AN AGENT-BASED APPROACH TO EXTENDING THE NATIVE ACTIVE CAPABILITY OF RELATIONAL DATABASE SYSTEMS

By

LIJUAN LI

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1998

To my parents

ACKNOWLEDGMENTS

Firstly, thanks are due to my advisor, Dr. Sharma Chakravarthy for his excellent advice, great guidance and support in this research. His tireless patience, knowledge, and intelligence have been invaluable. I am grateful to Dr. Stanley Su and Dr. Eric Hanson, for graciously agreeing to serve on my committee, for teaching wonderful classes which have greatly helped me in the advancement of my understanding of databases, and for taking time from their busy schedules to read and comment on my thesis. I have been very fortunate to have such excellent committee members; it has been a pleasure to interact with them all.

I would like to thank Sharon Grant and Hyoungjin Kim for maintaining a well administered research environment and being so helpful in times of need. Special thanks to Shiby Thomas, Hyoungjin Kim, Chris Carnes, Federico Zoufaly, Roger LE and Shuan Yang, for the many fruitful discussions and invaluable advice on many aspects of my research work. Also I would like to thank all my friends for their constant support and encouragement.

This work was supported in part by the Office of Naval Research and the SPAWAR System Center–San Diego, by the Rome Laboratory, DARPA, and the NSF grant IRI-9528390.

I would like to take this opportunity to thank my family for their endless love. Without their support, this work would not have been possible.

Finally, and supremely, I would sincerely thank the God for His love and for His sustaining hand upon me in every area of my life. I could have accomplished absolutely nothing apart from His will or without His mercy and grace which were purchased for me by the Lord Jesus Christ. Thank You for Your lighting up my life. Thank You for whatever You do with me. Thank You for the kindness You have shown to Your very undeserving servant. May honor be given to Whom honor is due. If there is anything of worth, or insight, or help in this thesis,

SOLI DEO GLORIA. (To God Alone be the Glory)

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

AN AGENT-BASED APPROACH TO EXTENDING THE NATIVE ACTIVE CAPABILITY OF RELATIONAL DATABASE SYSTEMS

By

Lijuan Li

May 1998

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

Active Database systems have been proposed as a data management paradigm to satisfy the needs of many applications that require a timely response to situations. The promises of Active Database System are significant. ECA rules are used to capture the active capability. While a number of research prototypes of active database systems have been built, ECA rule capability in Relational DBMSs is very limited. This thesis addresses the problem of turning a traditional database management system into a true active database system without changing the semantics of the existing system. The advantages of this approach are obvious. First, transparency is guaranteed. That means we can add the active database capability without changing the client programs. Second, all of relational DBMS's underlying functionality is retained. Third, ECA rules can be made persistent by using the native database functionality.

Active database semantics can be supported on an existing Sybase SQL Server by adding a mediator, termed ECA Agent, between the SQL Server and the 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 persistent in Sybase system. The Gateway Open Server in ECA Agent is designed to connect SQL Server by using Sybase connectivity products. The architecture of the ECA Agent and implementation details are presented in this thesis. Also alternative approaches are discussed in detail, and the features and limitations are identified.

CHAPTER1
INTRODUCTION

A traditional database is a passive repository of data where the DBMS only execute the explicit transactions and queries asked by a user or an application program. The DBMS uses a query-driven mechanism. This traditional view of databases as information repositories, which are used for storing and retrieving required information, works for many applications. However, the need for having a database system capable of reacting to specific situations without user or application intervention has been recognized in several newer applications. Frequently mentioned examples of such applications are network management, computer-integrated manufacturing (CIM), commodity trading, air-traffic control, plant and reactor control, tracking, monitoring of toxic emissions, workflow and process control, etc. Active database systems have been proposed as a new data management paradigm to satisfy these kinds of needs so that the system can monitor the state of the database for particular events, and trigger appropriate and timely responses when events occur. This can be done by defining event-condition-action (ECA) rules which are stored as part of the database [DAY94]. Active database systems, by the way of rule definition, event detection and action execution, are not only driven explicitly by a user or an application, but they are also able to recognize specific situations (in the database and external to the database) and react to them.

So far, a number of research prototypes of active database systems have been developed, such as HiPAC [CHA89], Ariel [HAN92], Sentinel [CHA93], Starburst [WID91], Exact [DPG91], Postgres [STO91], PEARD [JAH96], SAMOS [GAT92] etc. Most of them are developed from scratch or integrated directly into the kernel of the DBMS. The integrated approach provides the following advantages: [CHA89]

- Do not require any changes to existing applications.
- DBMS is responsible for optimizing ECA rules.
- DBMS functionality is extended.
- Modularity/maintenance of applications is better and maintenance is easier.

However, the implementation of an integrated approach requires access to the internals of a DBMS into which the active capability is being integrated. This require access to source code, makes the cost of integrated approach very high and requires a long integration time as well. Hence, most integrated systems are research prototypes.

There are alternative approaches to support active capability, such as Embedding Situation Check in application code and Polling. However these suffer from many limitations. For example, the Embedded Situation Check approach requires extra code in all applications. Since modularity is compromised, the management and maintenance of applications is difficult. Also, constraints and business rules are not clearly separated from the application [CHA89]. The polling for a relational DBMS is discussed in Chap 2.

To the best of our knowledge, there is no commercial DBMS that supports full active capability, although the promises of active database technology is well-understood and is considered significant. Currently, relational DBMS are widely used. Most users are familiar with RDBMS, and the advantages of relational DBMS are well known. Rule capability is provided in many commercial systems, but it is not sufficient as it only provides basic triggering capabilities.

The challenge is to turn a commercial database management system into a true active database system without making any changes to the underlying system. This thesis introduces an approach which adds a mediator to the Sybase SQL Server DBMS to provide ECA functionality in the SQL Server.

Although there are some advantages to integrating the active capability to the DBMS kernel, the use of a mediated architecture outside of the SQL Server provides many benefits, including:

- Transparency: The clients do not feel the mediator.
- System functionality: None of the existing DBMS's functionality would be lost.
- Extensiblility: A more distributed architecture can be designed.
- Scalability: The architecture is scaleable.
- Portable: Once the mediated approach has been developed, it may be ported to other Relational DBMSs.

The contributions of this thesis are as follows:

1. Present a mediated approach that significantly extends Sybase ECA functionality:
    1) A client can create multiple triggers on the same event.
    2) A client can create composite events and triggers on them.
    3) Reuse of previously defined events (both primitive & composite).
    4) Drop triggers associated with primitive or composite events.

5) Once events are created, they become persistent in the database system.

6) All primitive events and composite events can be detected, and actions are invoked within SQL Server.

Also, we discuss alternative approaches and show why mediated approach is a good solution.

2. Well defined mediated approach (ECA Agent), and present the architecture of ECA Agent.
3. Discuss how ECA Agent is implemented.
4. Provide an API for SQL Server, and show how this architecture is also suitable for other SQL Servers.

The remainder of this thesis is organized as follows. The related work is shown in Chapter 2. In Chapter 3 we discuss an alternative method to the layered approach. The architecture of ECA Agent is presented in Chapter 4. OpenServer is introduced in Chapter 5. The implementation details of Primitive and Composite Event are shown in Chapter 6 and Chapter 7. We then draw conclusions and discuss the future work in Chapter 8.

CHAPTER 2
RELATED WORK

In this chapter, several research efforts on implementing Active Database Management Systems, including the mediated approach are discussed.

2.1 Sentinel

Sentinel is an Object_Oriented Active Database System. It uses Open OODB as its platform, and integrates ECA rules capability into the kernel of Open OODB. The integration enhances Open OODB from a passive OODB to an active one.

2.1.1 Functional modules of Sentinel

The functional modules of Sentinel are integrated into the kernel of Open OODB. The integrated modules are:

- Local Event Detector (LED): This includes primitive event detection and composite event detection within an application or address space. A method can be specified as a primitive event, and the occurrences of the primitive events are notified to the local event detector when the method is invoked. Composite events defined within an application are detected by using a sequence of primitive events detected according to the operator semantics as well as the specified parameter context of the composite event [LEE96].
- Global event detector (GED): An inter-application event is detected by the

GED. The GED communicates with the LED through RPC and socket-based communication to detect global events [LIA96].

- Nested transactions: The transaction manager in Open OODB has been extended to nested transactions for concurrent as well as prioritized execution of rules in the client address space [SAY96] [BAD93].
- ECA rule editor: This is an external rule support in Sentinel. The editor provides users with a friendly environment to manipulate the rules without changing, recompiling, or relining any source code of their applications. Java is used to implement the interface [CHU98].
- Snoop preprocessor: The Snoop preprocessor converts the high-level user specification of ECA rules specified in the Snoop language into appropriate code for event detection, parameter computation, and rule execution [LEE96].

