AFRL-IF-RS-TR-2001-271
Final Technical Report
January 2002

# GENERALIZED ACTIVE AGENT SYSTEM FOR EXTENDING THE ACTIVE CAPABILITIES OF RELATIONAL DATABASE SYSTEM (RDBMS)

University of Texas at Arlington

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

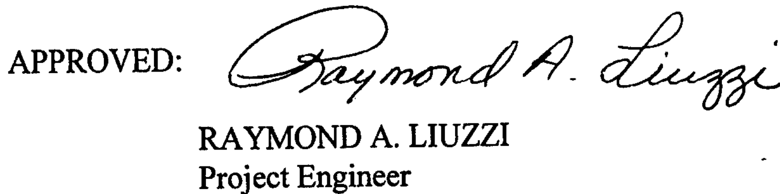
AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

20020405 033

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

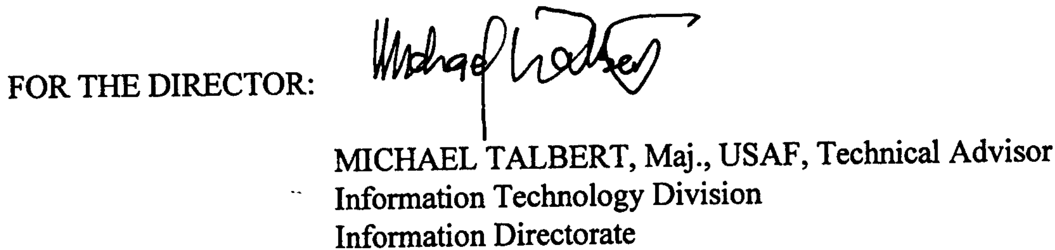
AFRL-IF-RS-TR-2001-271 has been reviewed and is approved for publication.

APPROVED:



RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR:



MICHAEL TALBERT, Maj., USAF, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

| REPORT DOCUMENTATION PAGE | *Form Approved* *OMB No. 0704-0188* |
|---|---|

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | JANUARY 2002 | Final Jul 00 - Sep 01 |

**4. TITLE AND SUBTITLE**
GENERALIZED ACTIVE AGENT SYSTEM FOR EXTENDING THE ACTIVE CAPABILITIES OF RELATIONAL DATABASE SYSTEM (RDBMS)

**5. FUNDING NUMBERS**
C - F30602-00-2-0604
PE - 62232N/62702F
PR - R427
TA - 00
WU - P4

**6. AUTHOR(S)**
S. Chakravarthy, Y. Kim, and G. Gopalakrishnan

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
University of Texas at Arlington
Computer Science and Engineering
416 Yates Street
Arlington Texas 76019-0015

**8. PERFORMING ORGANIZATION REPORT NUMBER**
N/A

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Air Force Research Laboratory/IFTD
525 Brooks Road
Rome New York 13441-4505

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
AFRL-IF-RS-TR-2001-271

**11. SUPPLEMENTARY NOTES**
Air Force Research Laboratory Project Engineer: Raymond A. Liuzzi/IFTD/(315) 330-3577

**12a. DISTRIBUTION AVAILABILITY STATEMENT**
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*
The objective of this effort was to investigate knowledge base management integration techniques for extracting intelligent information for a network-centered C4I environment. This report describes the problem of re-structuring a traditional database system into a true active database system without changing the semantics of the underlying system. The approach described provides a generalized event definition mechanism which extend the active capabilities of a RDBMS. This, in turn provides the mechanism whereby active database semantics can be supported on an existing SQL Server by an ECA Agent inserted between the SQL Server and its clients. Both a demonstration and results are also described.

**14. SUBJECT TERMS**
Computers, Software, Knowledge Bases, Data Bases, Artificial Intelligence

**15. NUMBER OF PAGES**
36

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

# Contents

1 INTRODUCTION 1
1.1 Advantage of Providing the Active Capability to a RDBMS 2
1.2 Reasons for Choosing Oracle as Test DBMS 2
1.3 Contributions of this work 2
2 OVERVIEW OF RELATED WORK 3
2.1 Sentinel 3
2.2 The GATEWAY Approach 3
3 DESIGN ISSUES OF THE ECA AGENT 4
3.1 The Architecture of the ECA Agent 4
3.2 Java Local Event Detector (Java LED) 6
3.3 Snoop Preprocessor 7
3.4 Java Database Connectivity (JDBC) 7
3.5 Interaction between the ECA Agent Modules 8
4 IMPLEMENTATION ISSUES OF THE ECA AGENT 9
4.1 Tables used 9
4.1.1 SysPrimitiveEvent 9
4.1.2 SysCompositeEvent 10
4.1.3 SysEcaTrigger 10
4.1.4 SysContext 11
4.1.5 EventContext 11
4.1.6 Version 12
4.2 The Language Filter and ECA Parser 12
4.3 Persistent Manager 14
4.4 Action Handling by the Java LED 15
5 IMPLEMENTATION OF PRIMITIVE EVENTS 15
5.1 ECA Agent 15
5.2 Syntax of Primitive Events 15
5.3 Processing of a Primitive Event 16
5.4 Syntax for specifying multiple triggers on the same event 19
5.5 Syntax for Dropping a Trigger 20

