited trigger capability. It can detect primitive operations like Insert/Delete/Update and a few other basic operations. When any of these operations are detected by the RDBMS, it takes actions specified by the user after testing certain conditions (which is actually optional). However, its ability to monitor the occurrences of a combination of these primitive events (which typically constitutes to the occurrence of a composite event in a RDBMS context) or their occurrences over a period of time is totally non-existent. In our thesis, we try to overcome this limitation of Oracle by using the mediator approach.

The development of Java Local Event Detector (Java LED) by the Sentinel group was one of the reasons, which motivated us to adopt the mediator approach. The Java LED is an application that can be used for the detection of both primitive and composite events. The challenge that we faced was to adopt Java LED in such a way that it would detect primitive and composite events in an RDBMS context. Even more important was to somehow save the information pertaining to the user's actions (like creating a primitive or composite event) so that the user could continue without having to redo anything that was done during the previous session. What this means is this; the LED makes use of a data structure (event graph) for detecting the primitive and composite events created by the user. However, the event graph exists only in the main memory. It ceases to exist once the user logs out. Now, when the user logs in again, the event graph has to be re-created so that the user need not create the events again. So, while correlating RDBMS operations to events recognized by the LED constituted to a part of the problem, the other problem was to persist information pertaining to the creation of these events and use them appropriately so that the user is provided with continuity.

The contributions of this thesis are:

1. We present a generalized approach that significantly extends the functionality of Oracle.

    - Clients are allowed to create ECA triggers as well as normal triggers.
    - Clients can create multiple triggers as allowed by the DBMS.
    - Clients can create triggers on both primitive and composite events.
    - Clients can create repeat triggers on composite events.
    - Triggers associated with both primitive and composite events can be dropped.
    - Events can be persisted into the database.

2. We discuss certain issues related to use of triggers in Oracle under different situations.
3. We present the architecture of the ECA Agent.
4. We discuss the generalized approach, compare it with other approaches and make an inference about its appropriateness.

The remainder of the thesis is organized as follows. In Chapter 2, we give an overview of the related work. In Chapter 3, we present the design and implementation of the ECA Agent. We move on to show how the primitive events and repeat-primitive events have been implemented in Chapter 4. We also talk about the issues concerning the dropping of a primitive event trigger in the same chapter. In Chapter 5, we deal with the implementation issues concerning composite and repeat-composite events. Continuing on the same note, we explain the process of dropping a composite event trigger in the same chapter. We conclude and indicate future work in Chapter 6.

CHAPTER 2

RELATED WORK

Several research projects have focused on providing a DBMS with Active capability. In this chapter, we present a summary of a few of these research prototypes.

2.1 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 [10]. 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 [11] 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.

2.1.1 Ariel query and rule language

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 [11]. 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 ruleset-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 Starburst

The Starburst system was developed at the IBM Almaden research center. It incorporates active capability into a relational DBMS by following an integrated approach. One of the main features of Starburst is that it has an integrated active database rule processing facility called the Starburst rule system.

2.2.1 Syntax of the rule language

The syntax for creating a rule in the Starburst rule definition language is as shown below:

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

The implementation of Starburst relies heavily on three extensibility features of the Starburst database system: attachments, table functions and event queues. Along with the core capabilities of rule management and rule processing, the implementation includes considerable infrastructure for program tracing, debugging and user interaction. The Starburst system also provides a rule set facility for application structuring, and rule processing within transactions in addition to automatic rule processing at the end of each transaction. Rule processing is completely integrated with database query and transaction processing, including concurrency control, authorization, rollback recovery, and error handling.

2.3 SAMOS

In the SAMOS project, active capability is provided to an Object-oriented database management system. It follows a layered architecture; that is, all components implementing the active behavior are built on the top of the underlying passive DBMS, ObjectStore, which is left unmodified. It was one of the earlier research projects, which focused on the definition of a powerful easy-to-use event definition language and has an expressive event definition language.

SAMOS uses colored Petri-nets for the detection of composite events. Besides supporting event detection, Petri-nets allow the precise specification of the semantics of complex events from the appropriate Petri-net structure [12].

2.4 Sentinel

Sentinel was a project wherein effort was made towards providing an Object-Oriented DBMS with active capability. It was developed at the University of Florida. It uses the Open OODB toolkit developed at the Texas Instruments as the underlying DBMS. At the time of developing Sentinel, the following design considerations were taken into account in order to provide the DBMS with an active capability:

- Provide support for both primitive events as well as composite events using a small set of event operators.
- Provide support for a broad set of event consumption modes to meet the requirements of composite events for a large number of applications.
- Allow rules to be triggered by events spanning several objects.
- Provide a uniform mechanism for associating rules to all instances of a class as well as individual instances of one or more classes.

At the time of developing Sentinel, the focus was also on developing a powerful event expression language. Snoop, a language for event specification, was a result of this effort. Snoop provides support for specifying the following composite event operators.

1. AND (Δ): Conjunction of two events E_1 and E_2 denoted as $E_1 \Delta E_2$ occurs when both E_1 and E_2 occur.
2. OR (∇): Disjunction of two events E_1 and E_2, denoted $E_1 \nabla E_2$, occurs when E_1 or E_2 occurs.
3. SEQ (>>): Sequence of two events E_1 and E_2, denoted $E_1 >> E_2$, occurs when E_2 occurs provided that E_1 has already occurred.
4. Not (!): The NOT operator, denoted $!(E_1, E_2, E_3)$ detects the non-occurrence of the event E_2 in the closed interval formed by E_1 and E_3.
5. A: The aperiodic operator, denoted $A(E_1, E_2, E_3)$ expresses the occurrence of the event E_2 in the half open interval formed by E_1 and E_3.
6. A*: This is a cumulative variant of A expressed as $A^*(E_1, E_2, E_3)$. It is useful when a given event is signaled more than once during a given interval, but rather than detecting the event and firing the rule every time the event occurs, the rule is fired only once. A* is detected when E_3 occurs and accumulates the occurrences of E_2 in the half open interval formed by E_1 and E_3.
7. P: A periodic event is defined as an event E that repeats itself within a constant and finite amount of time. It is denoted as $P(E_1, E_2, E_3)$ where E_1 and E_3 are any types of events and E_2 is a relative temporal event. P occurs for every amount of time string of E_2 in the half open interval $(E_1, E_3]$. The time string needs to be positive and should not have any wild card to prohibit continuous occurrences of P.

8. P*: This is a cumulative variant of P and is denoted by P*(E_1, E_2, E_3). P* occurs only once when E_3 occurs and accumulates the time of occurrences of the periodic event whenever E_2 occurs.
9. PLUS (+): Sequence of an event E_1 after a time interval TI, denoted E_1 + [TI] occurs when TI time units are elapsed after E_2 occurs.

In addition to the above, Sentinel also allows composite events to be detected in different contexts. Composite events can be detected in four different contexts. They are:

- Recent: In this context, not all occurrences/instances of a constituent event will be used in detecting a composite event. Only the most recent occurrence of initiator of any event that has started the detection of that event is used. When an event occurs, the event is detected and all the occurrences of events that cannot be the initiator of that event in the future are deleted. Furthermore, an initiator of an event will continue to initiate new occurrences of an event until a new initiator occurs.
- Chronicle: In this context, for an event occurrence, the initiator, terminator pair is unique. The oldest initiator is paired with the oldest terminator for each event.
- Continuous: In this context, each initiator of an event starts the detection of that event and is saved until a terminator occurs. A terminator is paired with every initiator. Thus the terminator may detect one or more occurrences of the same event. The initiator and the terminator are discarded after an event is detected.
- Cumulative: In this context, all occurrences of an event type are accumulated as instances of that event until a terminator occurs. Whenever an event is detected, all the occurrences that are used for detecting that event are deleted.

2.5 An Agent-Based Approach to Extending the Native Active Capability of Relational Database Systems

This was a research project at the University of Florida [7]. This was the first effort towards providing a RDBMS with active capability using a mediator approach. In this project, the Gateway Open Server (GOS) provided by Sybase was used for communicating with the Sybase server and for extending its active capability. Different modules required for rule parsing, event creation and persisting event information were implemented. Remote procedure calls were used for creation of events.

This project was fairly successful in extending the trigger capability of Sybase. However, since all the modules were written to interact with the GOS, it was restricted to providing Active capability to just one RDBMS.

2.6 A Generalized Active Agent System For Extending The Active Capabilities Of A RDBMS

This was a research project at the University of Florida [9]. It was taken up with the aim of developing a generalized mediator agent that could be used with any RDBMS and which would overcome some of the limitations of the earlier work on Sybase. In this project, Java was chosen to be the language for implementation, while Oracle was used as the test bed. Java, being an interpreted language, allows users to create events dynamically, at run time. However, there were several limitations of this project. It supported only a small sub-set of the trigger specifications provided by the vendor. Support for composite events was not complete and it was not possible to detect composite events in different contexts. It did not provide adequate support to drop events. Finally, it did not support repeat composite events.

Looking back, in this chapter, we have presented an overview of some of the related works. Some of the systems presented above used the integrated approach. We also discussed a couple of projects, which used the mediator approach for providing a RDBMS with active capability.

In the next chapter, we present the design of the Generalized ECA Agent (the mediator agent). We also explain how it has been extended in order to support a wider variety of trigger specifications and other features like repeat-composite events and dropping events.

CHAPTER 3

DESIGN AND IMPLEMENTATION OF THE ECA AGENT

In this chapter, we present the design and implementation of the ECA Agent in its present form. We also make a comparison with the older version of the design and explain about certain subtle and significant changes that have been made and the reasons for doing so. In short, we present the revised architecture of the ECA Agent in terms of its components and how they relate to each other.

3.1 Architecture of the ECA Agent

In this section, we present the requirements that contributed to the design of the ECA Agent and follow it up by presenting the design and implementation of the ECA Agent.

An ECA Agent is a stand-alone system that can, in general, be used with any kind of Relational DBMS such as, Oracle, DB2, and Sybase etc. The ECA Agent (which acts as a server termed the ECA Server) sits between the client(s) and the SQL server(s).

Messages from the clients are routed to the SQL server through the ECA Agent. Result from the SQL Server is returned to the client via the same ECA Agent.

The ECA Agent is a multithreaded program. It can process requests from multiple clients and route them to the appropriate server as shown below in Figure 3-1

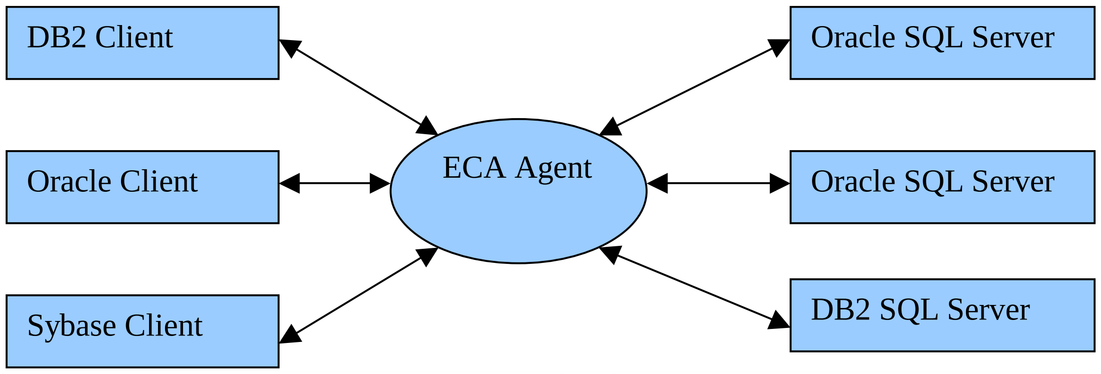

Figure 3-1 (Generalized ECA Agent)

From the perspective of the user, the ECA Agent works just like a virtual SQL server. The presence of the server is totally transparent to the client. . The ECA server augments the SQL Server with Active capability using the basic trigger capability provided by the SQL server.

Before we explain the design issues concerning the ECA Agent, we will take a look at how the ECA Agent can be used to extend the trigger capability and make it applicable in a real world application. Consider a Weather forecast scenario. Consider the following tables, Weather and WindSpeed. The structure of these tables are shown below:

WEATHER: This table stores information about the temperature of a particular city at a certain point of time.

Table 3-1 (Weather table)

CITY	TIME	TEM

WINDSPEED: This table stores the wind speed of a city at a particular point of time.

Table 3-2 (WindSpeed table)

CITY	TIME	WSPEED

Now, consider a third table, WeatherForecast. Every time the WindSpeed table is modified after a modification of Weather table, this table should collect temperature from Weather and wind speed from WindSpeed. Based on the values collected, it should decide whether the Weather is favorable for people to venture out of their homes

The structure of the WeatherForecast table is shown below:

Figure 3-2 (Weather Forecast table)

CITY	TIME	TEM	WSPEED	FAVOURABLE

Oracle provides us with the ability to create triggers on Insert/Delete/Update operations. However, in this present example, we need to monitor a modification of Weather followed by a modification of Wind Speed table (which can be thought of as a sequence operation). This is a feature, which is not supported by Oracle. This drawback of an RDBMS i.e. the inability to monitor a combination of primitive operations (on the same or different table) and take appropriate actions is what we try to overcome using the ECA Agent.

In addition to the drawback just mentioned, Oracle does not support prioritized firing of triggers, nor does it support the notion of detecting events in different contexts.

So, with the help of the ECA Agent, we should be able to associate each DML operation to a unique event name (primitive event). Using the Local Event Detector (LED), the ECA Agent should create a node for these primitive events. Now, if the user wants to monitor a combination of these primitive events, then all that the user needs to do is to specify a composite event trigger. This results in a composite event node being created inside the LED. Whenever a DML operation (Insert/Update/Delete) occurs, if the user has created a   primitive event on that DML operation, a trigger is fired following which, the ECA Agent notifies the LED about the occurrence of the primitive event. The LED registers the occurrences of this primitive event and propagates the same to the composite event node. The LED keeps checking if the occurrences of these primitive events satisfy the conditions for the composite event to be detected (Using LED, composite events can be detected in different contexts and coupling modes). If it is satisfied, then the LED detects the occurrence of a composite event. Following this, the action part specified in the composite event trigger is executed based on priority (illustrated in Appendix).

The previous effort [9] showed the feasibility of a generalized ECA server. Modules for ECA rule parsing and others were implemented. However, only a partial set of trigger specifications were supported. Composite events and other functionality such as drop triggers were not completely supported.

Next, we describe the requirements that contributed to the design of the ECA Agent, the different ECA Agent components, their function and how they interact with one another.

The ECA Agent was designed to perform the following functions:

- It should be able to associate RDBMS operations to primitive events. By being able to detect primitive events, it should further support composite event detection and thus extend the native trigger capability of Oracle.

- It should be able to mimic the multi-user feature provided by the RDBMS i.e. the ECA Agent should allow many users to interact with itself and hence with the SQL server
- Since Oracle does not support the notion of events, we need to pass information about the events being created for a particular operation in the trigger statement. In order to do this, we have extended the SQL trigger syntax provided by the vendor (Oracle) so that event description is included in the trigger statement. So, the ECA Agent should have a module that takes care of extracting the event information embedded in the SQL statement and use it for creating the corresponding events in the LED.
- Information about the creation of events should be persisted. This is needed so that the events created by users are not lost between sessions.
- Apart from this, we need a module that can take care of dropping triggers and events.

The design of the ECA Agent was based on taking these requirements into consideration. In
Figure 3-3, we present the architecture of the ECA Agent.