2.1.2 Event Operators

So far, Sentinel supports ten event operators. Composite event expressions are constructed by using any of these operators using the Snoop syntax. Following is a summary of the operators:

1. AND: Conjunction of two events, namely E1 and E2, denoted by **E1 ^ E2**, occurs when both events occur. The order of occurrence of E1 and E2 is irrelevant.
2. OR: Disjunction of two events, namely E1 and E2, denoted by **E1** Ñ **E2**, occurs when either E1 or E2 occurs.
3. SEQUENCE: Sequence of two events, namely E1 and E2, denoted by **E1>>E2**, occurs when E2 occurs only after the occurrence of E1.

4. NOT: NOT operator detects non-occurrence of an event, namely E2, in the closed interval formed by two events, namely E1 and E3, denoted by ∅ **E2[E1,E3]**.
5. ANY: The event , denoted by ANY(m,E1,E2,…..,En) where m <= n, occurs when m events out of n distinct events specified occur, ignoring the relative order of their occurrences.
6. A (Aperiodic): A aperiodic event, denoted by **A(E1,E2,E3)**, is detected for every occurrence of E2 during the half-open interval formed by E1 and E3.
7. A*: It is a cumulative variant of the A operator. A aperiodic-star event, denoted by **A*(E1,E2,E3)**, is detected when E3 occurs provided E1 has already occurred. The occurrences of E2 are accumulated during the half-open interval formed by E1 and E3.
8. P (Periodic): A periodic event, denoted by **P(E1,E2,E3)** where E2 is a relative temporal event, is detected for every time period specified by E2 during the half-open interval (E1,E3].
9. It is a cumulative variant of P operator. A periodic-star event, denoted by **P*(E1,E2,E3)** where E2 is a relative temporal event, is detected only once when E3 occurs provided the E1 has already occurred. The time specified in E2 is accumulated whenever E2 occurs.
10. PLUS: It is a variant of SEQUENCE operator. A PLUS event, denoted by **E1 + [T]** where [T] is a relative temporal event, is detected once a time interval [T] elapsed after the occurrence of E2.

2.1.3 Parameter Context

The proposed parameter contexts in Sentinel are *recent, continuous*, *cumulative* and *chronicle* [CHA94]. These contexts are precisely defined using the notion of initiator and terminator events. An initiator of a composite event is a constituent event that can start the detection of the composite event whereas a terminator is a constituent event that can detect the occurrence of the composite event. Following is a short explanation of each parameter context:

- **Recent**: Only the most recent occurrence of the initiator for any event that has started the detection of that event is used. When an event occurs, the event is detected and all the occurrences of events that cannot be the initiators of that event in the future are deleted.
- **Chronicle**: In this context, for an event occurrence, the initiator, terminator pair is unique. The oldest initiator is paired with the oldest terminator for each event, i.e. in chronological order of occurrence. The constituent events of an event E cannot occur in any other detection of the occurrence of the composite event.
- **Continuous**: This context is similar to chronicle, with the subtle difference that in the latter, pairing of the initiator is with respect to the terminator whereas in this context multiple initiators are paired with a single terminator of that event.
- **Cumulative**: In this context, for each constituent event, all occurrences of the event are accumulated until the composite event is detected. In other words, the parameters of a composite event include the parameters of all the

occurrences of each constituent event. All the occurrences of each constituent event are flushed whenever its associated composite event is detected.

2.1.4 Local Event Detector (LED)

We briefly describe the LED here as the LED developed for Sentinel is also used to implement the ECA Agent. Events are detected by the LED in Sentinel. The primitive and composite events are declared in an application and reflected to an event graph. Each event is represented as an event node in the graph, and the event nodes are connected by their subscription relationships. An internal node of the event graph represents a composite event, and a leaf node represents a primitive event. The root node of event tree represents a user-defined composite event. Event trees in an application are merged to form an event graph for detecting a set of composite events.

The methods that can generate primitive events are modified by the wrapper class methods using the sentry feature of the Open OODB system while preprocessing the application program. While preprocessing the application program, the Snoop preprocessor of Sentinel adds code for parameter collection and notification to the event detector.

The concept of event notification and list of subscribers was first introduced in Sentinel to do the event detection [CHK94].

Whenever a primitive event occurs, an unique instance number is given for its occurrence (t_occ), its occurrence time is set to the parameter list, and further notifications are sent to its subscribers. The node of the event tree maintains the occurrences of its constituent events with their parameter lists which are stored separately for each context as part of the node. When it is notified by one of its constituents, the node checks the status of its constituents' occurrence. If the composite event occurs by the last notification, it is detected, and further notifications are sent to its subscribers.

2.2 ACOOD

ACOOD is a fully implemented research prototype based on the ONTOS™ object-oriented DBMS which has C++ as its base language. It was originally developed and implemented by Mikael Berndtsson and his group at the University of Skovde [Ber91], and has been addressed in several reports including [BER92] and [BER94].

A layered approach is used for ACOOD to develop an active DBMS. This means that ACOOD is layered upon ONTOS™ and can use all functionality of this DBMS. ACOOD extends the DBMS with the functionality of active features. An application can use either the ONTOS™ client interface for traditional database operations or the ACOOD application interface when active features are needed (see figure 2.1).

ACOOD adopts the concept of ECA rules and represents events and rules as first class objects. By representing them as first class objects, ACOOD provides a framework for supporting runtime management of events and rules.

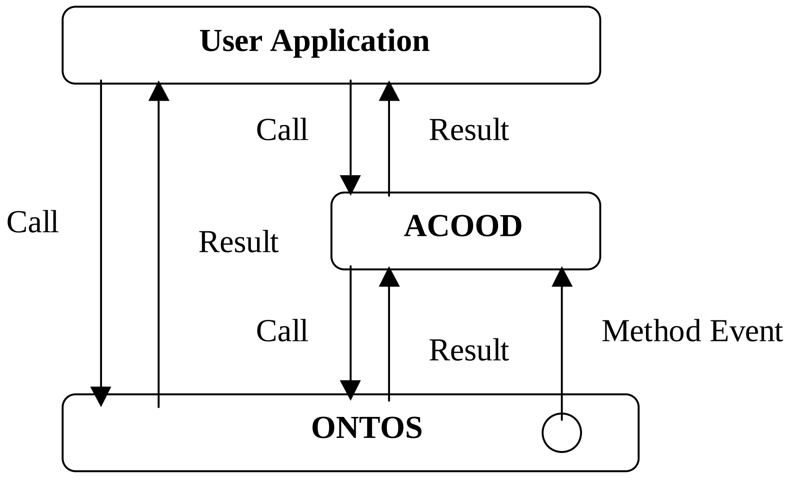


Figure 2.1: ACOOD Architecture and Interactions

2.3 Polling Subsystem

In [VAN96] Polling Subsystem is discussed and implemented to support detached rule coupling mode in Sybase. Figure 2.2 shows the architecture of this approach [VAN96].

This is a layered approach. Polling Subsystem actually is a periodic polling application. A polling application is a client of the database where the data of interest resides. To implement detached semantics, the polling application examines the current state of the system to determine if any events need to be triggered.

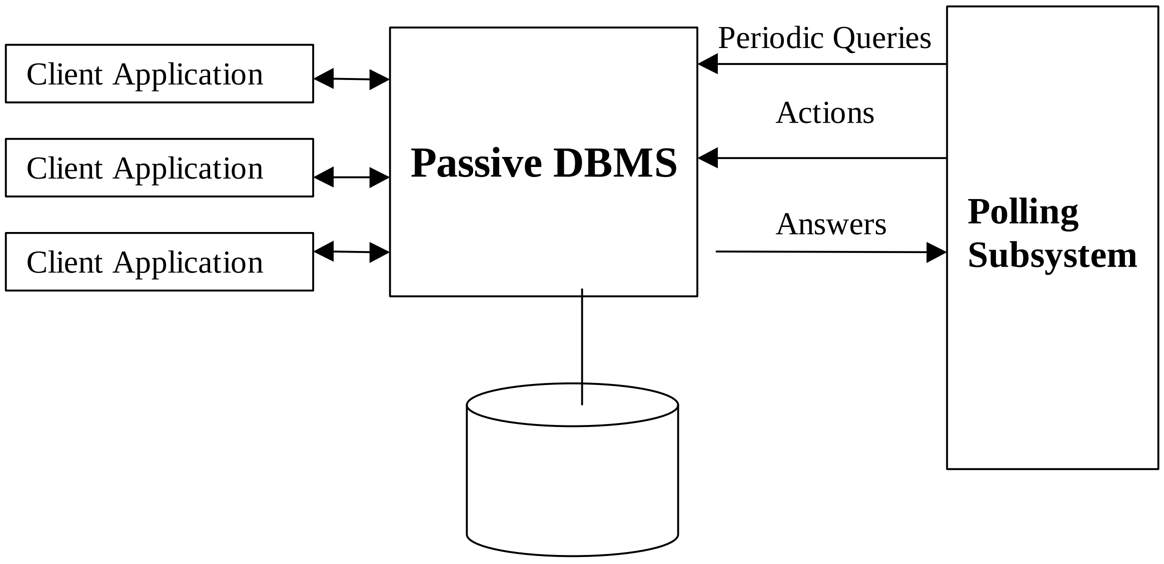


Figure 2.2: The Polling Architecture

The advantages of polling subsystem are:

1) Polling subsystem can be added/changed/removed without affecting the other parts of system.
2) This architecture is simple and easy to implement.