**6 CONCLUSIONS AND FUTURE WORK** 20

6.1 Future Work ........ 20

**7 REFERENCES** 21

## List of Figures

Figure 1 **The Gateway Architecture** ........................................ 4

Figure 2 **General ECA Agent System** ........................................ 5

Figure 3 **The Architecture of ECA Agent System** ........................................ 5

Figure 4 **Global Event History** ........................................ 7

Figure 5 **Language and Functional Modules in ECA Parser** ........................................ 13

Figure 6 **The Architecture of Persistent Manager** ........................................ 14

Figure 7 **Oracle SQL Statement** ........................................ 16

Figure 8 **Primitive Event Definition using Oracle SQL Statement** ........................................ 16

Figure 9 **Parsing and Generating of Primitive Event** ........................................ 17

Figure 10 **Notification Check** ........................................ 18

Figure 11 **Syntax of Defining a Trigger on Existing Event** ........................................ 19

Figure 12 **The Changed Event Definition on Existing Event** ........................................ 19

# List of Tables

Table 1 SysPrimitiveEvent 9
Table 2 SysPrimitiveEvent after the definition of event addStk 9
Table 3 SysCompositeEvent 10
Table 4 SysCompositeEvent after the definition of event addDel 10
Table 5 SysEcaTrigger 10
Table 6 SysEcaTrigger after the definition of event, addStk 11
Table 7 SysContext 11
Table 8 SysContext after addDel related to Recent context 11
Table 9 EventContext 11
Table 10 EventContext table after the composite event, addDel 12
Table 11 Version 12
Table 12 Stock_inserted and Stock-deleted 17
Table 13 SysEca Trigger after the existing event (addStk) definition 19

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


## Abstract

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

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

## 1 INTRODUCTION

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

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

## 1.1 Advantage of Providing the Active Capability to a RDBMS

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

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

## 1.2 Reasons for Choosing Oracle as Test DBMS

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

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

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

## 1.3 Contributions of this work

The contributions of this work are:

1. A client can create multiple triggers on the same event.
2. A client can create composite events and triggers on them.
3. Previously defined events can be reused.
4. Triggers associated with primitive or composite events can be dropped.
5. Once events are created, they can be persisted into the database system.
6. All primitive events and composite events can be detected, and actions are invoked in SQL server.

## 2 OVERVIEW OF RELATED WORK

### 2.1 Sentinel

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

- Primitive and composite event detection: Any method of any object class can be a primitive event. Before and after variants of method invocation are permitted as events. Composite events are formed by applying a set of operators such as AND, OR, SEQ, etc.
- Parameter contexts: The processing of composite events entails not only their detection, but also the computation of the parameters associated with the composite event. Parameter contexts are motivated by a careful analysis of several classes of applications.
- Online and batch detection of events: The composite event detector needs to support detection of events as they happen (online) when it is coupled to an application or in offline mode.
- Inter-application (global) events: A global event detector (or GED) provides support for composite events whose constituent events come from different applications.
- Multiple rules: An event can trigger several rules.
- Nested/cascading rules: Rule actions can raise events that trigger other rules.
- Coupling mode: Coupling modes refers to the execution point of condition/action pair relative to the event.
- Rule scheduling: In the view of multiple rules and nested execution, the architecture needs to support prioritized serial execution of rule, concurrent execution of all rules, or a combination of the two.

### 2.2 The GATEWAY Approach

The Gateway approach is implemented in [VAN96]. The Gateway architecture is a layered API approach. Figure 1 shows the Gateway architecture. The layer of the Gateway approach is transparent to the client. Open Server (ECA Server in the Figure 1) is used for this approach. The ECA Server receives a requirement and connects the SQL Server and sends all information to the SQL Server. The SQL Server authenticates the client and creates the connection. After connection is made, the Gateway takes control of the client process.

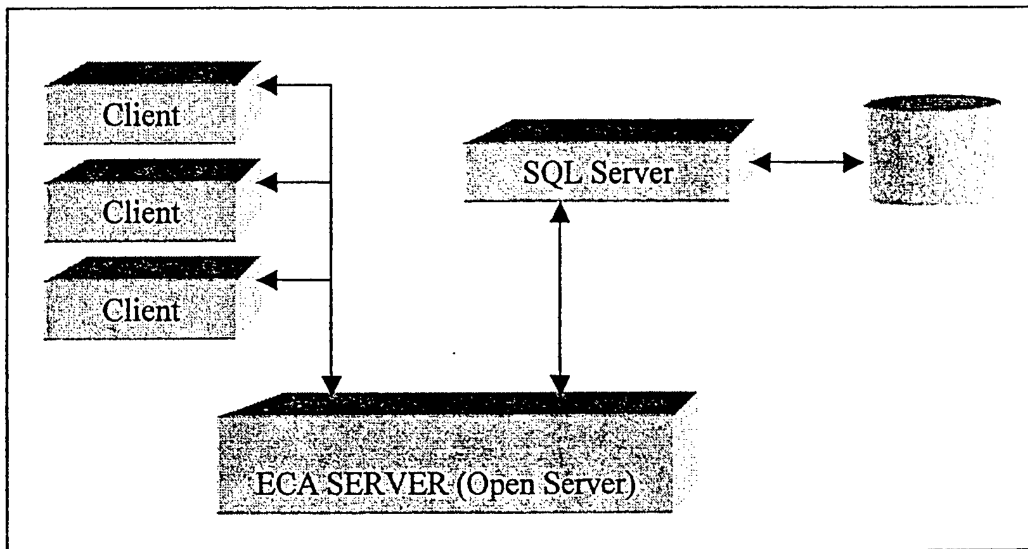



Figure 1 **The Gateway Architecture**

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

- According to the coupling mode, both immediate and deferred trigger semantics are implemented.
- Parallel execution of ECA Server and SQL Server is possible.
- Events are generated in the correct order.
- Events are not lost.
- Performance is enhanced.