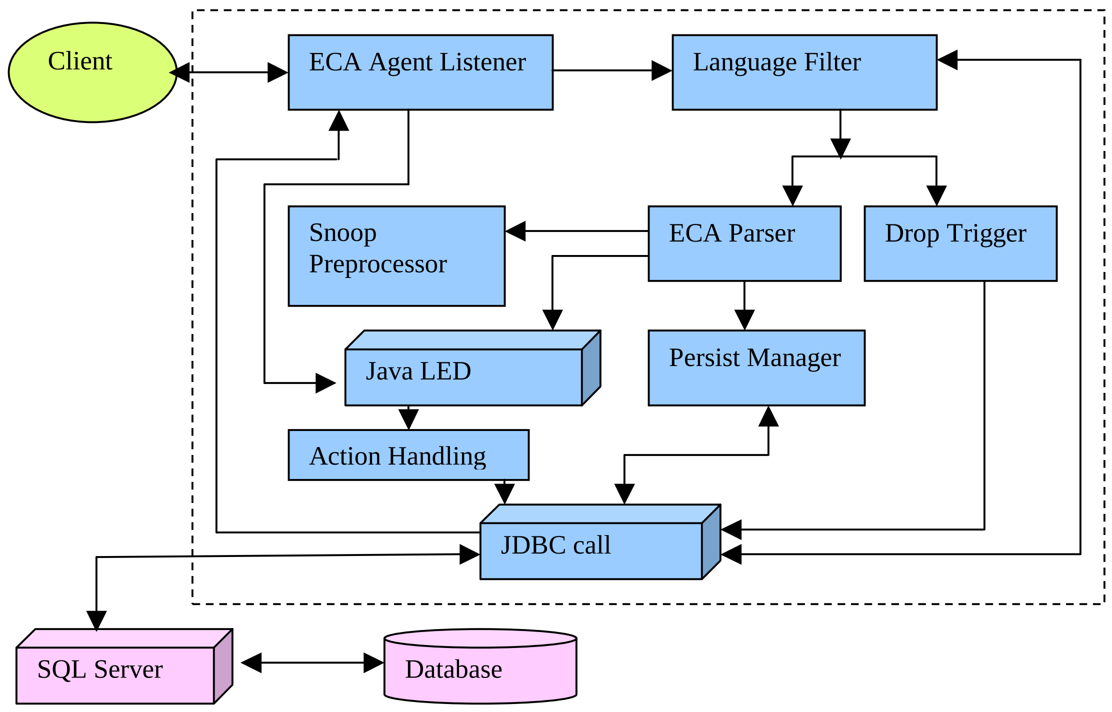

Figure 3-3 (ECA Agent - Architecture)

Some of the subtle changes that have been made as compared to the previous version are as follows: Essentially, we have added a module for dropping the triggers and events. Also, the ability of the client to talk to the SQL server directly has been disabled. In addition to this, we have used System tables for persisting event and trigger information.

We now proceed to explain how the different components of the ECA Agent work towards meeting the requirements listed earlier.

3.2 ECA Agent Listener

This module accepts the input from multiple clients, spawns a new thread for handling each of the new clients and then hands over the input to the Language Filter. This module is also responsible for creating a unique system wide directory structure for each individual user in order to prevent clashes between files (generated for registering events) created by different users and thus allow multi-user capability. This is explained in detail in 3.11.

3.3 Language Filter

This is the module, which filters all the incoming client requests. It checks to see if the request is an ECA command or a pure SQL command. In our thesis, client requests like select statement, creation of a normal trigger are considered as pure SQL commands while client requests that include creating a primitive or a composite event (trigger), dropping a trigger (created on primitive or composite events) are considered as ECA commands.

If the client request is an ECA command, the Language Filter passes this command to other modules of the ECA Agent for further processing. Otherwise, it is sent directly to the SQL Server by making a JDBC call. The ECA trigger command given by the user can fall into one of the following four categories. The words that have been highlighted and italicized indicate the extensions for including event information.

1. A primitive event trigger: A trigger created on one of the primitive operations like Insert, Delete or Update. We can associate a primitive event for each of the different types of triggers. Figure 3-4 shows the syntax for creating a primitive event.

```
Create trigger t_addw event addw after insert on Weather
Begin
   Insert into temp values (‘t_addw”, sysdate);
End;
```

Figure 3-4 (Primitive Event trigger)

2. A repeat trigger being created on a primitive event: Oracle allows users to create multiple triggers on the same operation. In our thesis, we support this feature of Oracle.

Once a primitive event has been created for a particular operation, the rest of the triggers that the user wants to create on the same operation and also associate with the same primitive event are called as repeat primitive triggers. Figure 3-5 shows the syntax for creating a repeat primitive event trigger.

```
Create trigger t1_addw event addw
Begin
   Insert into temp values (‘t1_addw’, sysdate);
End;
```

Figure 3-5 (Repeat Primitive Event trigger)

3. A composite event trigger: This type of trigger is created based on the occurrence of one or more primitive (or composite) events. Figure 3-6 shows the syntax for creating a composite event trigger. Here, we are creating a composite event with name ***seqw*** and it fires whenever two primitive events i.e. ***addw*** (which fires when an insert operation occurs) and ***delw*** (which fires when a delete operation occurs) occur, addw followed by delw.

```
Create trigger t_seqw event seqww = addw >> delw : Immediate Recent 1
Begin
   Insert into temp values (‘t_seqw”, sysdate);
End;
```

Figure 3-6 (Composite Event trigger)

4. A repeat trigger being created on a composite event: The Java LED allows several rules (combination of condition and action based on certain semantics) to be associated with a single event node. Consequently, the user can specify several action portions that get fired when a composite event is detected. In order to have several rules associated

with the node, the user would have to create repeat composite event trigger. The Figure 3-7 shows the syntax for creating such a trigger.

```
Create trigger t1_andW event seqw
Begin
   Insert into temp values ('t1_seqw', sysdate);
End;
```

Figure 3-7 (Repeat Composite Event trigger)

Based on the category of the ECA command, the Language Filter routes these commands to the appropriate module. Similarly, if the client request is a drop trigger command, then it is left to the language filter to figure out if the trigger being dropped is a primitive trigger or a composite trigger and route them to the proper modules.

Figure 3-8 below depicts the function of the Language Filter module.

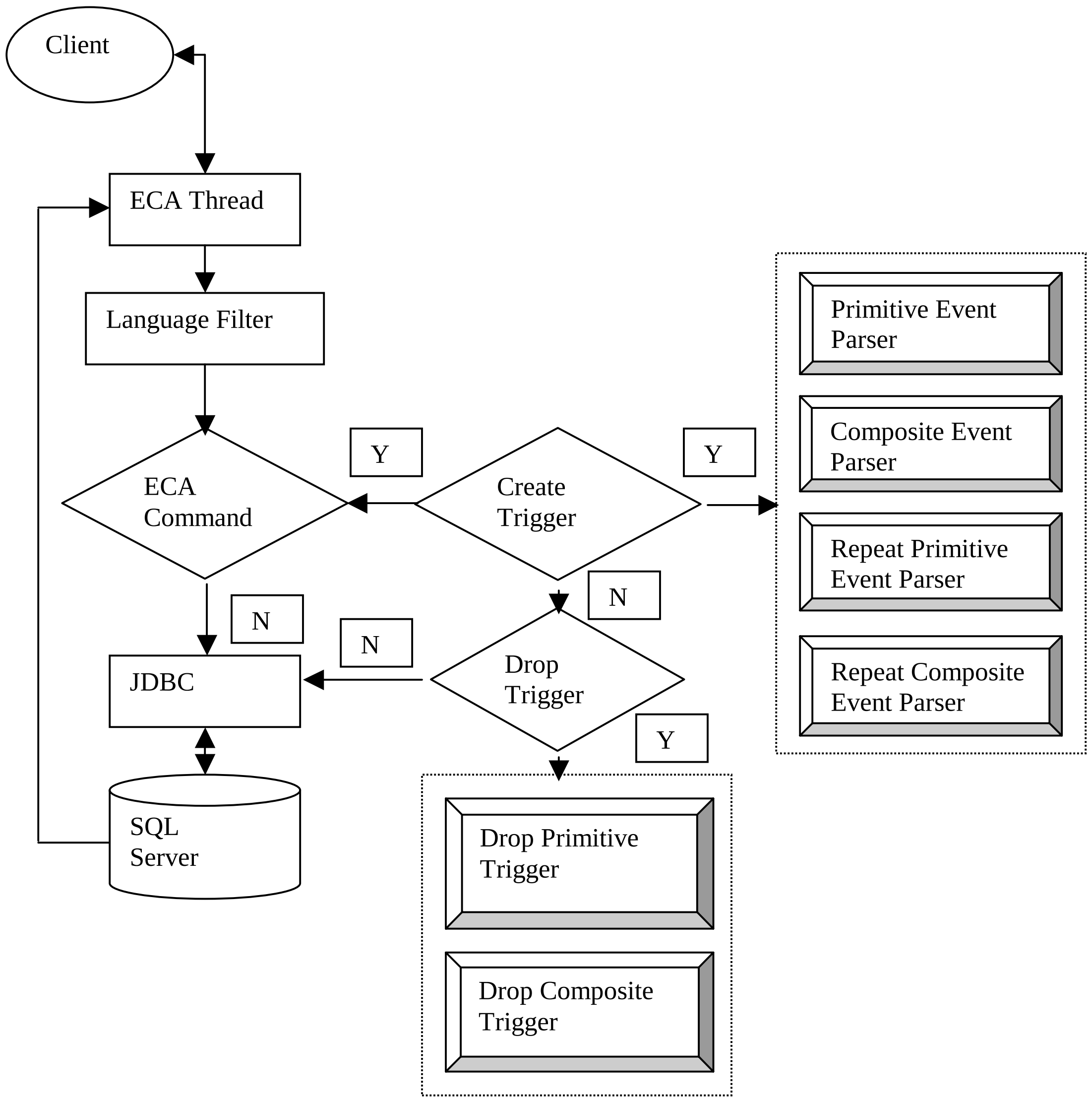

Figure 3-8 (Language Filter – Logical Flow)

3.4 ECA Parser

As explained earlier, we have extended the trigger syntax provided by the vendor (Oracle) in order to include event information. We know that a DML operation in a RDBMS can be associated to an event in Active terminology. So, if the user wants to monitor database operations, triggers are created. These triggers fire whenever the operations (like Insert/Update/Delete) occur. In order to associate the operations with events, we have seamlessly extended the SQL syntax for creating triggers, so that, event information can be included as a part of the trigger syntax. Now, when this modified SQL command is sent to the ECA Agent, it should have a module that extracts the event information and create the corresponding events in the LED. ECA Parser is the module, which handles these functions. In order to create the events at run time, the ECA Parser generates a “.java” file. This file is designed to have a method that can be called in order to register the primitive event with LED. We can use the RunTime class provided by Java in order to get a handle of the run time object associated with the ECA Agent. Using this, we can compile the “.java” file to get the corresponding “.class” file. Following this, we can use the Reflection method provided by Java and invoke the method for registering events at run time. The creation of a “.java” file also serves the purpose of persistence. Thus, if a user logs out and starts a new login session, the ECA Agent can still use the “.java” file (created earlier) for re-creating the events in LED.

Also, once the events are created, we need to create the corresponding trigger in the SQL server. However, since the SQL syntax has been modified, we cannot create a trigger with the same syntax; we will have to remove the event information so that the syntax conforms to the SQL syntax provided by the vendor. In order to do this, the ECA Parser invokes another module called the Persistence Manager.

In Figure 3-9, we show you the extended trigger syntax used for creating a trigger on a primitive event.

```
//A primitive event trigger
create trigger t_addw event addw after insert on weather
Begin
   Insert into temp values (‘t_addw’, sysdate);
end;
```

Figure 3-9 (Primitive Event trigger)

The ECA parser is designed to have four different modules. They are:

1. Primitive event Parser: This module parses the primitive event triggers.
2. Composite Event Parser: This module parses the composite event triggers.
3. Repeat Primitive Event Parser: This module parses repeat primitive event triggers.
4. Repeat Composite Event Parser: This module is responsible for handling repeat composite event triggers.

For example, if the Language filter decides that the given ECA command is a primitive event trigger, it passes control to the Primitive Event Parser. Here, the event information is extracted and an event node is created in the LED. Whenever the operation on which the primitive event has been created occurs, the trigger gets fired and subsequently the LED is informed about the occurrence of this event.

While this constitutes to what the ECA Parser does in order to handle the primitive events, the question still remains as to how composite events are to be handled. Certain drawbacks of the underlying RDMS assume significance. They are

- There is no notion of a composite event in Oracle and as such we cannot associate any single database operation to the occurrence of a composite event.
- When a primitive event is created, the action portion specified by the user is embedded in the trigger that is created. Hence when the trigger fires, the action specified by the user is performed. However, we cannot adopt the same approach for triggers created on composite events because, these composite events are detected outside the scope of the SQL server, in the Java LED. As a result of this, we also need to perform the action specified by the user outside the SQL server.

In view of these restrictions, the ECA Parser does the following in order to handle a composite event trigger. Once control is passed on to the Composite Event Parser, it extracts the event information from the SQL command. Following this, it invokes the appropriate API for creating the composite event node with associated rule(s) in the Java LED. Whenever a primitive event occurs, its occurrence is propagated to the composite event node; Now, LED checks for the associated semantics and fires the rule. From within this rule, ECA Agent invokes the method that performs the action specified by the user.

3.5 Persistence Manager

One of the main problems of using LED for creating and detecting events is that the event graph used by LED for detecting events is valid only for that session (because event graph used by LED for detecting composite events exists only in main memory). Once the user logs out (or the server or the client crashes) the event information is lost. So, we need to persist the events and the rules into the database so that they do not have to be re-defined for each session. Even when the ECA agent crashes, the previous state (event graph) should be re-created based upon what has been persisted in the system tables. ECA Agent creates ".java" file, which can be used for registering the events with LED. However, it should know which events are to be registered with the LED. In order to take the proper decision, the ECA Agent should have access to information about the triggers and events that were created by the user during the previous session. The Persistence Manger is responsible for the task of persisting event information into the database. It makes use of System tables for storing ECA information associated with a trigger/event. It uses the following System tables for persisting the event information.

1. SYSPRIMITIVEEVENT: It stores information about the primitive events.
2. SYSCOMPOSITEVENT: It stores information about the composite events.
3. SYSECATRIGGER: It stores information about the triggers that have been created

In addition to the task of Persisting ECA information, the Persistence Manger also performs the task of creating triggers. As explained in the previous section, the SQL syntax for creating trigger has been extended in order to include ECA information. However, we cannot use the same syntax for creating triggers. In order to overcome this problem, the Persistence Manger extracts the event related information from the SQL command. Once this is done, the ECA Agent appends certain ECA actions to the trigger body. These ECA actions are primarily comprised of updating system tables with the occurrence of the primitive event and collecting parameters needed to be passed for detecting a composite event. This is needed because; we are detecting the primitive events outside the scope of the SQL server. So when a trigger is fired, the ECA Agent should inform the LED about the occurrence of that primitive event by invoking the appropriate API with the proper parameters. These parameters are, event name, table name on which the trigger was defined and finally the version number i.e. the number that corresponds to the occurrence of that primitive event.

In addition to these parameters, the ECA Agent should also collect the transient values (referred as “old” and “new” column values) that exist after a trigger is fired. The reason we need to collect these transient values is that they cannot be accessed outside the body of a trigger statement. But, when a composite event is detected, we need to access the parameters that existed when the trigger(s) fired (i.e. when the primitive event(s) was detected), so that the action portion of the composite event trigger acts upon the proper set of values. For the composite event trigger to have access to transient values that existed as a result of the firing of a primitive event trigger, the ECA Agent creates tables that can hold these transient values along with the occurrence number of the primitive events. We explain these tables in a later section.

The Persistence Manager uses the following System Tables for updating the information when trigger is fired.