3) It is widely-used.

However, there are many limitations to polling subsystem which limit its practical usefulness:

1) If polling is too slow, it may miss taking appropriate action.
2) The order of events can not be determined.
3) Client applications may be delayed waiting for the polling to complete.
4) The architecture is not scaleable.
5) It is not suitable for time-constrained applications.

Therefore, a polling subsystem may be suitable for a problem that does not require every event to be captured and has a small data size.

2.4 The GATEWAY Approach

As we know, a polling subsystem is an appropriate solution when the problem does not require every event to be captured and the size of the data is small. But many problems require both a large data size and every event to be captured (although not used depending upon the parameter context used) once it occurs. Another approach, Gateway, is implemented in [VAN96]. The architecture of this approach is showed in Figure 2.3. The Gateway architecture is a layered API approach. This layer is truly transparent to the client. Open Server is used to implement Gateway (ECA-Server in the Figure) and supports both detached and deferred coupling mode in Sybase.

ECA-Server accepts connection requirement from a client and passes all client information to the SQL Server. SQL Server authenticates the client and creates the connection. After the connection has been established, the gateway takes control of the client process. SQL language and procedure requests from the client are preprocessed by the ECA Server. Coupling mode of events are processed in ECA Server, i.e. before any transaction is committed, the ECA Server can interrupt the transaction and perform inquires on the SQL Server.

This architecture has many advantages:

1) Both immediate and deferred trigger semantics can be implemented.
2) Events are generated in the correct order.
3) Events are not lost.
4) Parallel execution of ECA Server and SQL Server is possible.
5) Good performance.

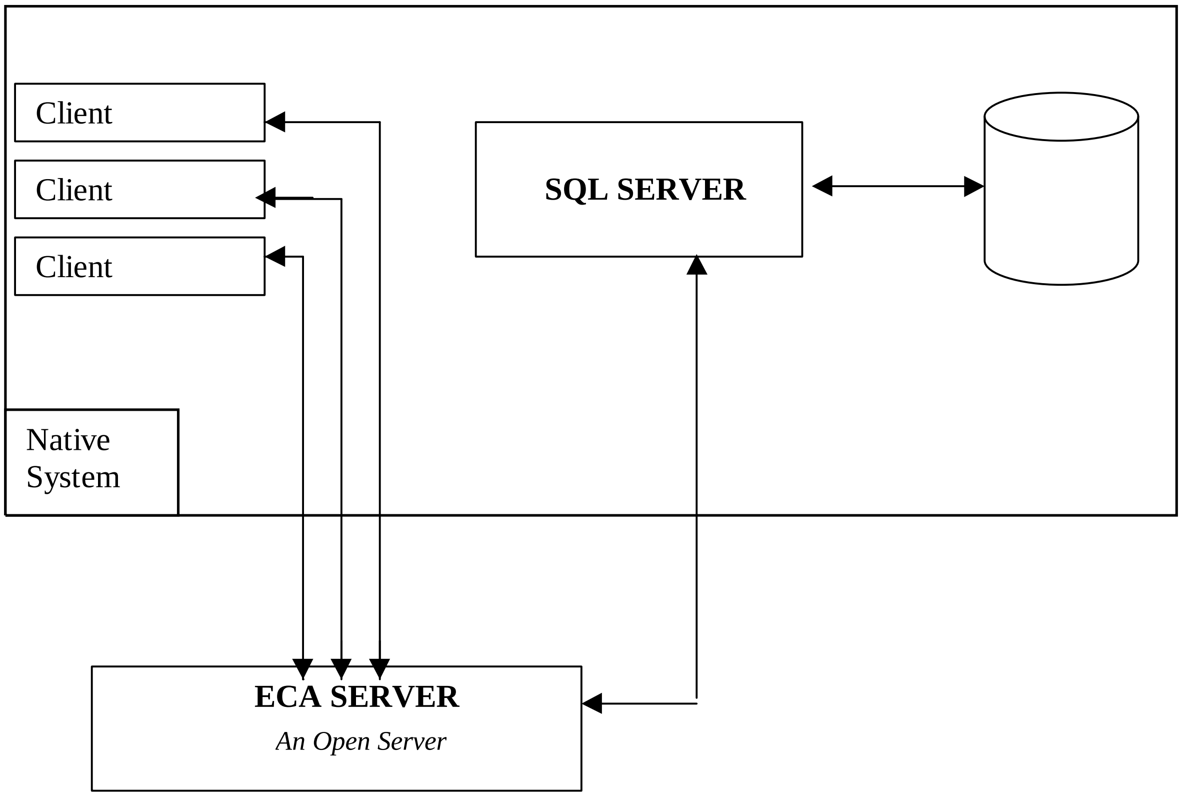


Figure 2.3: Gateway Architecture

CHAPTER 3
WRAPPER APPROACH WITH EMBEDDED SQL/C

As we have seen in the earlier discussion, Sentinel is an Object-Oriented Active Database System. Though it is implemented by integrating ECA functionality into the kernel of Open OODB, the LED (Local Event Detector) is independent of the OpenOODB. Therefore, we are able to reuse the LED developed for Sentinal and create an agent between LED and the SQL Server.

In this chapter, we discuss a solution that uses a wrapper method along with the mediated LED module to add ECA capability to the SQL Server.

3.1 Architecture

Because a client can place Transact-SQL statements in application programs that are written in C (Embedded SQL/C), and because a client can write methods in Embedded SQL/C, we can think of each method as an object, and define ECA rules on these methods. The ECA Preprocessor is implemented to wrap methods where events are defined. Figure 3.1 shows the architecture of the system.

Life cycle of ECA rules in this architecture:

1) Defines ECA rules in the Embedded SQL/C program.
2) Wrap the methods in the program and change all ECA rules into code that the LED can recognize by using ECA Preprocessor.
3) Translate the Embedded SQL statements into Client-Library function calls through the Embedded SQL Precompiler.

4) Compile and link the code generated in the last three steps with the LED Library and Open Client-Library to generate the executable code.
5) Run the executable code and return the result to the client.
6) Once the client stops running the code, the ECA rules also disappear.
7) The client can go to step 5 to start again, or the client can change ECA rules and start from step 1.