## 3 DESIGN ISSUES OF THE ECA AGENT

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

### 3.1 The Architecture of the ECA Agent

An ECA Agent is a general module in the sense that it can be used with any kind of relational DBMSs such as Oracle, DB2, Sybase, Informix, etc. In this section, we give the general design of an ECA Agent as it is shown in Figure 2. An ECA Agent provides the client interface. The ECA Agent (called the ECA Server) works between multiple clients and the SQL server. The message from clients is sent to the ECA Agent and then the ECA Agent sends it to SQL server. The Result from the SQL Server is returned to the client via an ECA Agent.

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

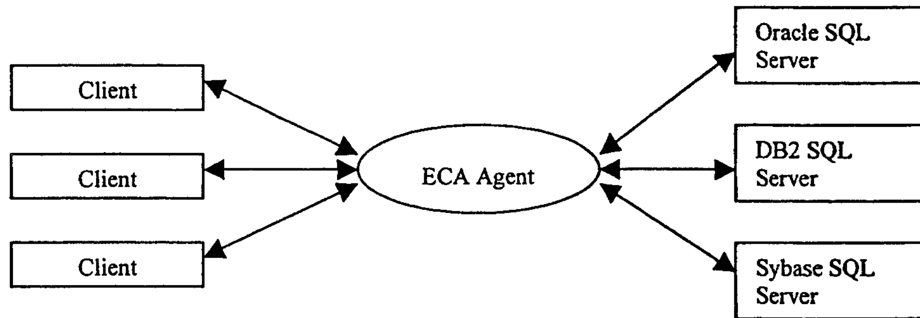



Figure 2 **General ECA Agent System**

From the point of the user, the ECA Agent works as a virtual active SQL Server. Not only can it provide all the native functions of a relational DBMS, but also it provides all the ECA active functions. Figure 3 shows the detailed architecture of the ECA Agent system.

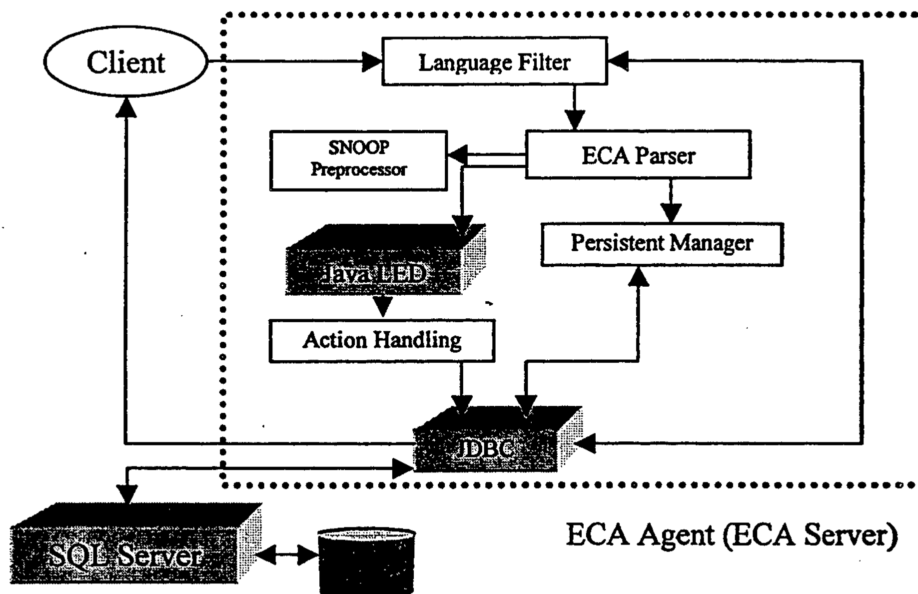



Figure 3 **The Architecture of ECA Agent System**

The ECA server has the following functional modules:

- Language Filter: First, all client requests are sent to the Language Filter. The Language Filter checks the request to determine if it is an ECA command (extended SQL command) or a pure SQL command. If it is an ECA command, it is sent to the

ECA Parser. If it is an SQL command, it is sent to the SQL Server through the Java database connection (JDBC).

- ECA Parser: An ECA command from the Language Filter is scanned and parsed. If there are no syntax errors, the parser generates the corresponding events and rules, which can be detected by the Java LED. The Parser sends the events and rules to the Persistent Manager.
- Java Local Event Detector: In a relational DBMS, triggers can detect only primitive events. We use the Java LED to detect and execute the rules (condition/action pair) associated with composite events.
- Persistent Manager: All events and rules need to be stored into the relational DBMS for subsequent use. The Persistent Manager stores all ECA information into system tables in a relational DBMS. The Persistent Manager restores the needed events and rules from these tables when the ECA Agent starts or recovers.
  The system tables include:
  1. SysPrimitiveEvent: This table stores primitive events.
  2. SysCompositeEvent: This table stores information about composite events.
  3. SysEcaTrigger: Every trigger in system is stored in this table.
- Java Database Connectivity (JDBC): JDBC is used to connect the ECA Agent to the SQL Server. Clients can request not only relational SQL commands but also user-defined commands through JDBC, and JDBC gets values from the SQL server, which is sent back to clients.
- Action Handling: When events occur, the related action defined for the event need to be executed. Actions can be the modification of a system table by SQL operation such as insertion, deletion, or update.