1. Version: This table stores the version number of the occurrence of a primitive event.
2. Notify: this table stores the parameters associated with a primitive event occurrence.
3. Notifycom: This table stores information about the occurrence of a composite event.

Once the ECA Action portion and the user action portion are appended to the trigger statements, the trigger is created in the database.

The Persistence Manager is also invoked on certain other occasions. For example, If the ECA Agent or the SQL server crashes and the user starts a new session; Persistence Manager re-creates the events and the rules and resets the System tables to the appropriate values.

3.6 Drop Trigger

In this thesis we have made a conscious effort towards separating the ECA actions that need to be performed when a trigger fires and the actions specified by the user. As a result of this, whenever the user wants to create a primitive event trigger, the ECA Agent creates two corresponding triggers in the database, one to perform user actions and the other to perform ECA actions. Consequently, when a primitive event trigger is dropped, we should check to see if repeat triggers have been created on that event. If there are no repeat triggers then the trigger that performs the ECA actions should also be dropped. Otherwise, only the trigger specified by the user should be dropped.

On the other hand, if a composite event trigger is being dropped, we should only drop the rule associated with that trigger if there are other triggers created on the same event. However, if no repeat composite triggers have been defined on that event, then the user should delete the composite event node from the LED.

The Drop Trigger is capable of dropping both primitive as well as composite [13] triggers. It can also handle triggers that have been created without having any ECA functionality associated with them.

The Drop trigger is made up of the following two modules:

1. Module for dropping Primitive triggers.
2. Module for handling Composite triggers.

3.7 Java Snoop Preprocessor

Snoop [14] is the event specification language that was developed by the Sentinel group. It gives the precise semantics for primitive events and event operators to express composite events. Since Snoop has a well-defined BNF, we use Snoop as the event specification language in our thesis. For a detailed specification of the BNF of Snoop refer to [14]. In addition to providing semantics for expressing events, it also provides support to specify the conditions (like coupling mode, context and priority) under which a composite event needs to be detected.

3.8 Java Local Event Detector

A relational DBMS can only detect primitive events, such as Insert, Update and Delete. In order to enhance the Active capability, we use Java LED so that we can augment the RDBMSs with the capability to detect composite events and execute the rules associated with these events. The LED when used in a stand-alone mode detects both primitive as well as composite events. A primitive event can be either a method call from an application (for non-database applications) or a database update operation. However, when used with the ECA agent, the primitive event detected by the SQL server as the execution of a trigger needs to be conveyed to the ECA agent so that it can invoke the corresponding primitive event on the LED. Composite events can be combination of these primitive events (or other composite events). The composite events can be detected in one of the four contexts based on how the user wants to detect it. The four contexts are: [15]

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

We will now move on to provide a brief glimpse of how composite events are detected in the Java LED. Assume that event e1 corresponds to an Insert operation on a table, e2 corresponds to a Delete and e3 corresponds to an Update. The occurrences of these events are depicted along the time line. The superscript is used to denote the number of occurrences of that same event.

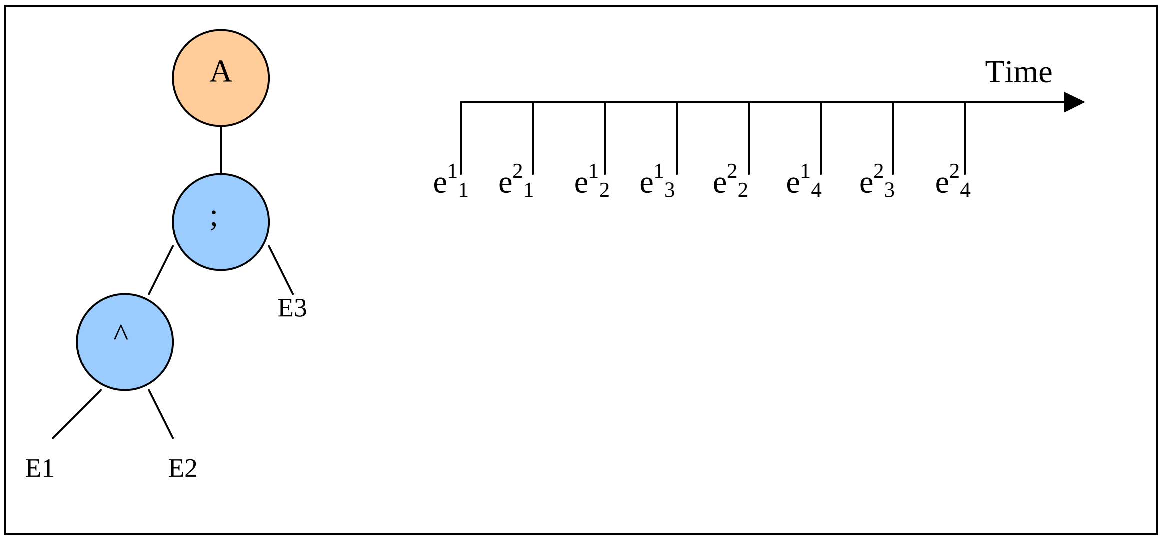

Figure 3-10 (Event Graph in LED)

As shown in Figure 3-10, the composite event that needs to be detected can be defined as (e1 ^ e2) ; e3. In the Recent context, the most recent occurrence of the event is used for detecting the composite event. The composite event A will include the event instances (e^2_1, e^1_2, e^1_3) and (e^2_1, e^2_2, e^2_3). In the chronicle context, the composite event detection will have the following event instances {e^1_1, e^1_2, e^1_3}. In the Continuous context, the first occurrence of A has instances {e^1_1, e^1_2, e^1_3}. The second occurrence of the event A consists of the instances {e^2_1, e^1_2, e^1_3}. In the Cumulative context, all occurrences of the event are accumulated as instances of the event A when A is detected.

3.9 Rule Scheduler

The rule scheduler is the functional module of the Java LED that controls the execution of rules. A rule in the ECA paradigm comprises of a condition and an action associated with an event. The rule scheduler controls the order in which the actions are executed. The ECA Agent spawns a dedicated new thread for each new client that subscribes for its services [13]. Each of these client requests is handled inside the Java LED by the creation of new threads. For multiple rule executions, a number of sub transactions are spawned as a part of the application process. The order of the rule execution is handled by assigning priorities to each of these threads. Rules are executed based on a descending role in priority. Rules are also associated with coupling modes. A rule created with an immediate coupling mode should be executed at the time the event is detected whereas a rule created with a deferred coupling mode should be executed at a later point of time.

3.10 JDBC (Java Database Connectivity)

JDBC is the bridge that connects the ECA Agent to the SQL Server. All the client requests are routed through the ECA Agent. This includes even the information that is sent to the SQL server for authenticating the client. Results obtained from the SQL Server, similarly, are routed back to the client through the ECA Agent through the JDBC Bridge. The process of a client connecting to a SQL Server through the JDBC is comprised of the following steps:

1. Load driver: The first step is to load the JDBC driver. This can be accomplished by using the **forName** static method of the **class** object. The call is made as follows:

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

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

2. Create connection: The next step is to establish the connection. This is accomplished through a call to the **getConnection** method in **DriverManager** class to find a specific driver that can create a connection to the URL requested. The call is made as follows:

```
String url = "jdbc:oracle:thin:@" + sqlhost + ":1521:" + database;
Connection con = DriverManager.getConnection (url, username, password);
```

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

```
Statement stmt = con.createStatement( );
```

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

4. Execute statement (and if (query)) return **ResultSet**: The SQL **Statement** object does not have a specific SQL statement associated with it. The SQL statement to be executed is determined when the call to **executeQuery or executeUpdate** is made, as follows:

```
String qs = "select * from weather";
ResultSet rs = stmt.executeQuery (qs);
```

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

On the other hand, if the statement to be executed is that of a database update, then the format will be as below:

String qs = "delete from stock";

Stmt.executeUpdate (qs);

5. Iterate through the **ResultSet: ResultSet** represents the collection of results from the query. Initially, the result set pointer will be pointing to the header of the result set. So, we should push the pointer so that it points to the first row of the actual result that the user wants. This is achieved by making a call as below:

Boolean more = rs.next();

Now, if the Boolean value is set to false, it means that there is no data to be retrieved. However, if the Boolean value is true, then we infer that data needs to be retrieved, which Can be achieved by the get method.

For example, if we want to retrieve the first column of the first row, it can be achieved by making the following call:

returnstring = rs.getString (1);

We use the same logic to iterate through the rest of the result set.

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

rs.close();

stmt.close();

conn.close();

3.11 Naming Mechanism (Enabling Multi-user/Multi-database)

Most of the commercial RDBMSs support the multi-user/database environment. In order to support such a feature, a DBMS has to have a mechanism by which it recognizes each of these users as a separate entity. Normally, a DBMS achieves this by converting the name of the object on which the user is working into a unique system wide internal name. As an example, suppose the user name is “ecaoracle”, the name of the instance to which the user has access is “orcl” and the table on which the user is working is “Weather”; then the system wide internal name for the table Weather will be “orcl.ecaoracle.weather”.

A similar approach has been used in the design of the ECA server in order to support multi user mode [13]. For example, if user “ecaoracle” creates an event by name addw, then the ECA Agent creates a file by name “addw.java”, which is prefixed with the characters “c_”. Now the file “c_addw.java” is placed in a directory with a unique system-wide name “ecaoracle_orcl” (username_database) where “ecaoracle” is the name of the user, and “orcl” is the name of the database instance. This is similar to the DBMS itself handling multiple users and databases.

3.12 Oracle Trigger syntax

Now, we will take a look at the trigger syntax in Oracle, the types of triggers supported and how these issues have affected the implementation of the ECA Agent. The Figure 3-11 depicts the trigger syntax of Oracle.

```
CREATE [OR REPLACE] TRIGGER <TRIGGER NAME> [INSTEAD OF TRIGGER NAME]
BEFORE | AFTER
DELETE | [OR] INSERT | [OR] UPDATE [OF <column [, <column>…]]
ON TABLE <TABLE NAME>
[REFERENCING [OLD [AS] <old>] [NEW [AS <new>]]
[FOR EACH ROW [WHEN, condition.]]
BEGIN
…
          /*PL/SQL Block*/
….
END;
```

Figure 3-11 (Trigger syntax of Oracle)

As shown in the Figure 3-11, we can broadly classify the triggers into the following types of triggers [Oracle Manual]:

1. Triggers created on the table: These triggers can be created on any of the primitive operations i.e. Insert, Delete or Update. They fire whenever any of these operations occur and modify values in the database.
2. Triggers created on Column Update: These triggers can be created only for update operation. The trigger fires whenever a particular column (of the table on which the trigger has been created) is modified.
3. Triggers created on multiple operations: In this type of trigger, we can create the triggers on multiple operations. The action portion in the trigger statement is executed irrespective of which operation occurs. However, this can be modified by having the action portion inside a conditional block, so that only actions that correspond to a particular operation are executed.
4. Create or Replace triggers: In this type of a trigger, if the trigger with the same name has already been created, then the trigger body is going to be replaced with the new trigger. However, if trigger has not yet been created, then a new trigger is created.

5. Instead of trigger: This provides a transparent way of modifying views that cannot be modified directly using Insert, Update or Delete statements.

In the present prototype, we provide support to the first three types of triggers. In addition to the above classification, triggers can also be classified based on a few other features as follows:

1. Statement level triggers and row level triggers: In a row level trigger, the trigger fires once for each row that has been affected by a database operation. However, in statement level triggers, the trigger fires just once, irrespective of the number of rows affected.
2. Before and After triggers: They specify when to file the trigger body in relation to the triggering statement that is being executed. “After” triggers are more efficient. This is because, with BEFORE triggers, the affected data segment must be read twice, once for the trigger and then again for the trigger statement. However, in AFTER triggers, this needs to be done just once.

Now, let us take a look at how the ECA Agent has been implemented. In our thesis, we have used System tables for persisting event information, which is used for the following purposes:

- Recovery in case of server or client crash
- Re-creating event graph for every user session
- Duplication checks and trigger validation
- Collecting proper parameters that are to be passed for composite event detection.

We start off by describing the structure and format of each of these tables. Then we proceed to explain what information is stored in each of these tables and how this information is used by each component of the ECA Agent.

The system tables explained below are tables maintained by the ECA server on behalf of the clients. The number of tables is designed to be small and contain minimally needed information.

3.13 System Tables

3.13.1 SysPrimitiveEvent

The structure of this table is shown in Table 3-3:

Table 3-3 (SysPrimitiveEvent – Structure)

Db name	User name	Event name	Table name	operation	Beaf operation	Time Stamp	Vno	Column names	Type

This table is used for storing information about the primitive events. Before the trigger from the user is submitted to the SQL server, it is parsed, analyzed, and transformed (if necessary), so that, inside the SQL server, it is persisted as a trigger acceptable to the SQL server and capable of performing the ECA actions, when the trigger is fired.

In the table SysPrimitiveEvent, the first two columns, i.e. the *dbname* and *username,* correspond to the name of the database (i.e. the instance of database to which your account has access) and the login id of the user respectively. The column, *eventname,* is used for storing the name of the event on which the trigger is being created. The column, *operation,* stores one of the three primitive operations (insert, delete, or modify) on which the trigger is defined. The column, *Beafoperation,* stores information about whether a trigger is a BEFORE event trigger or an AFTER event trigger (Oracle supports both). Timestamp stores the time of creation of the trigger while Vno stores the number of occurrences of the event up to that point (it is initialized to 0). The column, *columnnames*, is used to store the names of columns. This is used only in the case of Column Update triggers. For other triggers, this column is padded with NULL values. The column *Type* indicates the type of trigger. It can take one of the of the following values:

N – Trigger created on a table

U – Trigger created on column update

M – Trigger created on multiple operations

MU – Triggers created on multiple operations where one of the operations is an update of column.

The contents of the table SysPrimitiveEvent are used for the following purpose:

- The column eventname is used to authenticate the name of the primitive event. The name of the event has to be unique. So, when the user tries to create a new event, the ECA Agent checks this table to see if the event name specified by the user is a duplicate event name.
- The column version holds the version number of the latest occurrence of a primitive event.
- Inside the database, the table associated with a particular user is identified by appending the name of the database and login id to the name of the table (For example, orcl.ecaoracle.weather). In our thesis, we use the same approach for authenticating a user table.

Below, we show you the contents of the SysPrimitiveEvent table, once when the trigger is created and then, when the trigger defined on that operation (associated with the event name) is fired.

Table 3-4 (SysPrimitiveEvent – After creating a primitive event)

Db name	User Name	Event name	Table name	Operation	Beaf operation	Time stamp	Vno	Column names	Type
orcl	ecaorcl	addw	weather	Insert	after	Mar032002	0	NULL	N

The information that is persisted in the table is for a trigger created on an insert operation. The trigger fires whenever a tuple is inserted into the WEATHER table. When the trigger is fired, ECA Agent updates the version number of the corresponding tuple as shown in Table 3-5.

Table 3-5 (SysPrimitiveEvent – After detecting a primitive event)

Db name	User name	Event name	Table name	operation	Beaf operation	Time stamp	Vno	Column Names	Type
orcl	ecaorcl	addw	weather	insert	after	Mar032002	1	NULL	N

3.13.2 SysCompositEvent

This table is used for storing information about the composite events. The structure of this table is shown in Table 3-6.

Table 3-6 (SysCompositEvent – structure)

Dbname	Username	Eventname	eventdescribe	timestamp	coupling	context	priority

As we can see, the information that needs to be stored for a composite event is different from what is persisted for a primitive event. Let us take a look at the information that needs to be captured for a composite event; the first three columns are similar to the fields in the table SysPrimitiveEvent, and have the same meaning as before. The fourth column, *eventdescription*, stores the constituent events that make up the given composite event. The column, *coupling*, stores information about the coupling mode of the composite event [15]. The column, *context*, stores information about the context [15] in which the trigger is to be fired. The column, *priority*, stores the priority of the composite event. This field can have a positive number as its value. The last three columns i.e. coupling, context and priority are needed in order to specify the conditions under which a composite event needs to be detected.