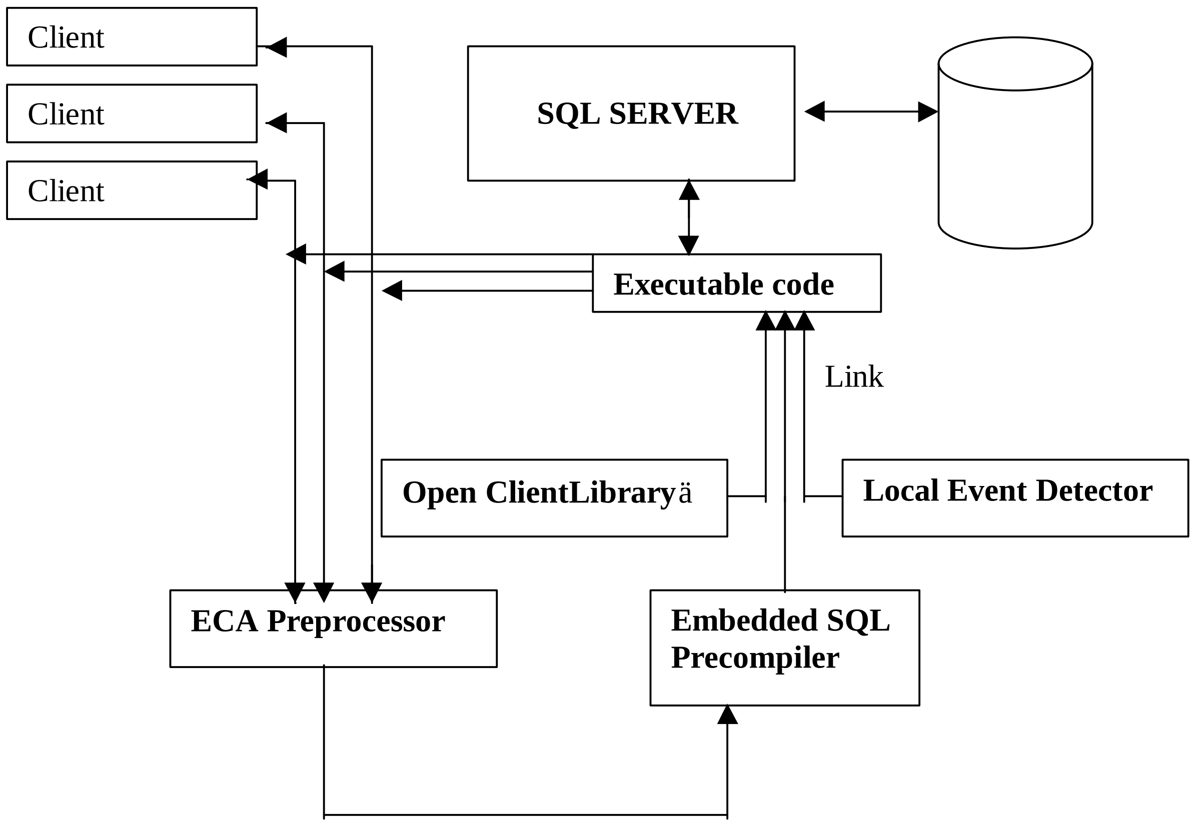


Figure 3.1: Architecture of Wrapper Approach of Sybase LED

3.2 SNOOP

SNOOP [CHM94] is an event specification language in Sentinel. It gives precise semantics of primitive events and several event operators proposed to express composite events. It supports temporal, periodic, explicit and composite events. Since it is well defined, we can take SNOOP as our event specification language. SNOOP BNF is given below [CHA92] [LIA96] [LEE95]

```
Event_exp ::= E1
        E1 ::= E1 OR E2 | E2
        E2 ::= E2 AND E3 | E3
        E3 ::= E3 SEQ E4 | E4
        E4 ::= NOT (E1, E1, E1)
               | A (E1, E1, E1)
               | A* (E1, E1, E1)
               | P (E1, [time string], E1)
               | P (E1, [time string]: parameter, E1)
               | P* (E1, [time string], E1)
               | P* (E1, [time string]: parameter, E1)
               | [time string]
               | E1 PLUS [time string]
               | (E1)
               | event_name
event_name::= name
                       | Eventname: Objectname
```

| *Eventname*:: *AppId*

AppId ::= *Sitename__Appname*

| Identifier

Name ::= Identifier

Eventname ::= Identifier

Objectname ::= Identifier

Also, we can find event and rule specification in [LIA96]. Syntax of the event and rule specification in SNOOP is as follows [VK94] [LEE95] [LIA96]:

Event_spec ::= **event** *event_modifier* method_signature

| **event** event_name = *event_exp*

| **begin** (event_name)

| **end** (event_name)

| **begin** (event_name) && **end** (event_name)

| **end** (event_name) && **begin** (event_name)

Event_modifier ::= event_name

Rule_spec ::= **rule** rule_name (event_name,

condition_function, action_function)

[, [*parameter_context*], [*coupling_mode*]

, [*priority*], [*rule_trigger_mode*]])

parameter_context ::= RECENT | CHRONICLE | CONTINUOUS |

CUMULATIVE

Coupling_mode ::= IMMEDIATE | DEFERRED | DETACHED

Priority ::= positive integer

Rule_trigger_mode ::= NOW | PREVIOUS

3.3 ECA Preprocessor

There is a SNOOP preprocessor (named SPP) is part of the Sentinel system. It is integrated with the OpenOODB preprocessor ppCC for wrapping methods of the reactive class and creating several files (such as the signature and static files for use by other components of Sentinel). We can separate the SPP from the OpenOODB, and add functions to support Sybase Embedded SQL/C. We name this new preprocessor as SQL-ECA Preprocessor. The SQL-ECA Preprocessor provides following functionality:

1) Embedded SQL/C only works for C language, and LED is designed based on C++. Initially, all user code is written in SNOOP and Embedded SQL/C. There is no OO concept in the user's code. To make everything consistent, the ECA Preprocessor puts all methods into a class _SYBASE.
2) It processes event and rule specifications defined by the user in SNOOP, and inserts translated C++ code into the user program.
3) It wraps all the methods in _SYBASE class.
4) It creates several files so that it can be connected to the rule editor and Global Event Detector.

3.4 Implementation of Sybase LED

Some of the modules shown in Figure 3.1, have already been implemented. LED implemented as part of Sentinel, has been used without modification. Open ClientLibrary™ and Embedded SQL/C precompiler are implemented in the Sybase system. The only part of the architecture requiring implementation is the ECA Preprocessor. Since ECA Preprocessor is based on SPP, changes can be made to SPP. There are four changes needed:

1) On preprocessing, all event definitions (including primitive and composite) are translated into LED function calls. The difference between the SPP and the ECA Preprocessor is that ECA Preprocessor translates all events into events on a single class _Sybase (class level event).

   For example: event e1 is defined in SNOOP and Embedded SQL/C as follows.

   event end (EventSellBook) void sell_book();

   It will be translated into:

   PRIMITIVE *BOOK_EventSellBook = new PRIMITIVE("Sybase_EventSellBook", "Sybase", "end","void sell_book()");

   But for SPP, the event definition should be :

   Class BOOK

   {…

   event end (EventSellBook) void sell_book();

   }

   it will be translated into:

PRIMITIVE *BOOK_EventSellBook = new PRIMITIVE("BOOK_EventSellBook", "BOOK", "end","void sell_book()");

2) SPP calls ppCC (Open OODB preprocessor) to wrap the method, ECA Preprocessor is separated from ppCC, and writes a wrapper module instead. This wrapper module changes the method into another name, and wraps the method into a new method whose name is the same as the original method. It also adds a notificaiton statement to the LED.

For example, method sell_book is defined as Figure 3.2:

```
void sell_book() {
EXEC SQL BEGIN DECLARE SECTION;
        int count;
        int quan;
        int key;
EXEC SQL END DECLARE SECTION;
exec sql whenever sqlerror call err_handle();
exec sql whenever not found goto not_found;
printf("please input call_number: "); scanf("%d",&key);
printf("How many books sold?"); scanf("%d",&count);
EXEC SQL CONNECT :username IDENTIFIED BY
:password;
  EXEC SQL select quan into :quan  from BOOKS_DB
  where call_no = :key;
 if ((quan-count)>=0) {
        quan=quan-count;
EXEC SQL update BOOKS_DB set quan = :quan
        where call_no = :key;
        printf("%d books sold!",count);
        }
        else {
                EXEC SQL update BOOKS_DB set quan = 0
                where call_no = :key;
                printf("only sold %d books!no more books !",quan);
                }
  EXEC SQL COMMIT;
not_found:printf("record not found!\n");
  EXEC SQL DISCONNECT DEFAULT;
}
```

Figure 3.2: Definition of void sell_book()

After preprocessing, the method is changed into Figure 3.3

```
void _wrapper__sell_book() {
EXEC SQL BEGIN DECLARE SECTION;
        int count;
        int quan;
        int key;
EXEC SQL END DECLARE SECTION;
exec sql whenever sqlerror call err_handle();
exec sql whenever not found goto not_found;
printf("please input call_number: "); scanf("%d",&key);
printf("How many books sold?"); scanf("%d",&count);
EXEC SQL CONNECT :username IDENTIFIED BY :password;
EXEC SQL select quan into :quan  from BOOKS_DB
  where call_no = :key;
 if ((quan-count)>=0) {
        quan=quan-count;
EXEC SQL update BOOKS_DB set quan = :quan
        where call_no = :key;
        printf("%d books sold!",count);
        }
        else {
                EXEC SQL update BOOKS_DB set quan = 0
                where call_no = :key;
                printf("only sold %d books!no more books !",quan);
                }
  EXEC SQL COMMIT;
not_found:printf("record not found!\n");
  EXEC SQL DISCONNECT DEFAULT;
}

void sell_book()
{
if (is_this_subscribed( "Sybase",  "void sell_book() ","begin"))
      {
        PARA_LIST *begin_sell_book_list = new PARA_LIST();
        Notify(this, "Sybase","void sell_book() ", "begin", begin_sell_book_list);
        }
_wrapper__sell_book();
if (is_this_subscribed( "Sybase",  "void sell_book() ","end"))
      {
        PARA_LIST *end_sell_book_list = new PARA_LIST();
        Notify(this, "Sybase","void sell_book() ", "end", end_sell_book_list);
       }
}
```

Figure 3.3: Method sell_book() after wrapper

3.5 Integration with Sentinel

As the code base for OO-LED and the relational-LED is same, it is possible to use this in conjunction with the Global Event Detector (GED) as shown in Figure 3.4. Once the GED server is started, event defined in Sybase can send event requests to the GED and the GED can receive event notifications from the Sybase LED. Figure 3.4 shows the relationship between LED and relational as well as OO applications.