3.2 Java Local Event Detector (Java LED)

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

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

Figure 4 shows a global event history. In the Recent context, the most recent occurrence of an event is used to detect a composite event. The composite event A will include the event instances $\{e_1^2, e_2^1, e_3^1\}$ (A is detected when $e_3^1$ occurs) and $\{e_1^2, e_2^2, e_3^2\}$ (A is detected

again when $e_3^2$ occurs). In the Chronicle context, a parameter of event A is computed by using the event instances { $e_1^1$, $e_2^1$, $e_3^1$ }. In the Continuous context, the first occurrence of A has the instances { $e_1^1$, $e_2^1$, $e_3^1$ }. The second occurrence of A consists of the event instances { $e_1^2$, $e_2^1$, $e_3^1$ }. In the Cumulative context, all occurrences of an event are accumulated as instances of the event A when the event is detected.

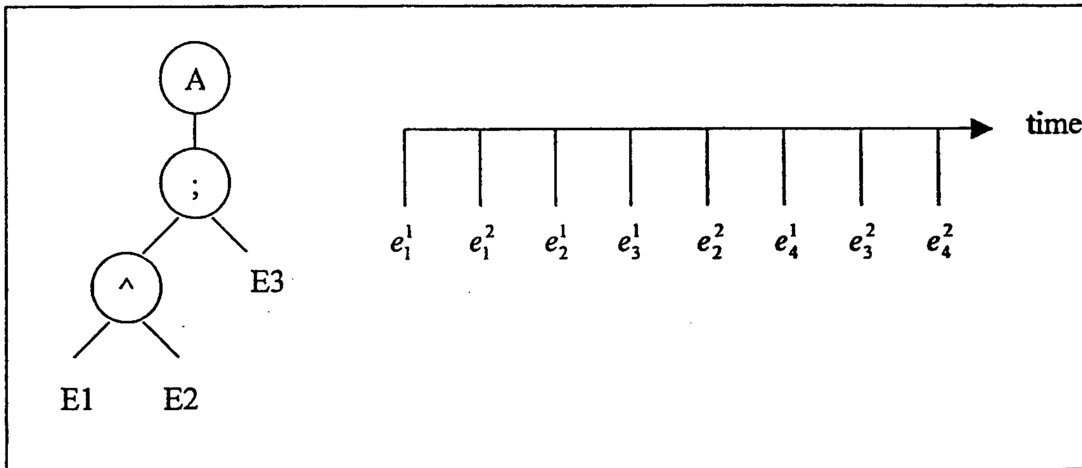



Figure 4 **Global Event History**

3.3 Snoop Preprocessor

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

3.4 Java Database Connectivity (JDBC)

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

1) Loading the Drivers: Loading the driver or drivers takes one line of code. If, for example, we want to use the Oracle driver, the following code loads it: Class.forName ("oracle.jdbc.driver.OracleDriver");
When we have loaded a driver, it is available for making a connection with a DBMS.

2) Making the Connection: The second step in establishing a connection is to have the appropriate driver connect to the DBMS. The following line of code illustrates the general idea:
Connection con=DriverManager.getConnection(url,"Login","Password");.
If we want to use Oracle as a test DBMS, the url should be
"jdbc:oracle:thin:@ tokyo.dbcenter.cise.ufl.edu :1521:ORCL."
3) Creating the JDBC Statements: A Statement object is what sends the SQL statement to the DBMS. We simply create a Statement object and then execute it, supplying the appropriate executes method with the SQL statement we want to send. For a SELECT statement, the method to use is executeQuery. For statements that create or modify tables, the method to use is executeUpdate. The following line of code is used to create the Statement object "stmt":
Statement stmt = con.createStatement();.
We need to supply this statement to the method we use to execute this statement:
stmt.executeQuery("select * from portfolio");.
We can get the result from the RDBMS using the following line of code:
ResultSet rs= stmt.executeQuery("select * from portfolio");.

## 3.5 Interaction between the ECA Agent Modules

In this section, we discuss how each ECA Agent module works and interacts. If a user sends his request to the Language Filter, the Language Filter filters the user request to determine if the user request is a SQL command (insert, delete, update, select, etc) or it is ECA command (extended create trigger statement which includes event definition). SQL commands are directly sent to SQL Server through JDBC. ECA commands, on the other hand, are sent to the ECA Parser. The ECA Parser scans ECA command and checks if the trigger name and the event name of the event are duplicates. Then the ECA Parser generates a Java source file. For example, if the primitive event name is addStk and the user name is ykim, the Java file name is ykimaddStk.java. After the file ykimaddStk is generated, it is compiled (we use the method exec () of java.lang.Runtime class to compile a Java source file. Refer to the Compile File method of ConstructClass.java) and then the method, call_addStk is invoked by dynamic method call using ExecuteMethod() of CallDynamicMethod class (we use the method invoke() of ClassLoader class).