Assume that we have composite event trigger created as shown in Figure 3-6. This trigger fires whenever the primitive events defined on Insert and Delete operations get detected, provided, the associated rule semantics are satisfied. Table 3-7 shows the contents of SysCompositEvent after the composite event has been created.

Table 3-7 (SysCompositEvent – After creating a composite event)

Dbname	Username	Eventname	Eventdescribe	timestamp	Coupling	context	priority
Orcl	Ecaoracle	seqw	addw >> delw	22-Jan-02	Immediate	Recent	1

3.13.3 <u>SysECATrigger</u>

The SysECATrigger table is used to store information about all the ECA triggers, both primitive and composite. This table is primarily used for making sure that the trigger names are unique. The structure of SysECATrigger is shown in Table 3-8

Table 3-8 (SysECATrigger – structure)

Dbname	Username	triggername	Triggerproc	Timestamp	eventname

As before, all the columns, except the column *TriggerProc* have the same meaning. The column *TriggerProc*, stores the name of the procedure that has been created for the trigger. A stored procedure is created for a composite event if the user wants to execute a series of actions as a stored procedure.

The contents of the table, SysECATrigger, after the creation of a trigger are shown Table 3-9.

Table 3-9 (SysEcaTrigger – After creating an event)

Dbname	Username	triggername	Triggerproc	timestamp	eventname
Orcl	Ecaoracle	t_seqw	NULL	22-Jan-02	seqw

So, whenever the user tries to create a new trigger, the application verifies that the name of the trigger is not a duplicate and only then, allows the user to create the trigger.

3.13.4 SysContext

The table, *SysContext*, stores the parameter list (table name, context, version number) of a composite event. Whenever a composite event is raised, the parameter list is inserted into this table [16]. The contents of the SysContext table are used for populating the temporary tables.

Table 3-10 (SysContext – structure)

Tablename	Context	Vno

The contents of the SysContext table after two occurrences of primitive events are depicted in Table 3-11.

Table 3-11 (SysContext – After a composite event is detected)

Tablename	Context	Vno
Weather	Recent	3
Weather	Recent	4

As shown, the SysContext will store all occurrences of the primitive events that constitute to the detection of a composite event. This information is later used for populating the transient tables. The contents of the transient tables can be later used for other purposes.

3.13.5 SysDrop

This table is used to store all the constituent events of a composite event. This table is primarily used when we want to drop a trigger defined on a particular event. When the user gives a drop trigger (set on a primitive or a composite event) command, the application checks to see if the event on which this trigger is defined on, is a part of a composite event. If so, the application, does not allow the user to drop this trigger unless the composite event trigger is also dropped. Table 3-12 shows the structure of SysDrop.

Table 3-12 (SysDrop – structure)

Conseventname	Context	Comeventname

The column, *Conseventname*, holds the name of all those events, which are a part of the composite event. The column, *context*, holds the context in which the composite event has been defined. The column, *Comeventname*, holds the name of the composite event.

The contents of the SysDrop table after the creation of a composite event, is shown in Table 3-13

Table 3-13 (SysDrop – After creating composite event in Figure 3-6)

Conseventname	Context	Comeventname
Delw	Recent	seqw
Addw	Recent	seqw

3.13.6 SysRuleTrigger

This table is used for storing the trigger names, associated rule names and the corresponding event name for composite and repeat-composite event triggers. Table 3-14 shows the structure of SysECATrigger.

Table 3-14 (SysRuleTrigger – structure)

EventName	TriggerName	RuleName

The contents of this table are primarily used for dropping composite events or rules associated with a composite event.

3.13.7 Version

The table, Version, stores the occurrence number of all the primitive events that the user has defined. It is updated globally; that is, irrespective of the type of primitive event that is raised; the Version table gets incremented by one each time. Table 3-15 shows the structure of Version.

Table 3-15 (Version – structure)

Vno

The content of the table after raising one primitive event is shown in Table 3-16.

Table 3-16 (Version – After a primitive event is detected)

Vno
1

The Table 3-17 shows the content of the table after three events have been raised.

Table 3-17 (Version – After the occurrences of 3 primitive events)

Vno
3

3.13.8 Notify

Every time a trigger defined on a primitive event is fired, it populates this table. The structure of Notify is shown in Table 3-18.

Table 3-18 (Notify – structure)

Eventname	Tablename	Vno

Once this table is populated, the ECA Agent fetches the values, and notifies the LED about the firing of the trigger. The LED then realizes that a primitive event has been raised and proceeds to register the same.

Table 3-19 shows the contents of Notify after a trigger defined on a primitive event is fired.

Table 3-19 (Notify – After a primitive event is detected)

Eventname	Tablename	Vno
Addw	Weather	1

3.13.9 NotifyCom

Similar to the table Notify, this table is populated whenever a composite event is fired. The structure of NotifyCom is shown in Table 3-20

Table 3-20 (Notifycom – structure)

Comeventname	info

The contents of the table after a composite trigger is fired are shown in Table 3-21.

Table 3-21 (Notifycom – After a composite event is detected)

Comeventname	Info
Seqw	addw 1 delw 2

The contents of the field info provide information about the primitive events and the version number of their occurrence that constitute to make up the composite event. This is not needed except to print a message indicating that this composite event has been raised. In the table shown, it indicates that a composite event, seqw, has been detected, due to as a result of the detection of two primitive events i.e. addw and delw.

3.14 Tables created to provide access to transient values outside trigger body

ECA Agent creates certain tables for each relation on which the user has created a primitive event. These tables are of the form R_Inserted, R_Inserted_tmp, R_Deleted and R_Deleted_tmp, where “R” refers to the name of the relation. Of these four tables, the tables of the form “_Inserted” and “_Deleted” are populated with transient values of a relation that existed at the time the trigger was fired and a primitive event occurrence is detected. The necessity for the ECA Agent to collect these parameters is as follows: The transient values that exist when a trigger fires are accessible only from within the trigger body. However, these values need to be accessed at a later time when a composite event is detected, so that the proper parameters associated with the constituent primitive event is available in the action portion of the composite event. In order to get the proper occurrence of the primitive event, we need to separate the values that exist in the R_Inserted and R_Deleted tables. Therefore a fifth column, which consists of the version number of the occurrence of that primitive event is introduced.

The tables of the form “_tmp” are known as transient tables. These tables are created at the time of creating R_Inserted/R_Deleted tables. They contain the transient values that existed in a table when the primitive events that were created on that table were detected. These tables are populated when a composite event is detected.

If the user is creating an event on either Insert or Update, then the ECA Agent creates R_Inserted and R_Inserted_tmp tables (along with R_Deleted and R_Deleted_tmp for update as it is a delete followed by an insert and hence has values in both tables). If the user is creating an event on Delete operation, then the ECA Agent will create a R_Deleted and R_Deleted_tmp tables.

The R_Inserted and R_Deleted tables will have all the columns of the table on which the primitive event has been created. In addition to this, it will have a column for storing the Version number of the occurrence of the primitive event. For example, if the user has created a primitive event trigger on the table Weather for Insert operation, then the structure of the table R_Inserted will be as shown in Table 3-22.

Table 3-22 (R_Inserted – structure)

City	Time	Temp	Wspeed	Vno

The corresponding transient table, i.e. R_Inserted_tmp, will have the same structure as the relation on which the primitive event is defined. The structure of R_Inserted_tmp is shown in Table 3-23

Table 3-23 (R_Inserted_tmp – structure)

City	Time	Temp	Wspeed	Vno

The contents of the Transient tables will be the result of a join (based on Version number) between the R_Inserted table and Version table.

3.15 Interaction between ECA Agent components

All the client requests are routed through the ECA Agent before being sent to the SQL server. In this section, we explain the different stages of processing that a client request undergoes after being submitted to the ECA Agent.

After a client logs on to the ECA Agent and submits a request, the ECA Agent listener sends this request to the Language Filter. The Language Filter checks to see if the request is a normal SQL command or an ECA command. If the client request is a normal SQL command, the Language Filter sends it to the SQL server directly by making a

JDBC call. On the other hand, if it is an ECA command, then it can fall into one of the following categories

1. A trigger being created on a primitive event or a repeat trigger on a primitive event.
2. A trigger being created on a composite event or a repeat trigger on a composite event.
3. A drop trigger command for dropping either a primitive or a composite event trigger.

Based on the type of command, it is passed on to one of the following modules in either ECA Parser or Drop Trigger module:

1. Primitive Event Parser.
2. Repeat Primitive Event Parser.
3. Composite Event Parser.
4. Repeat Composite Event Parser.
5. Drop Primitive Event.
6. Drop Composite Event.

We will explain the interaction between the modules by taking an example of creating a primitive event trigger and a composite event trigger. A more detailed explanation is provided in the chapters dealing with the Implementation issues.

Assuming that the user command is for creating a trigger on a primitive event, the Primitive Event Parser first validates the trigger syntax, the event name and the trigger name. Then, it extracts the name of the event and creates a ".java" file named after this event. This file is then appended with characters "c_" and then placed in a system wide unique directory structure having the format “username_database”. The “.java” will have the method that can be invoked for registering a primitive event with the LED. Next, the “.java” file is compiled to get the class file and following this, the method for registering the event in LED is executed. This will create a node for this primitive event in the LED. Following this control is passed on to the Persistence Manager module wherein event information is persisted in the System tables and the trigger is created in the SQL server. Now, if a user performs an operation on which the primitive event has been created; the corresponding trigger is fired and it populates the Notify table with the parameters associated with the occurrence of primitive event. After the Notify table is populated, control is returned back to the ECA Agent. The ECA Agent checks the Notify table and if it finds that the table has contents, it calls a method in the file “Led.java” for registering the occurrence of the corresponding primitive event with LED. After the LED registers the occurrence of the primitive event, it propagates the same to all the composite event nodes that have the primitive event detected as a constituent event. This completes the process of creating and detecting a primitive event.

On the other hand, if the user request is for creating a trigger on a composite event, the Language Filter will route this command to the Composite Event parser. Here, the parser creates a “.sjava” file having the event description and rule string, which is constructed from the composite event trigger syntax. The “.sjava” file is submitted to the Snoop Pre-processor. The Pre-processor returns two files; a text file containing a list of all the constituent events of the composite event and a “.java” file. The ECA Agent will now check the System tables to make sure that all the constituent events have been created properly. Once the authentication work is completed, the ECA Agent appends the

ECA action portion and the user action portion to the “.java” file. The ECA Action portion will consist of updating System tables and the temporary tables so that the old and new values of the tables that were modified is available for further manipulation. Once all the required action strings are written into the “.java” file, it is compiled to produce a class file. Finally, the ECA Agent invokes a method in this class file, which contains the API for registering the composite event in the LED. The difference between the node created for a primitive event and that created for a composite event is that; the composite event node will have rule(s) associated with it. A primitive event node does not have a rule associated with it. The reason a rule is associated is because the composite event can be detected in different contexts and different coupling modes. So the LED should have access to certain information (stored as a rule), which will help it to detect these events properly. A primitive event, on the other hand is detected just as a result of the execution of a trigger.

In this chapter, we have looked at the motivation behind using an ECA Agent for providing Active capability to an RDBMS. We then explained the issues that were taken into consideration for designing the ECA Agent. Finally, we presented the design and implementation of the ECA Agent.

In the next chapter, we will discuss the implementation issues concerning the implementation of the primitive event and repeat-primitive event triggers. We discuss how triggers in RDBMS are transformed and correlated with events in Active terminology. Finally we discuss the procedure for dropping a trigger created on a primitive event.

CHAPTER 4

IMPLEMENTATION OF PRIMITIVE EVENTS

In this chapter, we discuss, in detail, the issues concerning the creation and detection of primitive events. The reason we need to know about the occurrence of primitive events outside the scope of the SQL Server is to primarily aid the Java LED in the detection of Composite events. We have to remember here that; we are using another layer of software over the SQL Server in order to enhance the trigger capability of the SQL Server. This layer of Software is what is called *The Generalized Mediator Agent* or *The ECA Agent*. This ECA Agent uses the trigger mechanism of the underlying SQL Server as the basis and builds upon it, in order to enhance the triggering capability and turn the system into a true Active DBMS.

4.1 Triggers in Oracle: A Note

Oracle, by far, in comparison with all other RDBMSs (e.g. Sybase, DB2 etc.) has an expressive trigger mechanism. It supports the following features:

- It allows multiple triggers to be defined on the same event.
- It allows users to create triggers that have multiple events.
- It allows users to create triggers based on Column Update.
- It allows users to create both Statement level triggers and Row level triggers.

Before we proceed, we would like to mention some of the disadvantages of triggers in Oracle [Oracle Manual]. Later in this section, we propose techniques for overcoming these disadvantages.

1. Triggers do not fire in the same order that they are created. The Oracle manual says that the design of an application should always be independent of the order of the firing of the trigger.
2. Only Row level triggers allow us to access “OLD” and “NEW” values inside the trigger body.
3. If there are two row level triggers that have been created on the same event, and both these triggers try to access the “OLD” and “NEW” values from within the trigger body, it invariably leads to a problem called the mutating constraint problem.

Now that we have given you an insight about some of the properties of triggers in Oracle, we will proceed by taking a look at the extended trigger syntax for creating a primitive event.

4.2 Extended trigger syntax for creating a primitive event

Below, we show you the normal trigger syntax (Figure 4-1) as well as the extended trigger syntax (Figure 4-2) for creating a primitive event. As we can see, the difference between the extended trigger syntax and the normal trigger syntax is the inclusion of the keyword *event* followed by the name of the event (*addw*). By including these two, we inform the ECA Agent that this is an ECA trigger. If the keyword *event* is not included, the language filter thinks that this is a normal trigger and sends it to the SQL Server directly through JDBC.

```
//Normal trigger syntax
create trigger t_addw after insert on WEATHER
Begin
        insert into temp values (‘t_addw’, sysdate);
end;
```

Figure 4-1 (Normal trigger syntax)

```
//Extended trigger Syntax.
create trigger t_addw event addw after insert on WEATHER
Begin
        insert into temp values (‘t_addw’, sysdate);
end;
```

Figure 4-2 (Extended trigger syntax – primitive event trigger on table)

```
//Extended trigger Syntax for Update of Column
create trigger t_updw event updw after update of  temp, wspeed on WEATHER
Begin
        insert into temp values (‘t_addw’, sysdate);
end;
```

Figure 4-3 (Extended trigger syntax – Trigger on Column Update)

```
//Extended trigger syntax for multiple operations
create trigger t_comb event combW after insert or update or delete on WEATHER
Begin
        insert into temp values (‘t_addw’, sysdate);
end;
```

Figure 4-4 (Extended trigger syntax – Trigger on multiple operations)

4.3 Processing a Primitive Event

The processing of a primitive event involves the following two tasks:

1. Creation of the trigger in the SQL Server

2. Generation of the primitive event node.

We have split our explanation into two parts. First, we explain the process of creating a primitive event. Next, we explain the process of Event Notification when the trigger is fired.

4.3.1 Creation of a Primitive Event

As explained earlier, before a primitive event is created, the trigger has to undergo several stages of parsing that includes syntax check, duplication check etc. We explain each of these steps in detail below:

4.3.1.1 Syntax Check

This is the first stage of parsing. Here, the trigger command is checked to see if the keyword *event* is present or not. If it is not present, but, it has an event name, then, the parser decides that there must be some problem with the trigger syntax and returns an error message. In the case of both the keyword *event* and the event name not being present, the parser decides that the trigger is a normal trigger and sends it to the SQL Server directly. However, if both *event* and event name are present, the parser decides that the trigger is in fact an ECA trigger and forwards it for further processing.