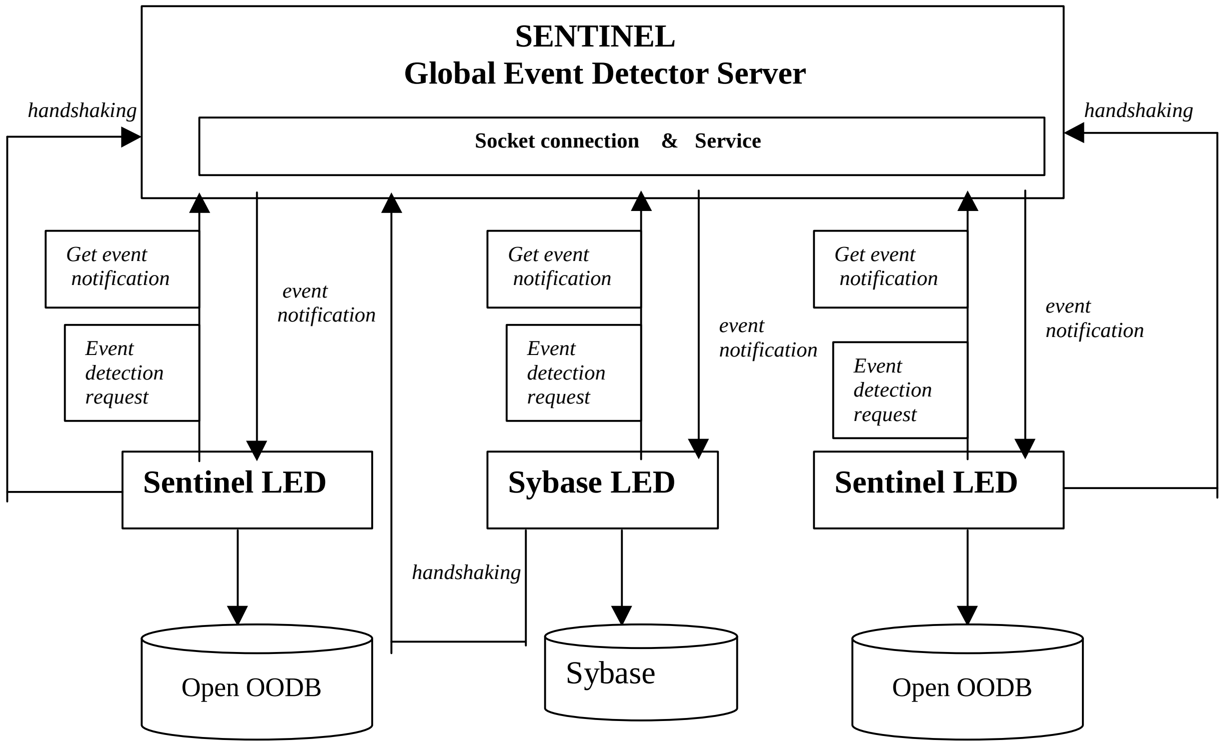


Figure 3.4: Integration with Sentinel

3.6 Features and Limitations

Using wrappers with Embedded SQL/C to implement a relational LED for Sybase offers the following benefits:

1) It is easy to implement.

   It is implemented from the existing code base used in Sentinel.

2) The architecture is simple and can be along with the other database systems.

3) It can be integrated into Sentinel so that events can be detected and rules executed in a heterogeneous system.

4) Because of the mediated approach, the Sybase LED can be added without any changes to the underlying system.

However, the following limitations needed to be recognized:

1) Composite events and rules are only existed at runtime.

   All ECA rules stay in memory. Once a user stops running the program, all ECA rules immediately disappear. Only when user restarts the program, can ECA rules be recreated.

2) ECA rules can only be defined statically.

   All ECA rules are defined before a program is compiled. This eliminates the possibility of runtime declaration of ECA rules. If a user wants to add, delete or update an ECA rule, the application program must be stopped, the source code modified, and the entire program recompiled (restart a new life cycle for all ECA rules).

3.7 Conclusion

A Wrapper with Embedded SQL/C may be an appropriate solution when an application is simple and small, with statically defined ECA rules. However, when applications are complicated, large, and need runtime declaration of ECA rules, this needs to be reexamined to accommodate dynamic ECA rule capability.

CHAPTER 4
DESIGN ISSUES OF THE SYBASE ECA AGENT

The previous approach suffered from several limitations although it its implementation is straightforward. It is mostly suitable for very simple systems. Even though we can make some changes to improve it, we cannot ensure the atomicity of an event as it depends on users. Also we cannot change the preprocessing mechanism. Therefore, an alternative solution which overcomes the above problems is necessary.

The ECA Agent proposed in this chapter is an alternative solution that overcomes the problems of the previous (wrapper) approach. We discuss implementation detail in the next few chapters. The ECA Agent is a mediated approach using Open Server to provide the Sybase SQL Server with ECA capabilities. This solution not only solves all of the limitations of the previous solutions, but also several advantages as described in the rest of the thesis.

4.1 Overview of the Open Server

Sybase Open Server is used with the ECA Agent. Sybase provides an OpenServer Library named Open Server™ Server-Library/C [SYB96]. The library is a non-preemptive multithread C library available from Sybase upon which the SQL Server is based. In fact, it is the basis for most Sybase server and middleware products including OmniSQL Server, all Sybase gateways, and Replication Server. The library allows for a C program to become a server to multiple Sybase client programs. This Server is called Open Server. It allows the programmer to authenticate logins, receive language requests, receive procedure calls, and return results in Tabular Data Stream (TDS) format, which all Sybase clients can receive. The following libraries may be used in Open Server:

- Open Client Client-Library/C
- Open Server™ Server-Library/C

Depending on its function, an Open Server's position in the client/server architecture is different. There are three functional categories for OpenServer: standalone, auxiliary, or gateway.

1) A standalone Open Server

Figure 4.1 shows a standalone Open Server application. A client connects directly to a standalone Open Server application and submits requests using remote procedure calls (RPCs), which is a command language the server is programmed to recognize, a cursor command, or any other kind of client command. The Open Server application programmer supplies code to process client commands.

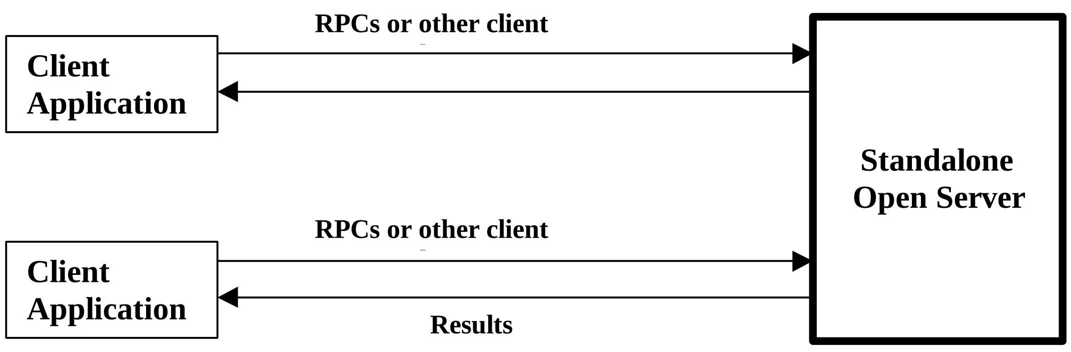


Figure 4.1: A Standalone Open Server

A Standalone Open Server does not satisfy the requirements of our mediated approach. Because it doesn’t connect to SQL Server, it cannot be used to create a middleware out of SQL Server.

2) An Auxiliary Open Server

An auxiliary Open Server application is shown in Figure 4.2. This category of Open Server application supports SQL Server by processing RPCs. The client connects directly to SQL Server and uses Transact-SQL for its language requests. To execute a procedure on the Open Server application, the client prefixes the procedure name with the name of the Open Server application, which causes SQL Server to initiate an RPC. An RPC is the only type of client command that can be sent to an Open Server via a SQL Server.

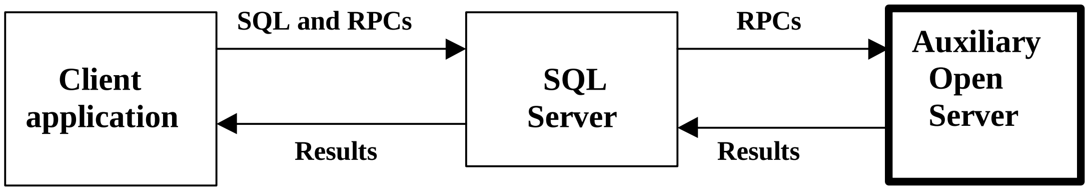


Figure 4.2: An Auxiliary Open Server

An Auxiliary Open Server can be used to create a layer for SQL Server. This architecture was discussed and implemented in [VAN96]. Since it suffers from the following limitations, we do not adopt this architecture for our implementation.

a) Deadlocks are possible.

The Auxiliary Open Server must maintain its own deadlock checking since the SQL Server can not be aware of the problem.

b) Poor performance.

c) Immediate trigger semantics require verification.

   There is no way to roll-back and RPC event notification if a transaction aborts. Verification of events is required.

d) The Auxiliary Open Server may be prevented from making additional inquires.

   The Auxiliary Open Server needs to make its own database inquires during condition evaluation, it must use a separate connection to the SQL server. It will have to wait for the initiating transaction to release its locks before the server can access any of the transaction's dirty pages

e) There is no way to determine the order of events.

   There is no accurate indicator to determine order the commit times of the transactions.

f) Lack of robustness.

   If the Auxiliary Open Server fails, the SQL Server will experience delays waiting for RPC time-outs. Updating the status of the Auxiliary Open Server can also have the effect of suspending other transactions which rely on the Auxiliary Open Server until the transaction completes.

3) A Gateway Open Server

Figure 4.3 shows an Open Server application in a gateway role. A gateway server enables a client to access a server that cannot accept the client connection directly. The gateway does not have to connect to a SQL Server, or for that matter, to any DBMS server. It could connect to a file system or an application program that can be treated as a server. The gateway program uses Server-Library to process client requests and Client-Library to connect to the SQL server.