The reason for generating, compiling the Java source file, and invoking the method is to obtain the event handle of the event so as to detect a composite event related to the primitive event (refer to Figure 4). If we have another primitive event, delStk, the processing steps on the primitive event, delStk in the ECA Parser is the same as those of the primitive event, addStk. If we have the composite event, addDel (this event needs both addStk and delStk to be detected by the Java LED), the Java source file, ykimaddDel.java is generated by the ECA Parser and compiled by the exec() method of java.lang.Runtime class. The call_addDel() method in the ykimaddDel.java file is invoked to initialize the Java LED and obtain the event handle of the composite event, addDel in the Java LED. The only difference in the file generated (ykimaddDel.java) between primitive and composite events is the addDel rule definition, which includes a condition and an action method. The Persistent Manager works on table modifications. It

stores trigger information of both primitive and composite events in the table, SysEcaTrigger, event information of primitive events in the table, SysPrimitiveEvent, and event information of composite events in the table, SysCompositeEvent (refer to section 4 on tables).

# 4 IMPLEMENTATION ISSUES OF THE ECA AGENT

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

## 4.1 Tables used

### 4.1.1 SysPrimitiveEvent

The table, SysPrimitiveEvent stores the information on primitive events. The structure of SysPrimitiveEvent is shown in Table 1.

Table 1 SysPrimitiveEvent

| dbname | username | eventname | tablename | Operation | Beafoperation | timestamp | vno |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

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

*Create trigger t_addStk after insert on stock event addStk …….*

The table, SysPrimitiveEvent contains data as shown in Table 2.

Table 2 SysPrimitiveEvent after the definition of event addStk

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

4.1.2 SysCompositeEvent

The table, SysCompositeEvent is used to store the information on composite events. The structure of this table is shown in Table 3.

Table 3 SysCompositeEvent

| dbname | username | eventname | eventDescribe | Timestamp | Coupling | context | Priority |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

In Table 3, the coupling mode should be one of IMMEDIATE, DEFERED, and DETACHED. The context can be one of RECENT, CHRONICLE, CONTINUOUS, or CUMULATIVE. The priority defines the priority of this composite event. The composite event definition, addDel is shown below. EventDescribe shows which primitive events are included to form this composite event and operator. A trigger with the composite event definition is shown below:

*Create trigger t_addDel event addDel = addStk ^ delStk RECENT IMMEDIATE 1....*

After this trigger is defined, the system table, SysCompositeEvent contains the data shown in Table 4.

Table 4 SysCompositeEvent after the definition of event addDel

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

4.1.3 SysEcaTrigger

This table is used to store the information on triggers created by the user. When a trigger is defined, a check for trigger duplication is performed by searching this table. The structure of the table, SysEcaTrigger is shown in Table 5.

Table 5 SysEcaTrigger

| dbname | Username | Triggername | TriggerProc | timestamp | eventname |
|---|---|---|---|---|---|
| | | | | | |

The column, TriggerProc in Table 5 contains the name of the procedure defined on a trigger. When the trigger fires, this procedure will be executed. Several different triggers can be defined on the same event.

Table 6 shows the SysEcaTrigger after the definition of event addStk.
Table 6 SysEcaTrigger after the definition of event, addStk

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

4.1.4 SysContext

The table, SysContext is used to store the occurrence number (Vno) of an event associated with a context. This table is used for keeping track of event occurrences to be used for detecting a composite event. The structure of this table is shown in Table 7.

Table 7 SysContext

| EventName | Context | Vno |
|---|---|---|
| | | |

Suppose the event, addDel associated with the context, RECENT has occurred three times. The following Table 8 shows the tuples inserted into SysContext table.

Table 8 SysContext after addDel related to RECENT context.

| EventName | Context | Vno |
|---|---|---|
| addDel | Recent | 1 |
| addDel | Recent | 2 |
| addDel | Recent | 3 |

4.1.5 EventContext

Table EventContext is used to store information on primitive events and contexts where a composite event occurs. The structure of the EventContext table is illustrated in Table 9.

Table 9 EventContext

| EventName | Context |
|---|---|
| | |

After the following create trigger statement is processed, the contents of the table, EventContext will be as shown in Table 10.

Create trigger t_addDel event addDel = addStk ^ delDtk RECENT ......

Table 10 EventContext table after the composite event, addDel

| EventName | Context |
| --- | --- |
| DelStk | Recent |
| AddStk | Recent |

4.1.6 Version

This table is used to store the occurrence number of a primitive event. Initially, this version number is zero. When a primitive event occurs, the version number of the primitive event is increased by 1. The version number is also stored into the SysPrimitiveEvent table. The structure of the table Version is shown in Table 11.

Table 11 Version

| VNO |
| --- |
| 1 |

The Version number is updated in the following way:

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

*delete from Version;*

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

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

4.2 The Language Filter and ECA Parser

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

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

In the ECA Parser, the ECA commands are filtered from the Language Filter. The ECA Parser tokenizes and parses the commands. The syntax is checked first. If there are no syntax errors, the ECA Parser will create corresponding events and rules that are compiled by the Java Local Event Detector (JavaLED). Figure 5 illustrates the Language Filter and the functional modules of the ECA Parser.