4.3.1.2 Duplication Check

Here, the ECA parser does the following:

- It checks SysECATrigger to see if a trigger with the same name already exists. In case it does, the ECA parser returns an error message.
- It checks the table SysPrimitiveEvent to see if a trigger has been defined with the same event name. If it finds one, it returns an error message.

4.3.1.3 <u>File Generation</u>

The next step is to create a file from which we make a call to the Java LED in order to register the primitive event and create a primitive event node in the event graph.

Before we proceed, we would like to remind you about our earlier discussion on how we make the ECA Agent capable of handling multiple users and multiple databases. As explained, we first append the string "c_" to the name of the event. Next, we generate the contents of the ".java" file. We then write these contents into the file *c_eventname.java*. This completes the process of generating the file. The Figure 4-5 shows the contents of the file *c_addw.java*, which is generated as a result of trigger shown in Figure 4-2.

```
import sentinel.led.*;
 import java.util.Vector;
 import java.util.Hashtable;
 import java.util.Enumeration;
public class c_addw{
 public static EventHandle addw =null;
  public static void call_addw(){
 ECAAgent myAgent = ECAAgent.initializeECAAgent();
 addw  =  myAgent.createPrimitiveEvent("addw","Led",  EventModifier.BEGIN,
"void addw()",  DetectionMode.SYNCHRONOUS);
 }
 }
```

Figure 4-5 (Java file created for primitive event (addw))

4.3.1.4 <u>Register the Event</u>

Once the ".java" file is generated, the next step is to compile this file. We use the method *getRunTime* in the class java.lang.Runtime, which returns the runtime object associated with the current java application. The next step would be to use the run time object and compile the ".java" file. For this, we use the *exec* method of the class java.lang.Runtime.

This method creates a native process and returns an instance that is a subclass of the class Process. This can be used for obtaining information about the process and controlling it. We use it to stall the execution of all other processes related to this application till this process is executed. The method *waitfor* is used for accomplishing this task.

```
String cmd = "javac -classpath Server\\classes; " + "metadata\\" + dir + "\\" +
 className + ".java";
    java.lang.Runtime rt = Runtime.getRuntime();
    try
    {
      Process pro = rt.exec(cmd);    // execute the command
      System.out.println(" Compiling.  :"+cmd);
      int a = pro.waitFor();   // wait until the current process terminate,
                               // So that the command completed
      System.out.println(" Compile Finished.  :"+cmd);
    }
    catch(Exception e4)
    {
      System.out.println(e4.toString());
    }
```

Figure 4-6 (Code for compiling the Java file dynamically)

Once the file is compiled, we need to execute a method (that makes a call to the Java LED) in order to register the event. In order to do this, we first need to load the compiled class by using the method *getSystemClassLoader* present in class *ClassLoader*. Next, we use the method *loadClass* to load the class. Finally, we invoke the method, which is responsible for making a call to the Java LED and registering the event. Figure 4-7 depicts these steps.

```
String c = className + ".class" ;
  String cmd =   "cl.bat " + dir + " " + c;
  java.lang.Runtime rt = Runtime.getRuntime();
  try
  {
    Process pro = rt.exec(cmd);     // execute the command
    int a = pro.waitFor();        // wait until the current process terminate
  }
  catch(Exception e4)
  {
    System.out.println(e4.toString());
  }
  /*load the new compiled class*/
  String strClass = className;          // "InsertStock", etc
  ClassLoader cl = null;
  cl = ClassLoader.getSystemClassLoader ();
  Class cla = null;
  try
  {
    cla = cl.loadClass(strClass);     // load the class
  }
  catch(ClassNotFoundException e4)
  {
    System.out.println(e4.toString());
  }
  // get the method from the class
  java.lang.reflect.Method meth = null;
  try
  {
    meth = cla.getMethod(methodName, paramTypes);  // find the particular method from the
                                                    // loadedclass
    Object obj = null;
    meth.invoke(obj, params);      // execute the particular method to register the primitive event
  }
  catch(Exception e5)
  {
    System.out.println(e5.toString());
  }
```

Figure 4-7 (Code for registering the primitive event in LED)

4.3.1.5 Create Temporary tables

The next step is to create the temporary tables. These temporary tables have the same fields as the fields of the table on which the trigger is being created. In addition, it also has a column for holding the version number indicating the occurrence number of the trigger as to when it was fired.

The temporary tables created for Insert and Update operations are of the form *R_Inserted*, where "*R*" refers to the name of the table on which the trigger has been created. Similarly, for Delete operations, the temporary tables are of the form *R_Deleted*.

Also, the creation of every *R_Inserted/R_Deleted* table is followed up with the creation of a corresponding *R_Inserted_tmp/R_Deleted_tmp* table. These tables are used for holding the parameters of the primitive events, the occurrences of which contribute to the detection of a composite event. The tables *R_Inserted_tmp* and *R_Deleted_tmp* are known as transient tables. They hold parameter tuples as values in them only when the composite event is fired. During the next run of the ECA Agent thread, these values are deleted so that values corresponding to a new composite event can be inserted.

4.3.1.6 Creating the Triggers

This is the penultimate step in the process of creating a primitive event. This is broken down into two separate parts so that a clear distinction is made between the actions specified by the user and the actions that the ECA Agent has to perform in order to provide active capability. The first part of this includes creating a trigger that performs ECA Actions when it is fired. The second part includes creation of a trigger that performs user actions when it is fired. Below, we explain how the triggers are created.

Consider the trigger statement in figure shown in Figure 4-2. This trigger, after passing through the Parser, is split into two triggers. The first trigger is appended with a "0" (zero). So, the name of this trigger becomes, *t_addw0*. This trigger is used for performing all the ECA actions. For the second trigger, we retain the same name as given by the user i.e., *t_addw*. This trigger is used for performing the user actions.

The structure of these two triggers is shown in Figure 4-8 and Figure 4-9.

```
Create trigger t_addw0 after insert on weather for each row
declare cursor c1 is select vno from version;
begin
   update SYSPRIMITIVEEVENT set VNO=VNO+1 where EVENTNAME='addw';
   update Version set VNO = VNO + 1;
   insert in to notify select EVENTNAME, TABLENAME, VERSION.VNO from
   SYSPRIMITIVEEVENT, VERSION where EVENTNAME='addw';
   for vno_rec in c1 loop
      insert into weather_inserted
             values(:new.CITY,:new.TIME,:new.TEM,:new.WSPEED, vno_rec.vno);
   end loop;
end;
```

Figure 4-8 (Trigger generated for performing ECA actions)

```
Create trigger t_addw after insert on weather for each row
Begin
          Insert into temp values (‘addw’, sysdate);
End;
```

Figure 4-9 (Trigger for performing user actions)

The reason we split the trigger into two parts is to allow the users to drop the trigger without losing the ECA functionality. This is best illustrated by an example.

Assume that there are two triggers, t_addw and t1_addw, defined on the same event insert. Now, suppose we associate the ECA Actions with the first trigger i.e. t_addw, then, when this trigger is dropped, we also loose the corresponding ECA functionality. This is not correct because we still have one more trigger t1_addw which fires on insert.

However, the firing is not accompanied by any ECA actions and hence no event is detected. Now, since we have split the trigger t_addw into two parts i.e. t_addw and t_addw0, the trigger t_addw can be dropped without any consequence. This is because; the trigger t_addw0 is still present. So whenever an insert operation takes place, this trigger gets fired and subsequently results in the detection of an event as also the firing of the trigger t1_addw.

4.3.1.7 Persisting the trigger Information

This is the last step in the creation of the primitive event. In this step, the information about the trigger is persisted in the SysECATrigger table. This enables the ECA Agent to perform duplication check when the user is creating a new trigger. This step is illustrated in Table 4-1.

Table 4-1 (SysECATrigger – After a primitive event is created)

dbname	username	triggername	triggerproc	timestamp	eventname
Orcl	ecaoracle	t_addw	NULL	22-Jan-02	addw

4.3.2 Primitive Event Notification

Once a trigger corresponding to a primitive event is fired, the following actions take place.

- The SysPrimitiveEvent table is updated with the latest version number.
- The Version table is updated with the latest version of the occurrence of primitive event on a global basis.
- A tuple is inserted into the Notify table, which has information about the trigger that was fired.

Once these operations take place, control is transferred to the ECA Agent. Inside the ECA Agent, the thread handling the particular user makes a query to find out whether the Notify table has any content in it. If it finds out that the Notify table has a tuple, it will enter a loop wherein it queries and gets the corresponding table name and version number. It then makes a call to the method PrimEvent, which is present in the file Led.java by passing all these (event name, table name, version number) as parameters.

The method PrimEvent now detects a primitive event and raises the same in the Java LED. The Figure 4-10 illustrates the different stages involved in Primitive Event Notification.

```
/*
check if there is primitive event raised. Every time a sql statement is executed, these
lines are executed to see if a primitive event has bee raised
*/
String qs2 = "select eventname from notify order by vno";
Jdbc selectSql = new Jdbc(rdbms,url,username,password,qs2);
qs2 = selectSql.GetFromNotify().trim ();
 String en = selectSql.GetFromNotify().trim ();
 /*
Added to take care of multiple triggers getting fired as a result of having a column
update and a statement level update on the same table.
*/
String delimit2 = " ";
Vector vEventTokens = new Vector(1,1);
StringTokenizer st2 = new StringTokenizer (en, delimit2);
while (st2.hasMoreTokens())
{
  vEventTokens.addElement(st2.nextToken());
}
if (!vEventTokens.elementAt(0).toString().toUpperCase().trim().equals("F") &&
      !vEventTokens.elementAt(0).toString().toLowerCase().trim().equals("empty"))
```

Figure 4-10 (Code for checking the occurrence of a primitive event)

```
{
  for (int j = 0; j < vEventTokens.size(); j++)
  {
   /* Jan-24-2002
  NOTE: This logic fails when you update a column in all the tuples.
  For E.g. update weather set city='SANDIEGO' We need to add one more loop to take
  care of this problem
  */
  qs2 = "select TABLENAME from notify where eventname='" +
   vEventTokens.elementAt(j).toString().trim()+ "'" + "order by vno";
   selectSql = new Jdbc(rdbms,url,username,password,qs2);
   String tn = selectSql.GetFromNotify().trim ();
   System.out.println("TNAME: " + tn);
  qs2 = "select VNO from notify where eventname='" +
   vEventTokens.elementAt(j).toString().trim()+ "'" + "order by vno";
   selectSql = new Jdbc(rdbms,url,username,password,qs2);
   String vnos = selectSql.GetFromNotify().trim ();
   Integer vnoi = new Integer(vnos);
   int vno = vnoi.intValue();
    if(test1 == null)
    {
     test1 = new Led();
     try
     {
      Thread.currentThread().sleep(1000);
      }
      catch (Exception e)
      {
       System.out.println(e.getMessage());
       }
      }
     test1.PrimEvent(vEventTokens.elementAt(j).toString().trim(),tn,vno);   //raise primitive
    //event.
     result = result + "Raising Primitive Event : EventName: (" + en +") TableName: ("+tn+")
    VersionNumber: ("+vno+").\n";
    }
  }
```

Figure 4-10 (cont.)

NOTE: In the mediator agent that was developed for Sybase using a Gateway Open Server, Remote Procedure Calls (RPC) were used for notifying the LED about the occurrence of the primitive event. These RPC calls were made from within the trigger body. As compared to this, there is a small amount of delay that is introduced in the present version because we are notifying the LED about the occurrence of the primitive event outside the trigger body. This approach had to be adopted as JDBC imposes a limitation and does not allow Remote Method Invocation (RMI) from within the trigger body.

4.4 Processing other type of triggers

In addition to supporting triggers created on primitive operations (Insert/Delete/Update), Oracle also supports certain other triggers. We have extended the ECA Agent considerably in order to support these triggers. We present each of these different types of triggers and explain how the ECA Agent handles these triggers.

4.4.1 Triggers on Column Update

In this case, the trigger fires only if the columns specified in the trigger statement are modified. These triggers are handled the same way as the triggers created on the entire table. The only difference here is that, we extract the column names that are present in the trigger definition and persist them in the SysPrimitiveEvent. Figure 4-11 illustrates steps where the column names are extracted and added to the trigger command before persisting the trigger.

```
String getCol = "select COLUMNNAMES from SYSPRIMITIVEEVENT where EVENTNAME='" + eventname + "' and " + "OPERATION='" + operation + "'";
getValue = new Jdbc(rdbms,url,username,password,getCol);
columnNames = getValue.GetValueFromTable().trim();
if (colUpdFlag)
{
 operation = operation + " of " + columnNames;
}
```

Figure 4-11

The contents of the SysPrimitveEvent table following the creation of a trigger on column update is as shown in

Table 4-2.

Table 4-2 (SysPrimitiveEvent – after a column update trigger is created)

Db name	User name	Event name	Table Name	operation	Beaf operation	Time stamp	Vno	Column names	T
orcl	ecaorcl	addw	weather	insert	after	Mar032002	0	tem,wspeed	U

4.4.2 Triggers created on multiple operations

Triggers created for multiple operations fire when any one of the operations specified in the trigger statement takes place. These type of triggers are subjected to a slightly different treatment. The difference is in the last two steps where we create the trigger for handling the ECA Actions and persist the trigger information. In this case, we extract all the operations that form a part of the trigger definition. This information is persisted in the SysPrimitiveEvent table. For example, if the trigger statement has all three operations specified in the create trigger syntax; the SysPrimitiveEvent table will have entries as shown in Table 4-3.

Table 4-3 (SysPrimitiveEvent – After a trigger on multiple operations is created)

Db name	User name	Event name	Table Name	operation	Beaf operation	Time stamp	Vno	Column names	T
orcl	ecaorcl	combw	weather	insert	after	Mar032002	0	NULL	M
orcl	ecaorcl	combw	weather	update	after	Mar032002	0	NULL	M
orcl	ecaorcl	combw	weather	delete	after	Mar032002	0	NULL	M

As shown in the figure, in the SysPrimitiveEvent table, instead of just one entry for the trigger, we have three entries, one corresponding to each operation. Then, the operations that have been specified in the trigger syntax are passed as an array to the Persist Manager, which is the module responsible for splitting the triggers into two parts and persisting them in the SQL Server. Once control is transferred to the Persist Manager, it updates certain flags based on the operations on which the trigger is being created. For example, if the trigger is being created on only insert and update operations, then, the Persist Manager decides that temporary tables of the form R_Inserted (and R_Inserted_tmp) should be created and used while temporary tables of the form R_Deleted (and R_Deleted_tmp) need not be created. However, if the trigger syntax has a delete operation specified, then, the Persist Manager creates temporary tables for both insert and delete operations (i.e. R_Inserted and R_Deleted). The figure below shows the syntax of creating such a trigger. Figure 4-12 depicts how this trigger is handled in the Persistence Manager.

```
String createTriggerCommand = "create trigger " + triggername + "0" + " " + triggerCommandPart
                  + " for each row  declare "
                  + "cursor c1 is select vno from version; "
                  + "begin "
                  + "if inserting "
                    + "then "
                    + " update SYSPRIMITIVEEVENT set VNO=VNO+1 where EVENTNAME='"
                    + eventname + "' AND operation='insert'; "
                    + " update Version set VNO = VNO + 1; "
                    + "insert into notify select EVENTNAME, TABLENAME,"
                    + " VERSION.VNO from SYSPRIMITIVEEVENT, VERSION where
                         EVENTNAME='" + eventname + "' AND operation='insert'; "
                    + "for vno_rec in c1 loop "
                    + strInsert +" vno_rec.vno); " + "end loop; "
                  + "end if;"
```

Figure 4-12 (Code for handling a trigger on multiple operations)

```
            + "if deleting "
             + "then "
             + " update SYSPRIMITIVEEVENT set VNO=VNO+1 where EVENTNAME='"
             + eventname + "' AND operation='delete'; "
             + " update Version set VNO = VNO + 1; "
             + "insert into notify select EVENTNAME, TABLENAME,"
             + " VERSION.VNO from SYSPRIMITIVEEVENT, VERSION where
                  EVENTNAME='" + eventname + "' AND operation='delete'; "
             + "for vno_rec in c1 loop "
             + strDelete +" vno_rec.vno); " + "end loop; "
            + "end if;"
            + "if updating "
             + "then "
             + " update SYSPRIMITIVEEVENT set VNO=VNO+1 where EVENTNAME='"
             + eventname + "' AND operation='update'; "
             + " update Version set VNO = VNO + 1; "
             + "insert into notify select EVENTNAME, TABLENAME,"
             + " VERSION.VNO from SYSPRIMITIVEEVENT, VERSION where
                   EVENTNAME='" + eventname + "' AND operation='update'; "
             + "for vno_rec in c1 loop "
             + strInsert +" vno_rec.vno); " + "end loop; "
            + "end if;"
            + "end;" ;
   Jdbc createTrigger = new Jdbc(rdbms,url,username,password,createTriggerCommand);
   System.out.println(createTrigger.ExecuteSqlUpdate("Create Trigger " + triggername + "0" + "
   on " + operation));
```

Figure 4-12 (cont.)