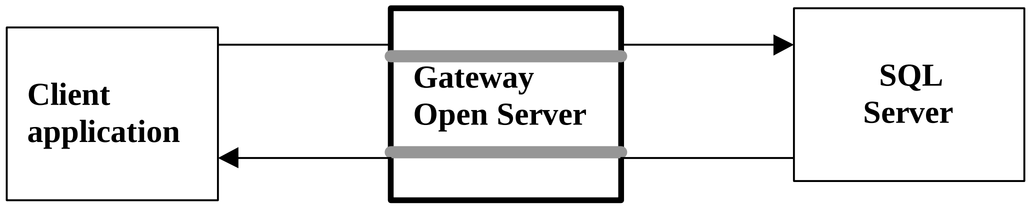


Figure 4.3 A Gateway Open Server

Gateway Open Server satisfies the requirement for the SQL Server to act as an intermediary for clients and servers that cannot communicate directly. By using the Gateway Open Server, the mediator approach used in this thesis is truly transparent to the client. And there are no significant limitations in this architecture. Gateway Open Server will be an important component of the ECA Agent.

4.2 Architecture of ECA Agent

ECA Agent is a multithread program. It is positioned between clients and the SQL Server so that the SQL Server can provide ECA capabilities with full transparency. From the user's point of view, the ECA Agent is a Virtual Active SQL Server. Not only can it provide all functions of the native SQL Server, but it also provides the ECA functions that active database requires. From a system's point of view, the ECA Agent is a middleware that connects the client and SQL Server and provides ECA service if necessary.

The architecture of ECA Agent is shown in Figure 4.4. There are eight functional modules in the ECA Agent:

- Gateway Open Server (GOS):

  GOS is responsible for the connection between a client and SQL Server. It provides the same interface as SQL Server to accept client commands and return the results to the client. GOS also accepts SQL requirements from the other parts of ECA Agent and forwards them to SQL Server. Results are returned to the corresponding part. Fully transparency is provided here.

- Language Filter:

  All client commands flow through the Language Filter. The ECA commands are seperated and sent to the ECA Parser while other SQL commands are sent back to the Gateway Open Server. Language Filter is also responsible for filtering different types of ECA commands. (i.e. primitive event command, composite event command, drop trigger command, etc.)

- ECA Parser:

ECA commands are filtered into the ECA Parser from the Language Filter. The ECA Parser scans and parses the commands. If there is no syntax error, the ECA Parser will create corresponding events and rules which depend on the Local Event Detector, send corresponding SQL to the Gateway Open Server,

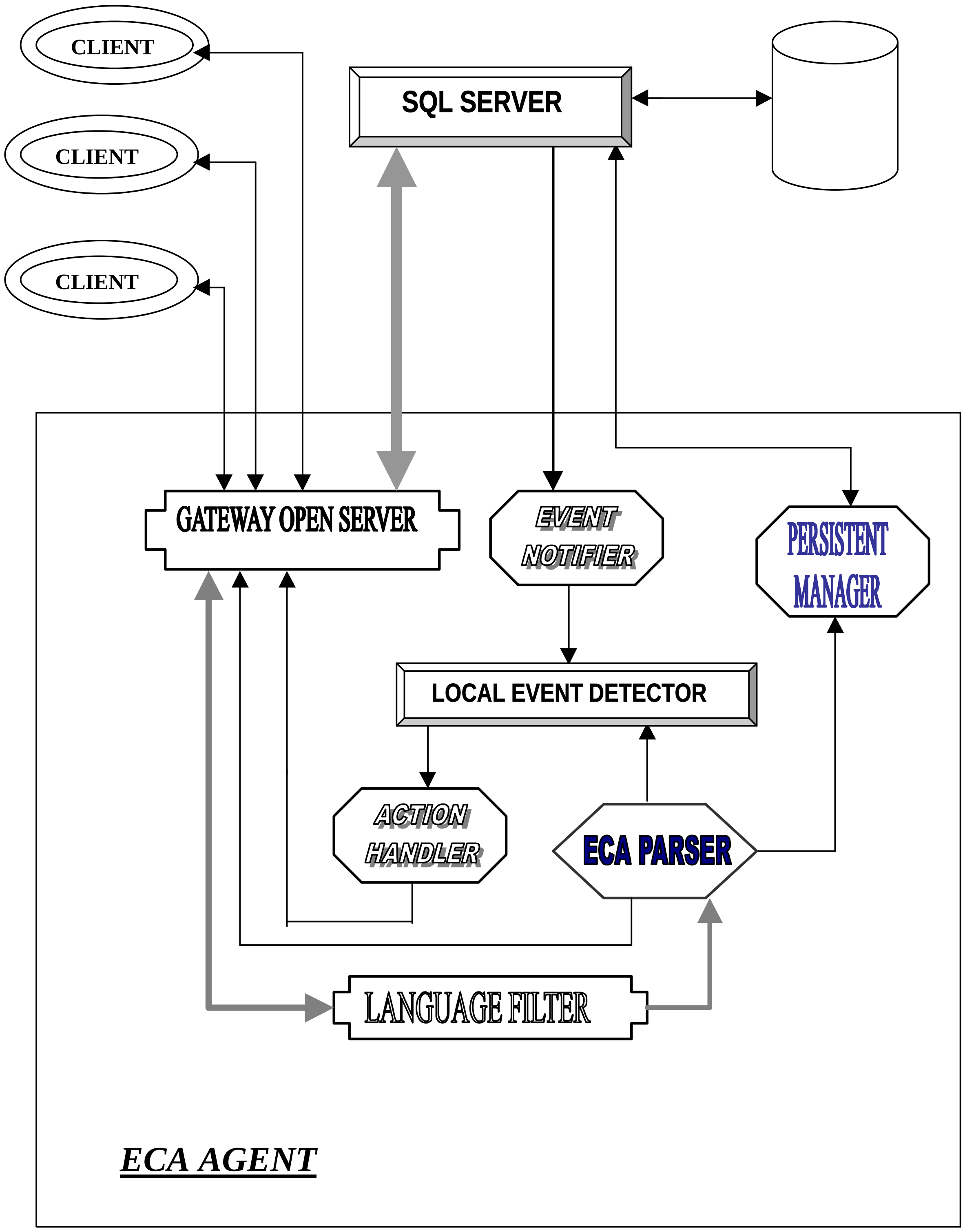


Figure 4.4: Architecture of ECA Agent

and send specifications of ECA rules to the Persistent Manager for persistent storing events and rules. If a parse error occurs, an error message is returned to Language Filter.

- Local Event Detector (LED):

  Since the SQL Server (trigger) detects only primitive events, LED is mainly responsible for composite event detection.

- Persistent Manager:

  All events and rules defined by a client need to be persistent. The information is stored in the following system tables using the SYBASE database:

  1) SysPrimitiveEvent: It is used to store information of primitive events.
  2) SysEcaTrigger: All triggers are stored here.
  3) SysCompositEvent: Information of composite events are stored in this table.

  Persistent Manager stores all ECA information into these tables when it receovess information from ECA Parser. On ECA Agent starting or recovery, Persistent Manager restores and creates all events and rules from these tables.

- Event Notifier:

  As soon as a primitive event occurs, SQL Server sends notification to the Event Notifier. The Event Notifier is responsible for receiving the notification, formatting it and sending the formatted notification to the LED.

- Action Handler:

  Once an event occurs, it calls the actions defined on this event. The Action Handler processes these actions, changes them into corresponding SQL commands (call corresponding stored procedures), and send SQL commands to the Gateway Open Server. The Gateway Open Server will send them to the SQL Server, get the results from the Gateway Open Server, and send them to the client as appropriate.

4.3 Workflow of ECA Agent

The ECA Agent is responsible for two major functions: “create ECA rules” and “event notification and action.”

The workflow of “create ECA rules” is shown in Figure 4.5. It includes seven steps:

1) Client provides input for creating a new ECA rule.
2) The command goes to the Gateway Open Server.
3) The Gateway Open Server forwards the command to the Language Filter.
4) The Language Filter checks if the command is an ECA command. If so, the Language Filter scans the command and sends the command to the ECA Parser. Otherwise, the command is returned to the Gateway Open Server. And the Gateway Open Server forwards the command to SQL Server and returns the result to the client.
5) The ECA Parser analyzes the command and checks for errors. If an error is detected, a message is returned to the Gateway Open Server. If no error is found, the ECA Parser creates event graphs using the LED, sends the new SQL commands to the Gateway Open Server, and forwards persistent requirements to the Persistent Manager.
6) Gateway Open Server sends SQL commands to the SQL Server and Persistent Manager if necessary, then returns the results to the client.

7) If the Persistent Manager receives the persistent requirement, it persistently stores ECA rules.