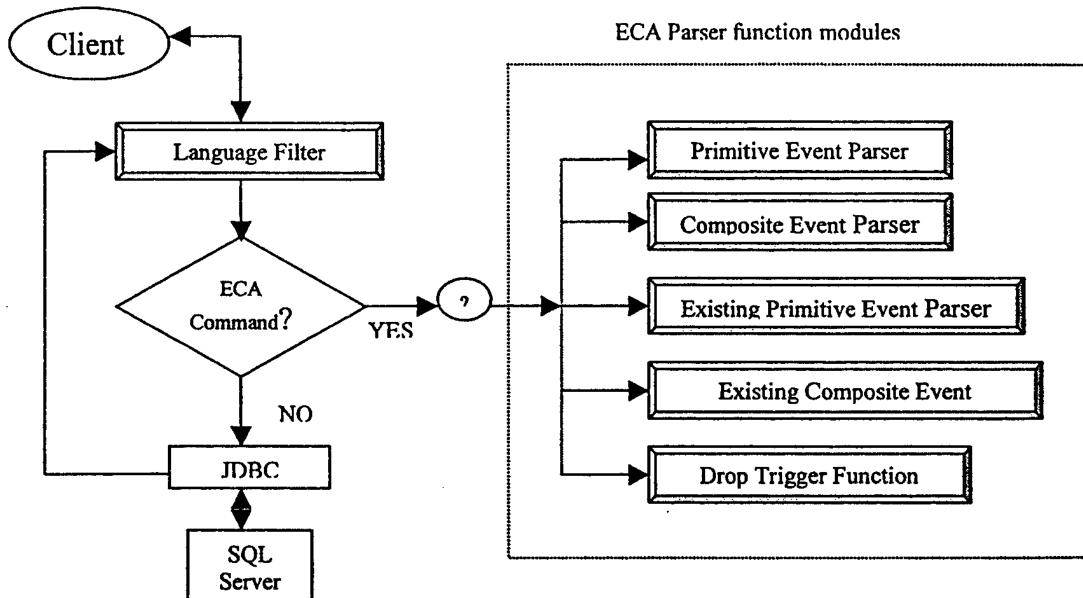



Figure 5 **Language and Functional Modules in ECA Parser**

There are five functional modules in the ECA parser:

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

4.3 Persistent Manager

All the data structures those are stored in memory for a program/application executes, goes away at the end of the program execution. That is, when the application program terminates, the memory used for that application is recovered by the operating system. Non-DBMS environments use the file system to store data across application runs. In a Relational DBMS, we can use tables to do this. The Persistent Manager does this as part of the ECA agent. The ECA rules handled by the ECA agent are kept in the RDBMS. The Persistent Manager is in charge of the management of the ECA rules. The list below shows the functions of the Persistent Manager:

- Maintain triggers: These triggers define the actions, which should be executed, when either primitive events or composite events occur.
- Maintain tables: The Persistent Manager, based on triggers, manages tables.
- Persist ECA rules: The information of all the ECA rules is stored into the Relational DBMS.
- Restoration of the ECA rules: When system restarts, all ECA rules are restored.
-

Figure 6 illustrates the Architecture of Persistent Manager.

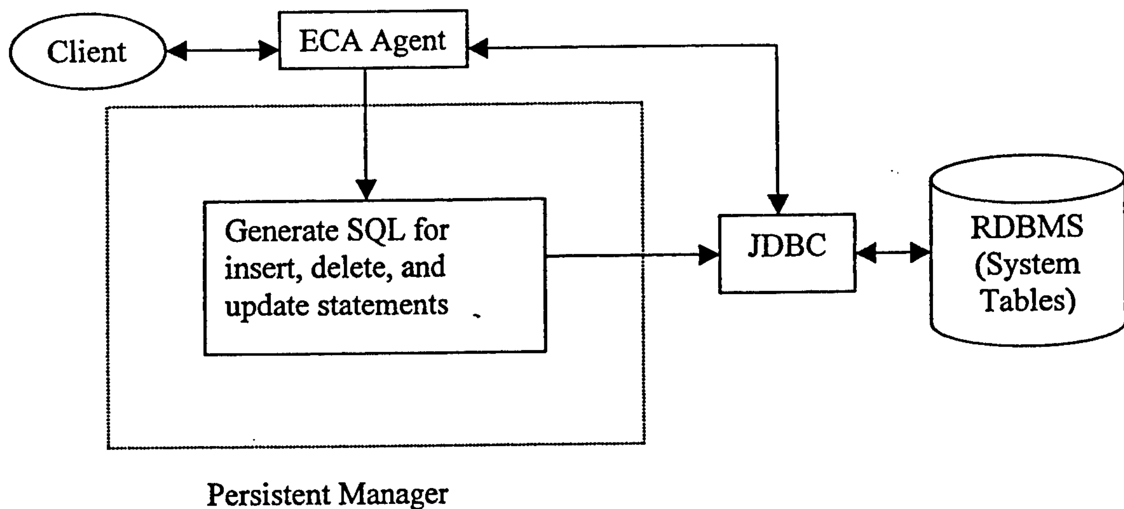



Figure 6 **The Architecture of Persistent Manager**

When a user connects the ECA Agent, a thread serving one client is created. The persistent Manager runs inside the thread. It generates ECA rules for the triggers stored in the system tables of the RDBMS. The generated ECA rules for triggers are sent to the RDBMS through JDBC and are used as trigger action parts when a related event occurs.