As shown in Figure 4-12, the Persist Manager sets the values of the two flags according to the events that are present in the trigger definition. Based on how these flags are set, one of the blocks that are responsible for creating the triggers are called.

4.5 Repeat Primitive Event

Oracle allows users to create multiple triggers on the same event. This feature is retained even after the ECA Agent is added with some minor modifications. The extended trigger syntax for a repeat trigger is shown in Figure 4-13. As shown, the repeat trigger syntax varies considerably from the ECA trigger syntax. In effect, the user is not allowed to change the time of the firing of the trigger. This means that if the primitive trigger has been created as an *After* event trigger, then the repeat primitive trigger set on the same event automatically becomes an *After* event trigger.

```
create trigger t1_addw event addw
Begin
update temperature set htem=(select max(tem) from weather where city in (select city
     from temptable)) where city in (select city from temptable);
end;
/
```

Figure 4-13 (Repeat Primitive Event trigger – syntax)

Figure 4-14 shows the different steps involved in the creation of a repeat primitive trigger. Following that, we proceed to explain these steps.

```
qs2 = "select TABLENAME from SYSPRIMITIVEEVENT where EVENTNAME='"
        + eventname + "' and USERNAME='" + username + "' and DBNAME='"
        + database + "'";
getValue = new Jdbc(rdbms,url,username,password,qs2);
tablename = getValue.GetValueFromTable().trim();

/*
get 'operation'
*/
qs2 = "select OPERATION from SYSPRIMITIVEEVENT where EVENTNAME='"
        + eventname + "' and USERNAME='" + username + "' and DBNAME='"
        + database + "'";
getValue = new Jdbc(rdbms,url,username,password,qs2);
operation = getValue.GetValueFromTable().trim();

qs2 = "select BEAFOPERATION from SYSPRIMITIVEEVENT where EVENTNAME='"
         + eventname + "' and USERNAME='" + username + "' and DBNAME='"
         + database + "'";
getValue = new Jdbc(rdbms,url,username,password,qs2);
beforeAfter = getValue.GetValueFromTable().trim();
```

Figure 4-14

From Figure 4-13, we see that the repeat primitive trigger syntax has just the event name. It does not specify the operation, the table name and the time of trigger firing. Evidently, we need to extract this information from the System tables before we try to create the repeat trigger. Figure 4-14 depicts these steps. We extract the table name, operation and the time of trigger firing from the table SysPrimitiveEvent. Once this is extracted, we append these to the trigger body at the appropriate before sending it to the SQL Server for persisting the same.

Figure 4-15 below shows the modified repeat trigger after it has been appended with the information extracted out of the system table. The words that have been italicized, refers to the information that has been extracted out of the table, SysPrimitiveEvent and appended to the trigger body.

```
create trigger t1_addw event addw after insert on WEATHER
Begin
update temperature set htem=(select max(tem) from weather where city in (select
   city from temptable)) where city in (select city from temptable);
end;
```

Figure 4-15

Once the trigger has been modified, it is sent to the SQL Server and the trigger gets created.

4.5.1 Repeat Triggers on Column Update

If the user creates a primitive trigger on Column update, the column names on which the trigger has been created is persisted in the SysPrimitiveEvent table. Now, if the user creates a repeat primitive trigger on an update operation, then the ECA Agent has to determine if the primitive trigger that was earlier set for the same event was a column update trigger or just a normal update trigger. In order to check this, the ECA Agent queries the SysPrimitiveEvent table and extracts the value type field (*Type*). If this field has a value “U”, then the ECA Agent determines that the primitive event was set on update of column(s) and hence takes the appropriate actions as shown.

```
• Extract column names from SysPrimitiveEvent table.
• if ((column Names = NULL) or (column Names = ‘f’))
          then normal trigger
  else
          operation = operation + “of” + column Names
```

As shown, the names of the columns, along with the keyword “of”, is appended to the variable, which holds the name of the operation. Then, the trigger is sent to the SQL server for persisting it in the System table.

4.5.2 Repeat Triggers on triggers created on multiple operations

As explained earlier, for a trigger created on multiple operations, the SysPrimitiveEvent table will have multiple entries, one corresponding to each operation. So, when a repeat trigger is submitted on the same event, querying the SysPrimitiveEvent table will return multiple operations instead of just one. So, these operations are split and then appended to the trigger syntax before the trigger is created on the system. These operations are depicted in Figure 4-16.

```
qs2 = "select OPERATION from SYSPRIMITIVEEVENT where EVENTNAME='"
         + eventname + "' and USERNAME='" + username + "' and DBNAME='" + database + "'";
getValue = new Jdbc(rdbms,url,username,password,qs2);
operation = getValue.GetValueFromTable().trim();
Vector vOp = new Vector (2,1);
StringTokenizer st = new StringTokenizer (qs2, " ");
while (st.hasMoreTokens ())
{
   vOp = st. addElement(st.nextToken());
}
while (i < vOp.size)
{
  if ( i!= vOp.size)
  {
    operation = operation + vOp.elementAt(i).toString +"or" ;
  }
  else
  {
     operation = operation + vOp.elementAt(i).toString
  }
}
```

Figure 4-16

4.6 Oracle Triggers: A brief discussion

As was mentioned earlier in the chapter, there are a few disadvantages with creating multiple triggers on the same event. The problem is that, the triggers do not fire in the same order that they have been created. The Oracle manual says that the design of an application should never be dependent on the order of the firing of the triggers. Despite this, if the use of repeat triggers becomes necessary, there are certain issues that the user needs to take care of. We discuss these issues in this section.

4.6.1 Order of Firing

In Oracle, the triggers do not follow a particular order of firing. So, a trigger that has been created in the end might be triggered before any other trigger. Oracle manual specifies that the applications should be designed in such a way that they are not dependent on the order of firing of the triggers.

In spite of this, if the user wants multiple actions to be performed as a result of the occurrence of an event, the user can create multiple triggers on the same event. However, if the user wants these actions to take place in a particular order, then the user should not create multiple triggers on that operation; Instead, the user should create one single trigger and specify all the actions to be performed as a part of a stored procedure. Now, in a stored procedure, all the actions specified are going to take place in the same order and thus the problem with multiple triggers can be overcome.

4.6.2 Mutating Constraint problem

Oracle allows "OLD" and "NEW" values to be accessed only from within the body of a row level trigger. Suppose the user has created multiple triggers on the same event, and the user wants to access these values in all the triggers, then if all these triggers are made row level triggers, it will lead to a problem called mutating constraint problem. To overcome this, the user should have only one row level trigger. From within this trigger, the user should access all the values and insert them into a temporary table. All the other triggers should be statement level triggers. If these statement level triggers need to access the "old" and "new" column values, they should access the temporary table and extract the values needed.

For example, if we have a trigger created on insert and we have two repeat triggers created on the same operation, the order that needs to be followed for avoiding the mutating constraint problem is shown in the Figure 4-17.

```
//Primitive Trigger
create trigger t_addw after insert on weather event addw
Begin
   update wspeed set hspeed=(select max(wspeed) from weather where city in (select city
   from temptable)) where city in (select city from temptable);
end;
//Repeat Primitive Trigger – Statement level trigger
create trigger t1_addw event addw
Begin
   update temperature set htem=(select max(tem) from weather where city in (select city
   from temptable)) where city in (select city from temptable);
end;
//Repeat Primitive Trigger – row level trigger. Inserting "new" values into the //temporary
table.
create trigger t2_addw event addw for each row
Begin
   delete from temptable;  insert into temptable values (:new.city, :new.tem, :new.wspeed);
   delete from temperature where city=:new.city; insert into temperature values
      (:new.city,:new.tem,:new.tem,:new.tem);
   delete from wspeed where city=:new.city; insert into wspeed values
      (:new.city,:new.wspeed,:new.wspeed,:new.wspeed);
end;
```

Figure 4-17

Till now, we have explained how primitive events can be created and how the ECA Agent detects the occurrence of the primitive event and subsequently notifies the user about it. Next, we will explain how the primitive events are dropped and issues related to dropping a primitive event.

4.7 Dropping a Primitive Event trigger

In this section, we discuss the topic of dropping a trigger created on a primitive event. We would also like to point out the symmetry that exists between the process of creating a trigger and dropping the same.

Recalling how a primitive trigger is created, we observe that when a primitive trigger is created, we actually create two triggers, one for performing the user actions and the other for performing the ECA actions. Consequently, we have two entries in the

System tables -- SysEcaTrigger and User_Trigger. For example, assume that we have created a trigger called t_addw on Insert operation; let the name of the event be addw. Then the entries in the SysEcaTrigger table will be as shown in Table 4-4.

Table 4-4

DBNAME	USERNAME	TRIGGERNAME	TRIGGERPROC	TIMESTAMP	EVENTNAME
orcl	ecaoracle	t_addw	t_addw_proc	Feb-07-2002	addw
orcl	ecaoracle	t_addw0	t_addw0_proc	Feb-07-2002	addw

Similarly, the metadata maintained by the underlying DBMS (Oracle) also has two entries for the above two triggers. Hence, when a primitive trigger is dropped, we should also drop the trigger that has been created for performing the ECA actions. However, we have to remember that the trigger for performing ECA actions should be dropped only if there are no repeat primitive triggers that have been created on the same event. The reason for doing this is that the ECA Actions should still be performed as long as there are other triggers defined on a particular operation; this will allow us to detect the occurrence of a primitive event.

When a user submits the request to drop a trigger, the language filter does the following:

- It checks the table SysEcaTrigger to see if the trigger exists. If it does not, the Language Filter returns an error message.
- It checks the table SysPrimitiveEvent. If the event name is present in this table, it decides that this is a Primitive trigger and passes control to the Drop Primitive Trigger module. However, if the event is not present in the table, it decides that the trigger may be a Composite trigger and passes the control to the Drop Composite Trigger module.

Inside the Drop Trigger module, the following actions take place.

- We first check the table SysDrop to see if the primitive event is a constituent event of any of the Composite events. If this is the case, the user is informed that this trigger cannot be dropped. But, if this event is not a constituent event of any other event, then the following actions are taken:
- The trigger is dropped from both the SysEcaTrigger table and the system table, User_Trigger.
- Next, we check the SysEcaTrigger table to see if there is more than one trigger defined on this event. If so, it is decided that there are repeat triggers and hence the trigger created for performing ECA actions cannot be dropped. However, if there is just one trigger defined on the given event, the ECA Agent decides that it corresponds to the trigger that has been created for taking care of the ECA actions. Now this trigger is dropped from the SysEcaTrigger table and the System table.
- Following this, the entry corresponding to the trigger in the table SysPrimitiveEvent is also deleted.
- Finally, the ".java" file and the ".class" file that were created at the time of creating the primitive event are deleted.

This completes the process of dropping a primitive trigger.

In this chapter, we have explained how primitive events are created. We have discussed how these primitive event occurrences are notified to the LED and to the user. We also discussed the procedure followed for dropping triggers created on primitive events. In the next chapter, we will present the procedure for creation and detection of composite events and repeat composite events. We also explain how composite events are dropped.

CHAPTER 5

IMPLEMENTATION OF COMPOSITE EVENTS

At present, none of the commercially available RDBMSs support composite events. However, the prototype that has been developed makes provision for any of these RDBMSs to support the detection of composite events. In this chapter, we explain how the trigger syntax has been extended in order to include information for the creation of composite events. We then continue by explaining the different steps involved in the creation of the composite events. Finally, we discuss the syntax and working of Repeat Composite events.

5.1 Extended trigger syntax for the creation of composite events

Figure 5-1 shows the extended trigger syntax for creating composite events in Oracle. The syntax remains more or less the same for other DBMSs but for minor changes. Refer to chapter 3 for the BNF of a composite event.

```
                                 //Include a periodic event here
create trigger t_seqw event seqw = delw >>  addw : IMMEDIATE RECENT 1
Begin
   insert into temp values('t_seqw',sysdate);
end;
```

Figure 5-1 (Composite event trigger – syntax)

Before we proceed, we would like to explain a few things about the syntax of the trigger.

As shown in Figure 5-1, the words that have been italicized are the extensions. The first part of the statement before colon (:), i.e., event seqw = delw >> addw gives the event description. Here event is just a keyword. seqw is the name of the composite event. The expression delw >> addw is the Snoop expression. The second part of the statement consists of the context, coupling mode and priority in that order. We will now move on to explain how this trigger statement is parsed and explain the steps involved in the creation of a composite event.

5.2 Processing of a Composite Event

We split the explanation into two parts:

- Creation of the Composite Event
- Composite Event notification

5.2.1 Creation of the Composite event

Before the trigger on the composite event is created, it undergoes several stages of parsing that includes syntax check, duplication check etc. This is explained below.

5.2.1.1 Trigger validation

Once the Language Filter recognizes the trigger submitted by the user as a composite event trigger, it passes the trigger statement to the Composite Event Parser for further processing. The composite event parser first checks the trigger statement for its syntax. It checks to see if the trigger statement has a colon (:). If it is not present, the parser immediately returns an error message asking the user to resubmit the trigger statement. The next step is to check whether the trigger name is unique. If it is not unique, an error message is returned. In addition to the trigger name, we need to check whether the event name is unique. Event name is obtained from the event expression.

seqw = delw >> addw

From the above event description, the parser extracts the name of the composite event i.e. seqw. If the above event name is already used (checked in the SysCompositeEvent table), an error message is returned.

5.2.1.2 Create the rule

The next step is to create a string for defining the rule. In order to do this, the Parser first gets the context, coupling mode and the priority of the Composite event. If the user has not specified any of these, then a default value will be assigned; the default values being Immediate (coupling mode), Recent (context) and 1 (priority). Once all the necessary information is obtained, the parser generates the string for creating a rule. This is shown in the Figure 5-2.

```
// event seqw = delw >> addw;
EventHandle seqw = myAgent.createCompositeEvent(EventType.SEQ, "seqw"
,c_delw.delw ,c_addw.addw);
// rule Rule_seqw [ seqw , c_seqw.True , c_seqw.seqwecaoracle , 1 , IMMEDIATE ,
//RECENT ];
myAgent.createRule("Rule_seqw",seqw, "c_seqw.True", "c_seqw.seqwecaoracle", 1,
CouplingMode.IMMEDIATE, ParamContext.RECENT);
```

Figure 5-2 (rule created for composite event trigger in Figure 5-1)

Since a composite event can have several rules associated with it, these rules should have unique name so that they are different from other rules. So, the rule that is initially created for a composite event has the form “Rule_eventname” while all other subsequent rules that might be created as a result of repeat-composite event triggers have the form “Rule_triggername_eventname”.