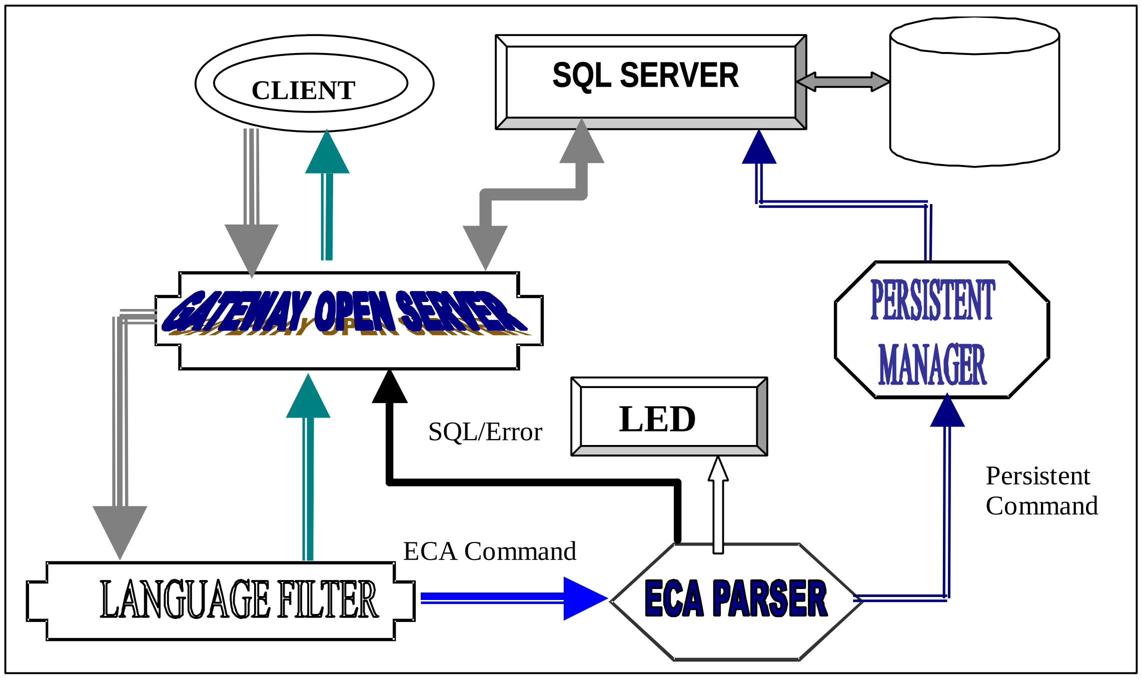


Figure 4.5: The Workflow of Creating ECA rules.

Figure 4.6 shows the workflow of “event notification and action.” There are six steps for event notification and action:

1) The client sends SQL commands to the Gateway Open Server. If the commands are not ECA commands, the commands will pass through to the SQL Server.
2) If the commands invoke triggers, the SQL Server sends a notification to the Event Notifier.
3) Event Notifier receives the notification from SQL Server, decodes the notification message, and notifies the LED.

4) LED, after receiving the notification detects if occcurrences of one or more events. If so, LED sends event information to the Action Handler.

5) Action Handler processes event information, changes it into SQL commands, and sends it to Gateway Open Server.

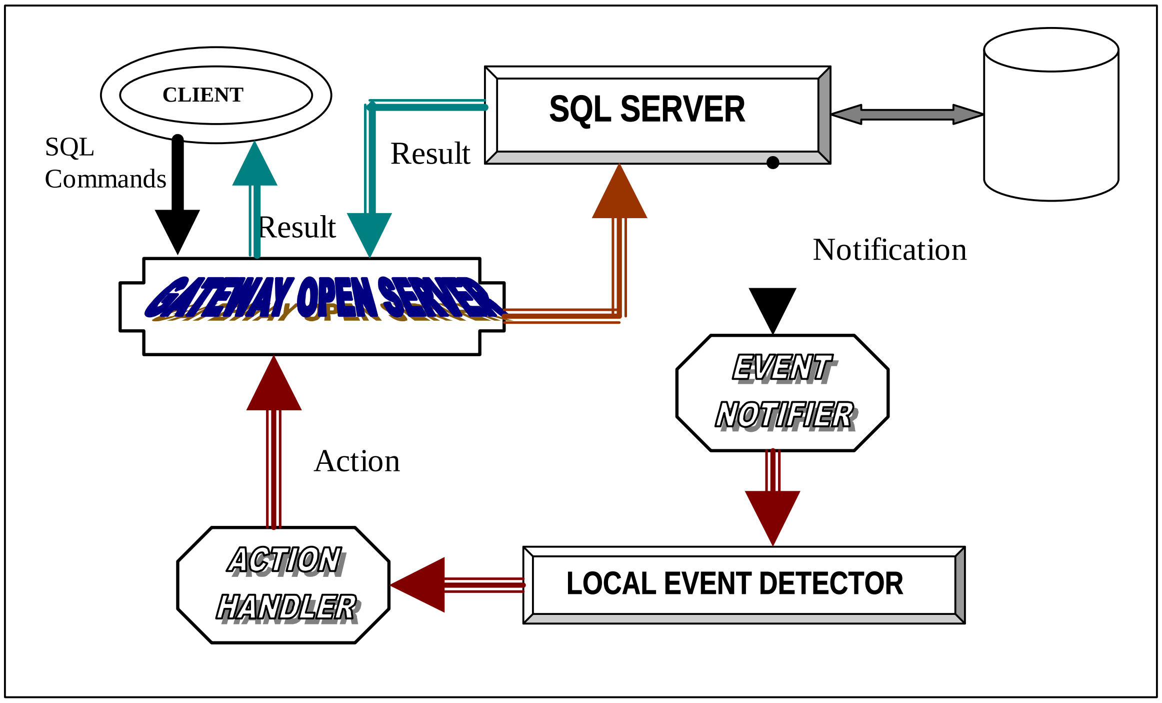


Figure 4.6: Workflow of event notification and action

6) The Gateway Open Server sends the commands to the SQL Server and returns the results to the client.

4.4 Functionality and Features

Currently, ECA Agent provides extends the functionality of for Sybase SQL Server with the following:

1) The client can create multiple triggers on the same event.

2) The client can create composite events and triggers on them.

3) It can reuse previously defined events (both primitive & composite).
4) It drops triggers associated with primitive or composite events.
5) Once events are created, they will be made persistent.

All primitive events and composite events are detected, and actions are invoked in the SQL Server.

The advantages of the ECA Agent are obvious. The ECA Agent not only overcomes the limitations identified in the previous approach (discussed in Chapter 3), but it also has its own advantages. In summary, the features of the ECA Agent are as follows:

1) The primitive event is atomic.
2) ECA rules can be defined dynamically: The user can define ECA rules at any time. Once the ECA rules are defined, the ECA rules are effective in the system.
3) Transparency: As the agent has been added as a mediator, the clients are not aware of its presence. Also, ECA rules can be created in the system using the currently available interface. Any existing SQL Server application can continue to function without modification.
4) User Friendly: A user only needs to know the event operations to define an ECA rule. The event and rule definitions are similar to that of the standard SQL trigger syntax.
5) System consistency: None of an existing DBMS's functionality is lost.
6) Extensible: A more distributed architecture can be designed.

7) Semantic Independence: There is nothing about active database semantics that demands that the rule execution must be integrated into the DBMS.
8) Scaleable Architecture.
9) Persistent: By using DBMS storage mechanism, all ECA rules are persistent.
10) Good Modularity/Maintenance.
11) All events can be detected: Events will not be lost.
12) Parallel execution of ECA Agent and SQL Server.

4.5 Integration with Sentinel

ECA Agent uses LED as its composite detector. Also LED provides an interface to connect to Sentinel GED. This enables connection of this architecture with Sentinel. Once we start the GED server in Sentinel, events defined in Sybase can send event requests to GED and GED can receive event notification from ECA Agent. Figure 4.7 shows the relationship between Sentinel and ECA Agent:

A new architecture CEDaR (Composite Event Detection and Rule execution) is proposed for Sentinel [CHA98] to support ECA rules for distributed heterogeneous environments. CORBA is used as the underlying transport mechanism that provides connectivity to multiple heterogeneous systems. CEDaR makes integration with ECA Agent easier. An interface should be defined for ECA Agent by using CORBA IDL. The integration architecture is shown in Figure 4.8.