### 4.4 Action Handling by the Java LED

The Actions executed by the Java LED can be the execution of SQL commands in the Oracle Server, the update of meta-data, and the invocation of stored procedures when a composite event is detected. Every ECA rule has its event definition as well as rule (only action in case of a trigger) definition. Every event action is processed according to its rule definition.

## 5 IMPLEMENTATION OF PRIMITIVE EVENTS

This section discusses the implementation details of primitive event detection. Although relational DBMSs provide a trigger mechanism, it is very limited in functionality. Full-fledged active capability is useful for many applications. The ECA Agent uses the trigger mechanism of underlying RDBMS to support active capability and extends it significantly to turn the RDBMS into a fully active RDBMS.

### 5.1 ECA Agent

Triggers supported by the RDBMS have a number of limitations:

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

The ECA Agent overcomes the above-mentioned limitations by providing a complete active database capability. The functionality extended by the ECA Agent includes the specification of any number of triggers on the same event. Suppose, we have already defined one primitive event as

*Create trigger t_addStk on Stock after insert event addStk.........*

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

*Create trigger t_addStk1 event addStk.........*

### 5.2 Syntax of Primitive Events

In this section, we describe how we define a primitive event using SQL statement. In defining a primitive event, the only difference between Oracle SQL statement and this ECA Agent statement for primitive event is the event declaration using the keyword, *event*. That is, we extend the trigger definition to provide ECA active capability. Figure 7 shows Oracle SQL statement. Figure 8 shows syntax of primitive event definition of the ECA Agent using Oracle SQL grammar.

```
Create trigger t_addStk after insert on stock
For each row
Begin
dbms_output.put_line('trigger t_addStk occurs on primitive event, addStk');
End;
```

Figure 7 **Oracle SQL Statement**

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

Figure 8 **Primitive Event Definition using Oracle SQL Statement**

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

5.3 Processing of a Primitive Event

The processing of a primitive event involves parsing and generation of the primitive event node in the event graph. Figure 9 shows the processing steps of the primitive event. The processing of the primitive event includes the following steps:

1) Syntax check: The command is checked. If a syntax error is found, an error message is returned to the client.
2) Duplication check on trigger: The duplication of trigger name is not allowed in the RDBMS. Trigger name of an event is checked. If trigger name is duplicated, an error message is returned to client.
3) Persistent code generation: SQL code is generated and persisted.
4) Java LED code generation: If there is no error in defining a primitive event, code is generated to create a primitive event node in the Java LED. We create the primitive event, *addStk* using the Java LED as follows:
   *a)* ECAAgent myAgent = ECAAgent.initializeECAAgent( );
   b) EventHandle addStk = myAgent.createPrimitiveEvent("addStk", "Led", EventModifier.BEGIN, "void addStk( )", DetectionMode.SYNCHRONOUS);
   c) The above will be inserted into the file, *ykimaddStk.java* in our implementation.
5) File generation: We create the new file to get an event handle for the primitive event. ( For example, in our implementation, *ykimaddStk.java* is created. This file is compiled automatically and implicitly using Java Runtime class). We use dynamic

method call using Java Reflection (In our implementation, the called method is *call_addStk* inside the file, *ykimaddStk.java*).

6) Create related tables: The tables, Stock_inserted and Stock_deleted are created. The Stock_inserted table is used to store the inserted tuples of a table. It is very similar to table Stock except that it has VNO column. VNO is used to store the occurrence number of the insert operation. For example, the event, addStk is detected whenever a tuple is inserted into Stock table. VNO is increased by 1 for each insertion. (The value of VNO is used to collect parameter for composite event). Table 12 shows Stock_inserted and Stock_deleted tables.