5.2.1.3 Create the ".sjava" file to be submitted to the Snoop Preprocessor

Next, the parser creates a ".sjava" file and submits the same to the Snoop Preprocessor. The format of the ".sjava" file is shown in Figure 5-3.

```
//c_seqw.sjava
import sentinel.led.*;
 import java.util.Vector;
 import java.util.Hashtable;
 import java.util.Enumeration;
 public class c_seqw{
static String rdbms = "";
  static String url = "";
  static String username = "";
  static String password = "";
  public static EventHandle seqw =null;
 public static void call_seqw(String Prdbms, String lru, String Pusername, String Ppassword){
 rdbms = Prdbms;
 url = lru;
 username = Pusername;
 password = Ppassword;
 ECAAgent myAgent = ECAAgent.initializeECAAgent("ecaoracle_ORCL");
 //event seqw = delw >> addw;
 rule Rule_seqw[seqw,c_seqw.True,c_seqw.seqwecaoracle,1,IMMEDIATE,RECENT];
}
 public static boolean True(ListOfParameterLists parameterLists) {
 System.out.println("***** From Condition ***** ");
 return true;
}
```

Figure 5-3 (File which is submitted to the Snoop Pre-processor (SPP))

The Snoop Preprocessor parses the ".sjava" file and returns two files. The first file is a ".txt" file and the second file is a ".java" file.

```
//c_seqW.txt
delw
addw
```

Figure 5-4 (Text file returned by SPP)

```
//c_seqw.java
public class c_seqw {
static String rdbms = "" ;
static String url = "" ;
static String username = "" ;
static String password = "" ;
public static EventHandle seqw = null ;
public static void call_seqw ( String Prdbms , String lru , String Pusername , String
Ppassword ) {
rdbms = Prdbms ;
url = lru ;
username = Pusername ;
password = Ppassword ;
ECAAgent myAgent = ECAAgent.initializeECAAgent ( "ecaoracle_ORCL" ) ;

// event seqw = delw >> addw;
EventHandle seqw = myAgent.createCompositeEvent(EventType.SEQ, "seqw"
,c_delw.delw ,c_addw.addw);

// rule Rule_seqw [ seqw , c_seqw.True , c_seqw.seqwecaoracle , 1 , IMMEDIATE ,
RECENT ];
myAgent.createRule("Rule_seqw",seqw, "c_seqw.True", "c_seqw.seqwecaoracle", 1,
CouplingMode.IMMEDIATE, ParamContext.RECENT);
}
```

Figure 5-5 (Java file returned by SPP)

5.2.1.4 Get the ECA Action and create the composite event

Once the Parser has verified that all the constituent events have been defined, it proceeds to create the final version of the ".java" file. Before it creates this, it gets the ECA Actions and the User Actions and appends it to the file obtained from the pre-processor. It then compiles this file by following the procedures we described in the previous chapter. Then the compiled file is loaded and the method for registering the composite event is executed. The final version of the ".java" file that is generated is shown below.

```
import sentinel.led.* ;
import java.util.Vector ;
import java.util.Hashtable ;
import java.util.Enumeration ;
public class c_seqw {
static String rdbms = "" ;
static String url = "" ;
static String username = "" ;
static String password = "" ;
public static EventHandle seqw = null ;
public static void call_seqw ( String Prdbms , String lru , String Pusername , String Ppassword )
{
rdbms = Prdbms ;
url = lru ;
username = Pusername ;
password = Ppassword ;
ECAAgent myAgent = ECAAgent.initializeECAAgent ( "ecaoracle_ORCL" ) ;
// event seqw = delw >> addw;
EventHandle seqw = myAgent.createCompositeEvent(EventType.SEQ, "seqw"
,c_delw.delw ,c_addw.addw);
// rule Rule_seqw [ seqw , c_seqw.True , c_seqw.seqwecaoracle , 1 , IMMEDIATE ,
//RECENT ];
myAgent.createRule("Rule_seqw",seqw, "c_seqw.True", "c_seqw.seqwecaoracle", 1,
CouplingMode.IMMEDIATE, ParamContext.RECENT);
}
public static boolean True ( ListOfParameterLists parameterLists ) {
System.out.println ( "*****FromCondition*****" ) ;
return true ;
}
```

Figure 5-6 (Java file generated by ECA Agent by appending user actions and ECA actions)

```
public static void seqwecaoracle(ListOfParameterLists paramLists) {
 Enumeration en = paramLists.elements();
Vector sysContext = new Vector(10,2);
  while (en.hasMoreElements()) {
     ParameterList pl = ((ParameterList)en.nextElement());
     try { System.out.println("inside .....");
           String eventname = pl.getObject("eventname").toString();
           String tablename = pl.getObject("tablename").toString();
            int vno = ((Integer)pl.getObject("vno")).intValue();
            sysContext.addElement(eventname);
            sysContext.addElement(tablename);
            sysContext.addElement(new Integer(vno));
            ctc = "insert into SYSCONTEXT values('" + tablename + "','RECENT'," + vno + ")";
            ctrigger = new Jdbc(rdbms,url,username,password,ctc);
            ctrigger.ExecuteSqlUpdate("insert into table SysContext ");
        } catch(TypeMismatchException e) {System.out.println("Error in get Object in
checking Parameter"); }
          catch(ParameterNotFoundException ee) {System.out.println("Error Parameter");}
      }
 String info = "";
for (int i=0;i<sysContext.size();i++)
   info = info + " " + sysContext.elementAt(i).toString();
     String spc0 = "delete from weather_deleted_tmp";
    Jdbc storedProCom0 = new Jdbc(rdbms,url,username,password, spc0);
    storedProCom0.ExecuteSqlUpdate("delete from weather_deleted_tmp");
    spc0 = "insert into weather_deleted_tmp select * from weather_deleted where
sysContext.context='RECENT' and sysContext.TABLENAME='weather' and
weather_deleted.vNo=sysContext.vNo";
    storedProCom0 = new Jdbc(rdbms,url,username,password, spc0);
    storedProCom0.ExecuteSqlUpdate("insert into weather_deleted_tmp");
    String spc1 = "delete from weather_inserted_tmp";
    Jdbc storedProCom1 = new Jdbc(rdbms,url,username,password, spc1);
    storedProCom1.ExecuteSqlUpdate("delete from weather_inserted_tmp");
    spc1 = "insert into weather_inserted_tmp select * from weather_inserted where
sysContext.context='RECENT' and sysContext.TABLENAME='weather' and
weather_inserted.vNo=sysContext.vNo";
    storedProCom1 = new Jdbc(rdbms,url,username,password, spc1);
    storedProCom1.ExecuteSqlUpdate("insert into weather_inserted_tmp");
    //action function
     String spc = "insert into temp values('t_seqw',sysdate)";
     Jdbc storedProCom = new Jdbc(rdbms,url,username,password, spc);
     storedProCom.ExecuteSqlUpdate( " insert into table ");
     spc = "insert into notifyCom values('seqw','" + info + "')";
     storedProCom = new Jdbc(rdbms,url,username,password, spc);
     storedProCom.ExecuteSqlUpdate( " update notifyCom ");
     System.out.println ("****From Composite Event seqw Action of Rule****");
    }}
```

Figure 5-6 (continued)

As shown in the figure, the action portion of the composite event trigger will have the following main portions:

- It will update the contents of the table SysContext with the table name and version number of the two constituent primitive events whose occurrence resulted in the detection of the composite event. These tuples will also have the context in which the composite event was detected.
- It will delete the contents of the R_Inserted_tmp and R_Deleted_tmp tables.
- Following this, tuples corresponding to the primitive event occurrences that resulted in the detection of composite event (in the particular context) are fetched from R_Inserted and R_Deleted table and inserted into R_Inserted_tmp and R_Deleted_tmp tables. In order to make sure that proper tuples are selected, the version number of the constituent primitive event is used. This will help in isolating the particular tuple of interest from other tuples, which might have been present because of the occurrence of the same primitive event. The contents of R_Inserted_tmp and R_Deleted_tmp are as shown.
- Next, the action specified by the user in the composite event trigger action portion is executed.
- Finally, the table NotifyCom is updated in order to inform the ECA Agent and hence the client about the detection of the composite event.

5.2.1.5 Persist Event Information

This is the final step in the creation of a composite event trigger. In this step, we persist the composite event information in the tables SysCompositEvent, SysECATrigger and SysDrop. Finally, we associate the trigger name with the rule name and persist this information in the SysRuleTrigger table.

The contents of the tables SysCompositEvent and SysECATrigger are used for validating the trigger name and the event name when a new composite event trigger is created. The contents of the table SysDrop are used while dropping a primitive (or composite event). The contents of the table SysRuleTrigger is used to get the name of the rule while dropping composite and repeat-composite event triggers.

The content of these tables after creating the composite event trigger (shown in Figure 5-1) is shown in Table 5-1, Table 5-2, Table 5-3 and Table 5-4.

Table 5-1 (SysEcaTrigger)

Dbname	username	Triggername	Triggerproc	timestamp	eventname
Orcl	ecaoracle	t_seqw	NULL	22-Jan-02	seqw

Table 5-2 (SysCompositEvent)

Dbname	username	eventname	Eventdescribe	timestamp	coupling	context	priority
Orcl	ecaoracle	seqw	delw >> addw	22-Jan-02	Immediate	Recent	1

Table 5-3 (SysRuleTrigger)

TriggerName	EventName	RuleName
t_seqw	Seqw	Rule_seqw

Table 5-4 (SysDrop)

Conseventname	Context	Comeventname
addw	Recent	seqw
delw	Recent	seqw

5.2.2 Composite Event Notification

Whenever a primitive event is detected, the occurrence of this primitive event needs to be propagated to all those composite events that have subscribed to the primitive event. In the composite event node, the semantics associated with that composite event is checked, and if it is satisfied, the composite event is raised. The rule scheduler schedules the actions associated with a composite trigger. The Figure 5-7 shows how a composite event node is represented inside Java LED and how the primitive events are propagated.

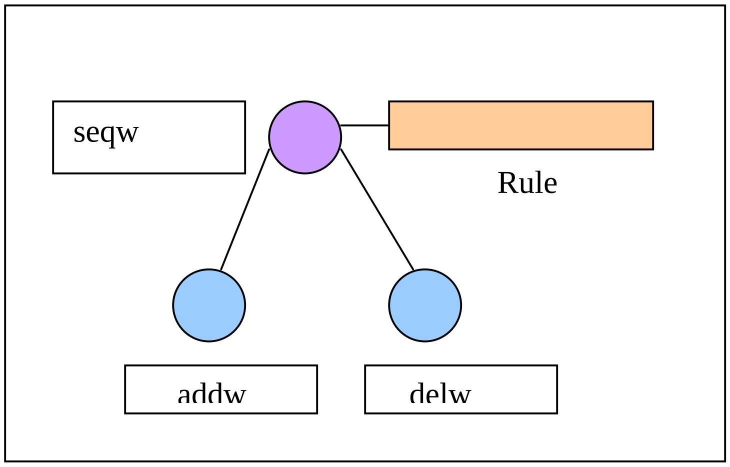

Figure 5-7 (Event graph in LED after creating a composite event)

5.3 Repeat Composite Events

The syntax for creating a repeat composite trigger is the same as it is for creating a repeat primitive trigger. Figure 5-8 depicts the syntax of creating a repeat trigger on a composite event.

```
create trigger t2_seqW event seqW
Begin
            Insert into temp values (‘t2_seqW’, sysdate);
end;
```

Figure 5-8 (Repeat Composite Event trigger – syntax)

As shown in the figure, the repeat trigger syntax neither specifies the event description, nor the context, coupling mode or the priority information. ECA parser extracts this information and creates the appropriate trigger. Below, we present the different steps involved in the creation of a repeat trigger on a composite event.

5.3.1 Trigger Validation

First, the ECA Agent checks to see if the trigger statement has the keyword *event* and the *event name* specified. If not, it returns an error message. Next, the ECA Agent checks the SysECATrigger table to see if this is a duplicate trigger. If so, it returns an error message. Finally, the ECA Agent checks the SysCompositeEvent to see if the event has been specified. It returns an error message if no event has been specified.

5.3.2 Extract the action part and create a rule

In this step, the action specified by the user in the repeat trigger statement is extracted. Next, we extract the parameters associated with this composite event from the SysCompositeEvent table. We then create a new rule to fire the new action portion specified in the repeat trigger. The new rule created will have a name that is a combination of the trigger name and the event name. This will ensure that all the rules that are created on a particular composite event as a result of specifying repeat triggers are still going to have a unique name. The new rule will have the form shown below:

```
/*
 rule Rule_t1_seqw_seqw [ seqw , c_seqw.True , c_seqw.t1_seqw_seqwecaoracle , 1 , IMMEDIATE ,
RECENT ];
*/
myAgent.createRule("Rule_t1_seqw_seqw ",seqw, "c_seqw.True", "c_seqw. t1_seqw_seqwecaoracle ", 1,
                                                    CouplingMode.IMMEDIATE, ParamContext.RECENT);
```

Once the rule is created, the next step would be to write the rule into the ".java" file i.e. *c_triggername.java*. This will ensure a distinct file name for each new repeat composite event trigger that the user is going to create. After, we write the rule into the ".java" file, we need to compile the file and register the composite event. Once this is done, we need to register all the rules that have been created as result of specifying repeat composite triggers.

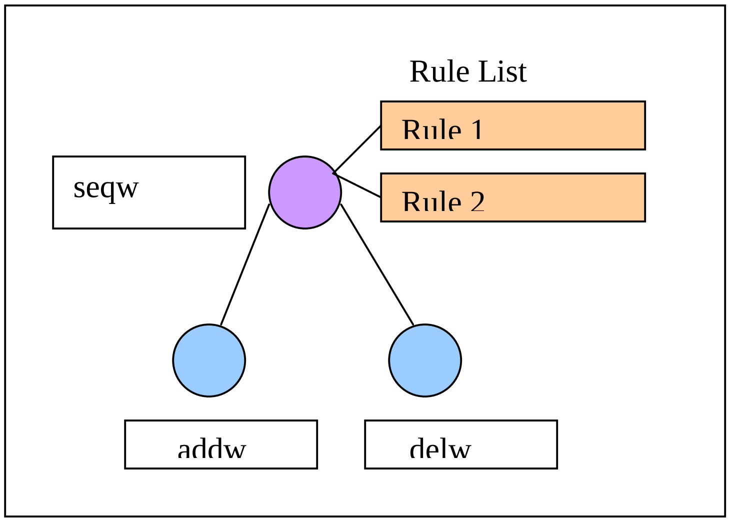

Figure 5-9 (Event graph in LED after repeat composite event trigger is created)

This completes the creation of a repeat composite event. Whenever a composite event is now detected, all the rules associated with that occurrence and which satisfy the semantics associated with the occurrence are fired.

Now that we have presented the steps for creation and detection of composite and repeat-composite events, we will explain the procedure for dropping the composite events and the repeat-composite events.

5.3.3 Persisting the Trigger information

After the composite event is created, we need to persist the trigger and event information. In this step, we persist information in the system tables i.e. SysCompositEvent, SysECATrigger and SysRuleTrigger. As explained earlier, the contents of the tables SysECATrigger and SysCompositEvent are used for authenticating the trigger name and event name. The contents of the table SysRuleTrigger is used while dropping the composite event triggers.

5.4 Dropping a Composite Trigger

Dropping a Composite event involves the same steps as with the Primitive event. We outline these steps below.

- Once the control is transferred to the Drop Composite Trigger module, it checks the table, SysDrop, to see if this Composite Event is a component event of any other composite events. If so, it informs the user that this trigger cannot be dropped.
- If the event is not a constituent event of any other event, then the ECA agent drops the trigger corresponding to it from all the system tables i.e. SysEcaTrigger, SysCompositeEvent and SysDrop.
- Next, the ECA Agent extracts the name of the event on which the trigger has been created from the table SysEcaTrigger. The ECA Agent now checks the table SysEcaTrigger to see how many triggers have been defined on a given event. If there is more than one trigger defined on the given event, then the ECA Agent should disable the rule that corresponds to the given trigger. So, the ECA Agent queries the SysRuleTrigger table to get the rule that corresponds to the given trigger. It then disables this rule in the LED and follows it up by deleting the tuple

corresponding to it from both SysECATrigger and SysRuleTrigger tables. Finally, if the trigger being dropped is a repeat composite event trigger, we delete the files generated for executing the action portion.

- However, if the SysECATrigger has just one trigger associated with a given event, then the ECA Agent decides that the node corresponding to that composite event has to be deleted from the event graph. Consequently, it performs this action by invoking the API for deleting a node from the event graph. Following this, the tuples corresponding to this trigger is deleted from the tables SysEcaTrigger, SysRuleTrigger and SysCompositEvent. Finally, the ".java" and the ".class" files corresponding to this event are deleted.

This completes the process of dropping a composite trigger.

In this chapter, we discussed the process of creating both composite and repeat-composite events. We also explained how composite events are dropped. This culminates our discussion on how we have been able to provide a RDBMS with active capability using mediator approach based Generalized ECA Agent. In the next chapter, we present the conclusions, contributions and provide a few pointers to the future work involved in this area.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this thesis we have explored the shortcomings that existed in the previous version of the ECA Agent and tried to overcome these shortcomings. In short, the following are the contributions of this thesis.

ECA Agent is complete in terms of its capability to support the trigger specifications provided by the vendor (Oracle). The ECA Agent, at present, supports the following functionalities.

- It supports both row level and statement level triggers created on the table.
- It supports triggers created on column updates.
- It supports triggers created on multiple operations.
- It supports composite event creation and detection in all the four contexts.
- It supports the following composite events at this time; OR, AND, SEQ, NOT, A, A*.
- It supports repeat composite events.
- It supports the dropping of both primitive and composite events as well as repeat primitive and repeat composite events.

In this thesis, we have explained the different modules that together make up the ECA Agent and also explain how the functionalities supported by the ECA Agent has been achieved.

6.2 Future Work

Though the trigger set supported by the ECA Agent has definitely been improved, the ECA Agent still needs to support certain other types of triggers like INSTEAD OF triggers and CREATE or REPLACE triggers.

In this thesis, we have used Oracle as the test bed. This can be further expanded so that the ECA Agent supports any RDBMS with certain minor modifications.

The result obtained after a trigger is fired is sent only to the person who has created the trigger. However, we need to explore if the result can be propagated to other users who have subscribed to the occurrence of the particular event. This is a potential area of interest in which further research can be carried out.

Currently, we cannot form a distributed active system that includes different RDBMSs. This is currently supported for stand-alone applications. We plan on subscribing to triggers from one application (Oracle) to another (Sybase).

APPENDIX
SYSTEM TABLES – SCHEMA

1. SYSPRIMTIVEEVENT

    DBNAME VARCHAR (30) NULL

    USERNAME VARCHAR (30) NULL

    EVENTNAME VARCHAR (30) NULL

    TABLENAME VARCHAR (30) NULL

    OPERATION VARCHAR (30) NULL

    BEAFOPERATION VARCHAR (30) NULL

    TIMESTAMP DATE NULL

    VNO INT NULL

    COLUMNNAMES CHAR (50) NULL

    TYPE CHAR (5) NOT NULL

2. SYSCOMPOSITEEVENT

    DBNAME VARCHAR (30) NULL

    USERNAME VARCHAR (30) NULL

    EVENTNAME VARCHAR (30) NULL

    EVENTDESCRIBE VARCHAR (30) NULL

    TIMESTAMP DATE NULL

    COUPLING CHAR (10) NULL

    CONTEXT CHAR (12) NULL

PRIORITY CHAR (10) NULL

3. SYSECATRIGGER

    DBNAME VARCHAR (30) NULL

    USERNAME VARCHAR (30) NULL

    TRIGGERNAME VARCHAR (30) NULL

    TRIGGERPROC VARCHAR (30) NULL

    TIMESTAMP DATE NULL

    EVENTNAME VARCHAR (30) NULL

4. SYSCONTEXT

    TABLENAME CHAR (20)

    CONTEXT CHAR (12)

    VNO INT NULL

5. SYSDROP

    CONSEVENTNAME CHAR (30)

    CONTEXT CHAR (12)

    COMPEVENTNAME CHAR (30)

6. SYSRULETRIGGER

    RULENAME CHAR (50)

    EVENTNAME CHAR (50)

    TRIGGERNAME CHAR (50)

7. VERSION

    VNO INT NULL

8. NOTIFY

    EVENTNAME VARCHAR (30) NULL

    TABLENAME VARCHAR (30) NULL

    VNO INT NULL

9. NOTIFYCOM

    COMEVENTNAME VARCHAR (50)

    INFO VARCHAR (250)

APPENDIX B
ECA AGENT WORKING – PSEUDOCODE

1. ECAAGENT LISTENER

```
listen to Client request at specified port
accept Client request and spawn a new dedicated thread for every new client
if (client logging for the first time)
        create system tables
        create system wide unique directory structure to place files
else
        check for the existence of unique system wide directory structure
        if (directory does not exist)
                create directory
delete the contents of Notify and Notifycom tables
invoke persistence manager
restore all rules (re-create the event graph for a repeat user)
        if temporary tables exist initialize the temporary tables (delete all values)
        reset version number in version table to 0
invoke language filter and pass client request and wait for return of control
check Notify table by extracting all event names
if (Notify has contents)
```

```
        extract event name(s), table name(s) and version  number(s) from the
Notify table
        (more than one tuple present only if both update trigger and update of
    column trigger defined)
        for each tuple present in the Notify table
            invoke an instance of LED class.
            invoke the method in LED class for
         registering the primitive event occurrence in LED
    check Notifycom table for contents
    if (Notifycom has contents) {only if composite events have been detected}
        inform client about the detection of composite event
    stop the thread and wait for a new request.
```

2. Creation of a Primitive Event

```
    ECA Agent Listener forwards Client request to language filter
    language filter:
    if (1st keyword is “create”, 2nd keyword is “trigger” and fourth keyword is not
“event”)
    forward client request to Primitive Event Parser
    Primitive Event Parser:
    if (trigger name is duplicate)
        return error message
    if (event name is duplicate)
        return error message
```

```
get the system wide unique directory name of the form username_database
generate the string that contains the API for registering the primitive event
generate the ".java" file
compile the ".java" file to obtain the ".class" file.
execute the method in the ".class" file and register the primitive event.
        {arguments passed will be class name, method name and NULL for
parameters}
if (temporary tables do not exist)
        create temporary tables
invoke persist manager:
        extract sql extension
        create trigger for performing ECA actions[3]
        create trigger for performing user actions
        persist event information in System tables
return control to ECA Agent Listener
```

3. Trigger for performing ECA actions

```
update SysPrimtiveEvent by incrementing version number
update Version by incrementing version number
insert parameters into Notify table (event name, table name, version no)
update temporary tables (R_Inserted or R_Deleted) with the new column
        values and the latest version number
```

4. Creation of a Repeat Primitive Event

```
ECA Agent Listener forwards Client request to language filter
```

```
language filter:
if  (if 1st keyword = create, 2nd keyword = trigger, 4th keyword = event)
    check SysPrimitiveEvent for the existence of event
    if (event exists)
        pass control to Repeat Primitive Event Parser
    else
        pass control to Repeat Composite event Parser
Repeat primitive event parser:
if (duplicate trigger name)
    return error message
if (duplicate event name)
    return error message
get table name and operation on which the primitive event trigger has been defined.
get time of primitive event trigger execution (After or Before)
invoke persistence manager for creating the trigger
Persistence Manager:
remove sql extensions
append extracted information to the trigger statement
create the trigger in the SQL server
persist event information in System tables.
```

5. Creation of a composite event

   ECA Agent forwards Client request to language filter

```
language filter:
if (1st  keyword = “create”, 2nd keyword = “trigger”, 4th keyword = “event” and 5th
keyword = “=”)
forward client request to Composite event parser
Composite event parser:
break the composite event trigger string into two parts (before and after colon)
    {2nd half has event detection info and user action part, 1st half has trigger
name and event description}
if (duplicate trigger name)
    return error message
if (duplicate event name)
    return error message
extract the event description string
extract event detection info from 2nd half of trigger
    if (context not specified)
        assign default context
    if (coupling not specified)
        assign default coupling
    if (priority not specified)
        assign default priority
create the rule string from event description and event detection info
generate a “.sjava” file and write the event description and rule string
submit “.sjava” file to Snoop Pre-processor
```

Snoop pre-processor:

extract constituent event from event description string.

create the API for registering the composite event from the rule string.

return a text file-containing list of all constituent events

return ".java" file containing the API for registering the composite event with LED

return control to Composite event parser

Composite event parser (cont.):

check for the existence of constituent events

append ECA action [6] and user action to ".java" file

compile the ".java" file to get the ".class" file

execute the method in the ".class" file for registering a composite event

{parameters passed are method name and type of argument list}

invoke Persistence Manager:

persist event information and trigger information in System tables

persist rule, event, trigger information in SysRuleTrigger

persist constituent event information in SysDrop

6. ECA Actions for a Composite event trigger:

update Syscontext with parameters associated with detection of composite event

delete contents of transient tables

update transient tables by getting values from SysContext and R_Inserted/Deleted tables based on an equi join (on version number) between SysContext and Version table update Notifycom with the tuple corresponding to the detection of that composite event.

7. Creation of a Repeat composite event trigger

```
ECA Agent Listener forwards Client request to language filter\
language filter:
if (1st keyword = “create”, 2nd keyword = “trigger” and 4th keyword = “event”)
check SysPrimitiveEvent for the existence of event
    if (event exists)
        pass control to Repeat Primitive Event Parser
    else
        pass control to Repeat Composite event Parser
Repeat composite event parser:
extract ECA info (context, coupling, priority) from SysCompositEvent based on evenname
create a new rule string with unique rule name (triggername_eventname)
extract user action.
insert user action into unique method (call_triggername_eventname)  which is called from within the new  rule.
get the length of the “.java” file created for  composite event trigger.
Start writing the new rule API and the method at this point. (this will overwrite the  last character “which is the closing brace for that class”)
insert the closing brace into the file
compile the “.java” file to get the “.class” file
```

register the new rule with the LED.

8. Dropping a primitive event trigger

```
ECA Agent Listener forwards Client request to language filter
language filter:
if (1st keyword = “drop”, 2nd keyword = “trigger”)
    check SysECATrigger to see if an event has been defined
    if (event not defined)
        send command to sql server by making a JDBC call\
    else
        check if event name present in SysPrimtiveEvent
        if (present)
            forward command to Drop Primitive trigger
        else
            forward command to Drop composite trigger
Drop Primitive event trigger:
check SysDrop to see if primitive event is a constituent event of a composite event
if (constituent event)
    not possible to drop primitive event
    return message
else
    delete corresponding tuple from SysECATrigger
    drop primitive event trigger (created for user action part)
```

```
        check SysECATrigger to see  #of triggers defined on that event
        if (#of triggers >1)
                repeat triggers defined on primitive event
                cannot drop trigger created for ECA action
                return control to ECA Agent Listener
        else
                no repeat triggers defined on primitive event
                delete ECA action trigger from SysECATrigger
                delete tuple from SysPrimtiveEvent
                drop trigger from SQL server
                delete the ".java" file and the ".class" file.
                return control to ECA Agent listener
```

9. Dropping a composite event trigger

```
ECA Agent Listener forwards Client request to language filter\
language filter:
if (1st keyword = "drop", 2nd keyword = "trigger")
        check SysECATrigger to see if an event has been defined
        if (event not defined)
                send command to sql server by making a JDBC call\
        else
                check if event name present in SysPrimtiveEvent
                if (present)
                        forward command to Drop Primitive trigger
```

```
          else
                    forward command to Drop composite trigger
Drop composite event trigger:
check SysECATrigger and get the number of triggers defined on the event
if (#of triggers == 1)
          delete tuple from SysECATrigger, SysCompositeEvent, SysDrop and
SysRuleTrigger
          delete ".java", ".txt" and ".class" files created for composite event trigger
          invoke API to delete composite event node from LED
else
          get rule name assocoiated with trigger from SysRuleTrigger
          invoke API for disabling the rule in LED.
          delete tuple from SysRuleTrigger and SysEcaTrigger
```

LIST OF REFERENCES

1. Chakravarthy, U.S., *{Rule management and Evaluation: An Active DBMS Perspective}*. Special issue of ACM Sigmod Record on rule processing in databases, 1989. **18**(3): p. 20--28.
2. Hanson, E.N., *Rule Condition Testing and Action Execution in {A}riel*, in *Proc 1992 ACM-SIGMOD Conf. on Management of Data*. 1992: San Diego, California.
3. Chakravarthy, S., E. Anwar, and L. Maugis, *Design and Implementation of Active Capability for an Object-Oriented Database*. 1993, Tech. Report, University of Florida: Gainesville.
4. Widom, J., R.J. Cochrane, and Lindsay, *{Implemented Set-Oriented Production Rules as an Extension of Starburst}*, in *{Proceedings 17th International Conference on Very Large Data Bases}*. 1991: Barcelona (Catalonia, Spain). p. 275--286.
5. Stonebraker, M. and G. Kemnitz, *The Postgres Next-Generation Database Management System.* Communications of the ACM, 1991. **34**(10): p. 78--92.
6. Gatziu, S. and K.R. Dittrich, *SAMOS: an Active, Object-Oriented Database System.* in IEEE Quarterly Bulletin on Data Engineering, 1992. **15**(1-4): p. 23--26.
7. Li, L. and S. Chakravarthy. *An Agent-Based Approach to Extending the Native Active Capability of Relational Database Systems*. in *ICDE*. 1999. Australia: IEEE.
8. Song, Z., *A Generalized Approach For Extending The Active Capability Of RDBMSs*, in *Database Systems R&D Center, CISE Department*. 2000, University of Florida: Gainesville.
9. Kim, Y., *A Generalized Active Agent System For Extending The Active Capabilities Of A RDBMS*, in *Database Systems R&D Center, CISE Department*. 2000, University of Florida: Gainesville.
10. Brownstorm, L., et al., *{Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming}*. 1985: Addison Wesley.
11. Stonebraker, M., L. Rowe, and M. Hirohama, *The Implementation of {POSTGRES}*. IEEE Transactions on Knowledge and Data Engineering, 1990. **2**(1): p. 125--142.
12. Peterson, J.L., *{``Petri Nets.''}*. Computing Surveys, 1977. **9**(3): p. 23--252.
13. Gopalakrishnan, G., *Making Sybase fully Active: Supporting Composite events and Prioritized rules*, in *ITLAB, CSE Department*. 2002, University of Texas at Arlington: Arlington.
14. Chakravarthy, S. and D. Mishra, *Snoop: An Expressive Event Specification Language for Active Databases.* Data and Knowledge Engineering, 1994. **14**(10): p. 1--26.

15. Krishnaprasad, V., *Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation*, in *MS Thesis*. 1994, Database Systems R&D Center, CIS Department, University of Florida, Gainesville, FL 32611.

16. Subramaniam, N., *A mediator based approach to support ECA rules in DB2 RDBMS*. 2002, The University of Texas at Arlington: Arlington.