Table 12 Stock_inserted and Stock_deleted

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

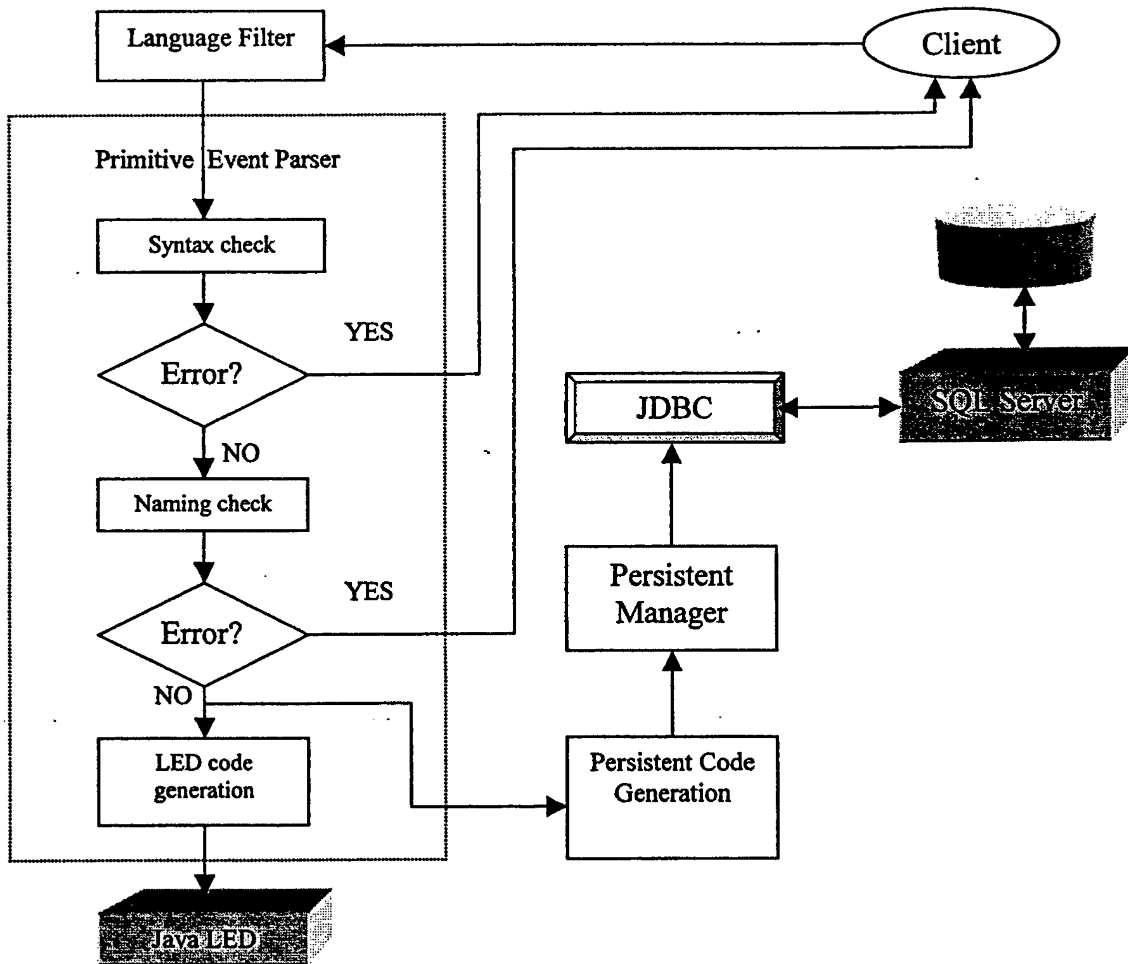



Figure 9 **Parsing and Generating of Primitive Event**

7) Trigger creation for primitive event: After an operation is executed on 'Stock', the trigger will be fired for the operation. The trigger for the operation includes the following steps:

- Set and get event occurrence number:

  *Update SysPrimitiveEvent set VNO=VNO+1 where eventName = 'addStk';*
  *Delete from Version;*
  *Insert into Version select VNO from SysPrimitiveEvent where eventName = 'addStk';*

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

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

- Table, eventContext is made when we define a composite event.
- Insert an inserted tuple into table, Stock_inserted (in the table, Stock_deleted, *new* is replaced with *old*).
- Primitive Event Notification:

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

- When a primitive event occurs (in this case, insert operation), event name and table name and version number are inserted into notify table.
- Then, all the contents of notify table on the primitive event are inserted to the Java LED using primEvent method of Led.java when a primitive event occurs. The raiseBeginEvent method inside primEvent raises the primitive event.
- Notify table check is shown in Figure 10.

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

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

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

     if(test1 == null) test1 = new Led();
       test1.PrimEvent(en,tn,vno)    // to raise primitive event
  }
```

Figure 10 **Notification Check**

5.4 Syntax for specifying multiple triggers on the same event

Once an event is defined, the user can define additional triggers on the event. Figure11 illustrates the syntax of defining trigger for a previously defined event.

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

Figure 11 **Syntax of Defining a Trigger on Existing Event**

In this example, we do not have to specify *after [operation]* and *on [ table-name]*. In our implementation, if a user creates another trigger on an existing event, Repeated Primitive Event Parser (the method, RepeatPrimitive(.........) in ECAparser class) is called. The trigger is transformed into original Oracle SQL syntax as shown in Figure 12. This functionality is used to add an additional trigger action to an existing event.

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

Figure 12 **The Changed Event Definition on Existing Event**

After this trigger definition, only SysEcaTrigger table is updated adding the tuple (as shown in shadow in Table 13):

Table 13 SysEcaTrigger after the existing event (addStk) definition

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

### 5.5 Syntax for Dropping a Trigger

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

*Drop trigger trigger_name;*

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

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

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

## 6 CONCLUSIONS AND FUTURE WORK

In this report, we presented the details of design, architecture, and implementation of the ECA Agent system. The contributions of the project are as follows:

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

### 6.1 Future Work

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

will enhance the performance of the overall system. We also plan on extending this to Sybase and DB2 RDBMSs.

## 7 REFERENCES


BER91 Berndtsson, M., "ACOOD: an Approach to an Active Object Oriented DBMS," *Master's report*, University of Skovde, September 1991.

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

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

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

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

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

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

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

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

DAS99 Dasari, R., "Events and Rules for Java: Design and Implementation of a Seamless Approach," *Master's report*, University of Florida, 1999.

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

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

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

LI98 Li, L., "An Agent-Based Approach to Extending the Native Active Capability of Relational Database Systems," *Master's report*, University of Florida, Gainesville, 1998.

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

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

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

***MISSION***
***OF***
***AFRL/INFORMATION DIRECTORATE (IF)***

*The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.*