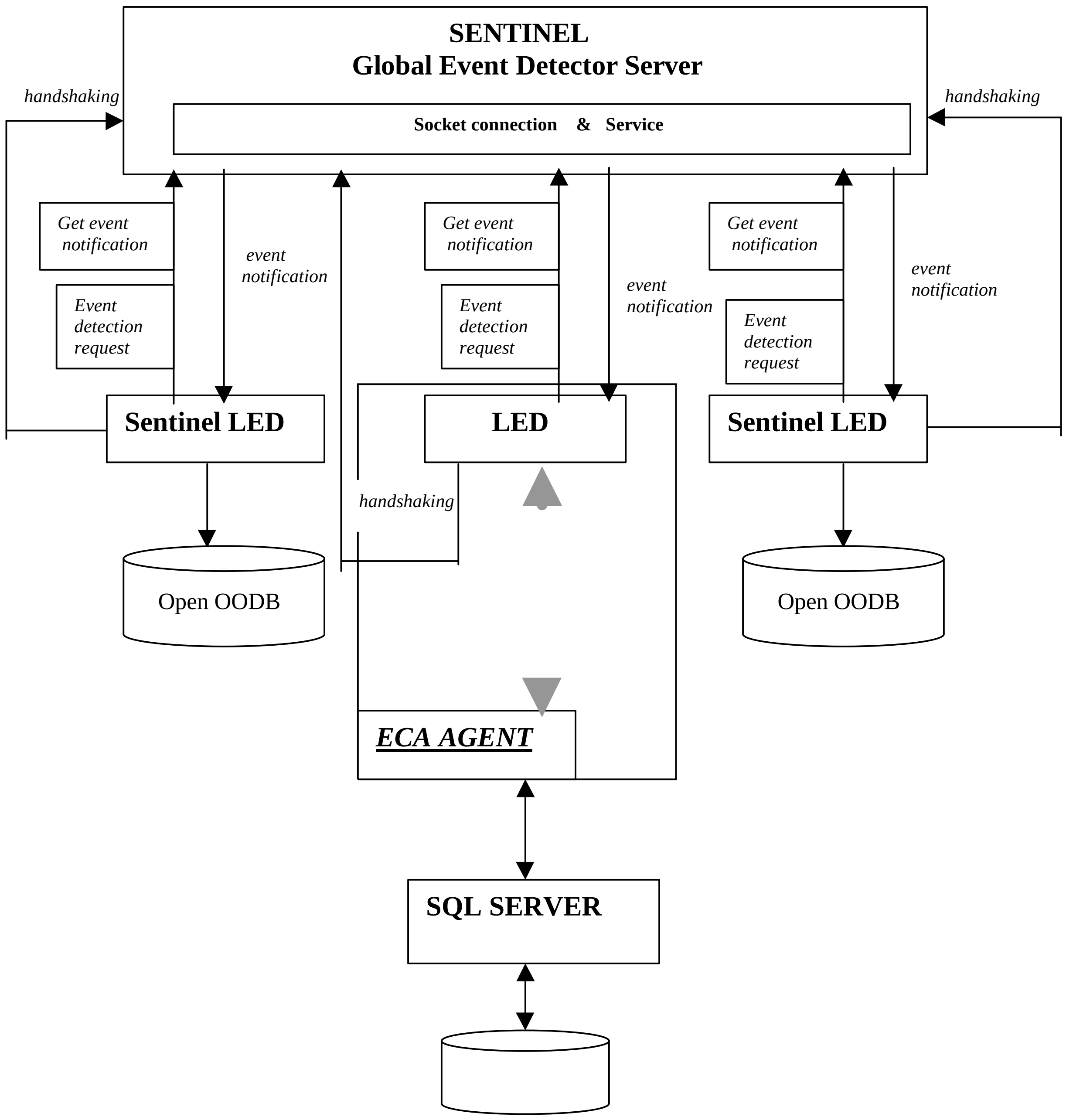


Figure 4.7: Integration with Sentinel

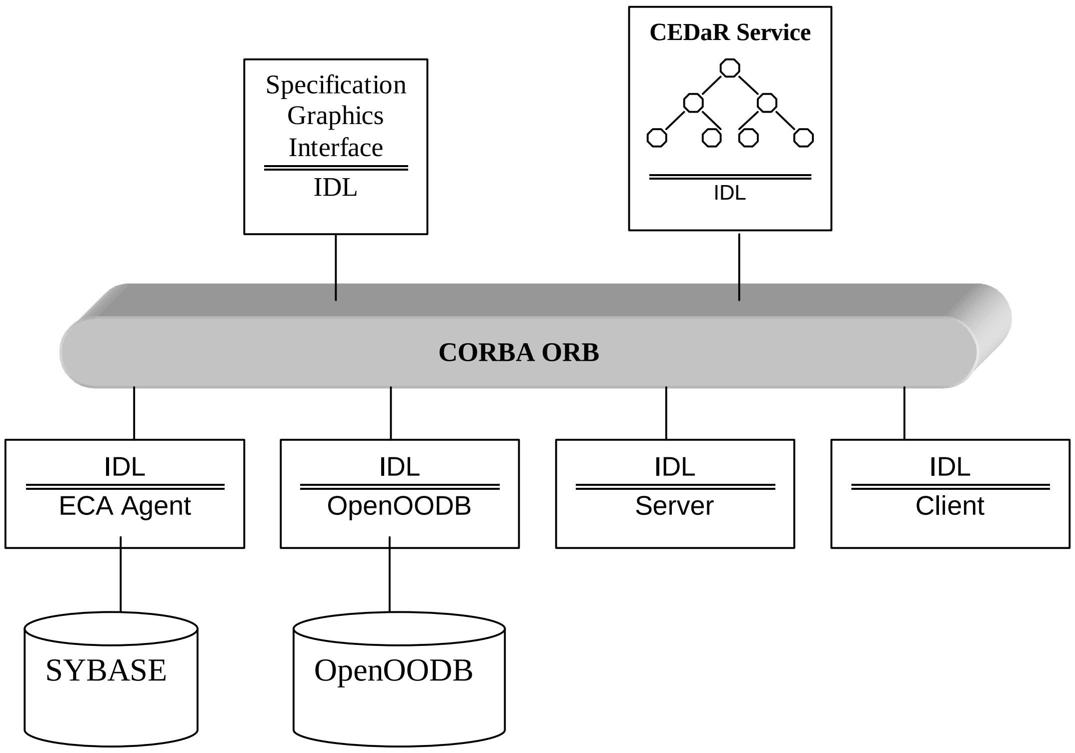


Figure 4.8: Integration with CEDaR

4.6 Conclusion

This architecture proposed in this chapter is flexible, transparent, and supports all the necessary features. This architecture is likely to be specific systm independent as well. We hope to port this architecture to other COTS relational database systems.

CHAPTER 5
DESIGN AND IMPLEMENTATION OF THE GATEWAY OPEN SERVER AND THE PERSISTENT MANAGER

The Gateway Open Server provides an interface between clients and the SQL Server. It plays an important role in the ECA Agent. This chapter discusses the implementation of the Gateway Open Server.

5.1 Setting Up the Environment

To create an Open Server, we need to set up the environment. To set up the environment, environment variables need to be set and the network connections in the interface file need to be configured so that clients and the server can communicate with each other.

The following environment variables are set for our Gateway Open Server:

- SYBASE = local/sybase

  SYBASE implies the path of the SYBASE directory.

- DSQUERY = SYB_ECA_AGENT

  Here we set DSQUERY for clients, it points to the Gateway Open Server name: “SYB_ECA_AGENT”.

- DSLISTEN = SYB_ECA_AGENT

  DSLISTEN is set for server, here is the name of Gateway Open Server “SYB_ECA_AGENT”.

The interface file need to be configured to contain network information about servers on the system, including the server name, the network address, and the port number on which the server listens for queries. Figure 5.1 shows the content of the interface file we set for our Gateway Open Server:

```
## CISE-DATASERVER-0 on fountain
##      Services:
##              query   tcp     (9455)
##              master  tcp     (9455)

CISE-DATASERVER-0 5 5
      query tli tcp /dev/tcp \x000224ef80e3a2c20000000000000000
      master tli tcp /dev/tcp \x000224ef80e3a2c20000000000000000

## SYB_ECA_AGENT on localhost
##      Services:
##              query   tcp     (12000)
##              master  tcp     (12000)

SYB_ECA_AGENT
      query tli tcp /dev/tcp \x00022ee07f000001000000000000000
      master tli tcp /dev/tcp \x00022ee07f000001000000000000000
```

Figure 5.1 Interface File for Gateway Open Server

Master lines are used by server applications to listen for queries over the network. Query lines are used by client applications to connect to servers over the network. The Gateway Open Server name is **SYB_ECA_AGENT**, which can work on any local machine and has a port number of 12000. SQL Server is **CISE-DATASERVER-0**, which runs on the machine "fountain" and connects with port number 9455.

When a client connects to a server, it does the following:

1) Determines the name of the server by referring to the DSQUERY environment variable.
2) Checks the interface file for an entry whose name matches the name of the server, stops at the first instance of that name, and reads the query line.
3) Uses the network information provided by the query line to connect to the server.

Servers use the interface file to listen for clients. To listen for queries, a server:

1) Determines its own name by using the DSLISTEN environment variable
2) Looks in the interfaces file for an entry whose name matches the name by which it knows itself and finds the master line
3) Uses the network information provided by the master line to listen for queries.

5.2 Implementation of the Gateway Open Server

The main functions of the Gateway Open Server are:

- Set up structures: CS_CONTEXT is the basic control structure for an Open Server. There exist other structures that also require set up.
- Listens for connection requests from clients and SQL server.
- Processes client requests and SQL server results.
- Clean up: Cleans all structures and allocated memory for the Gateway Open Server when the server is normally shut down.

Figure 5.2 shows the main steps for implementing Gateway Open Server.

After Gateway Open Server starts running, it listens for the client requests. If the request i

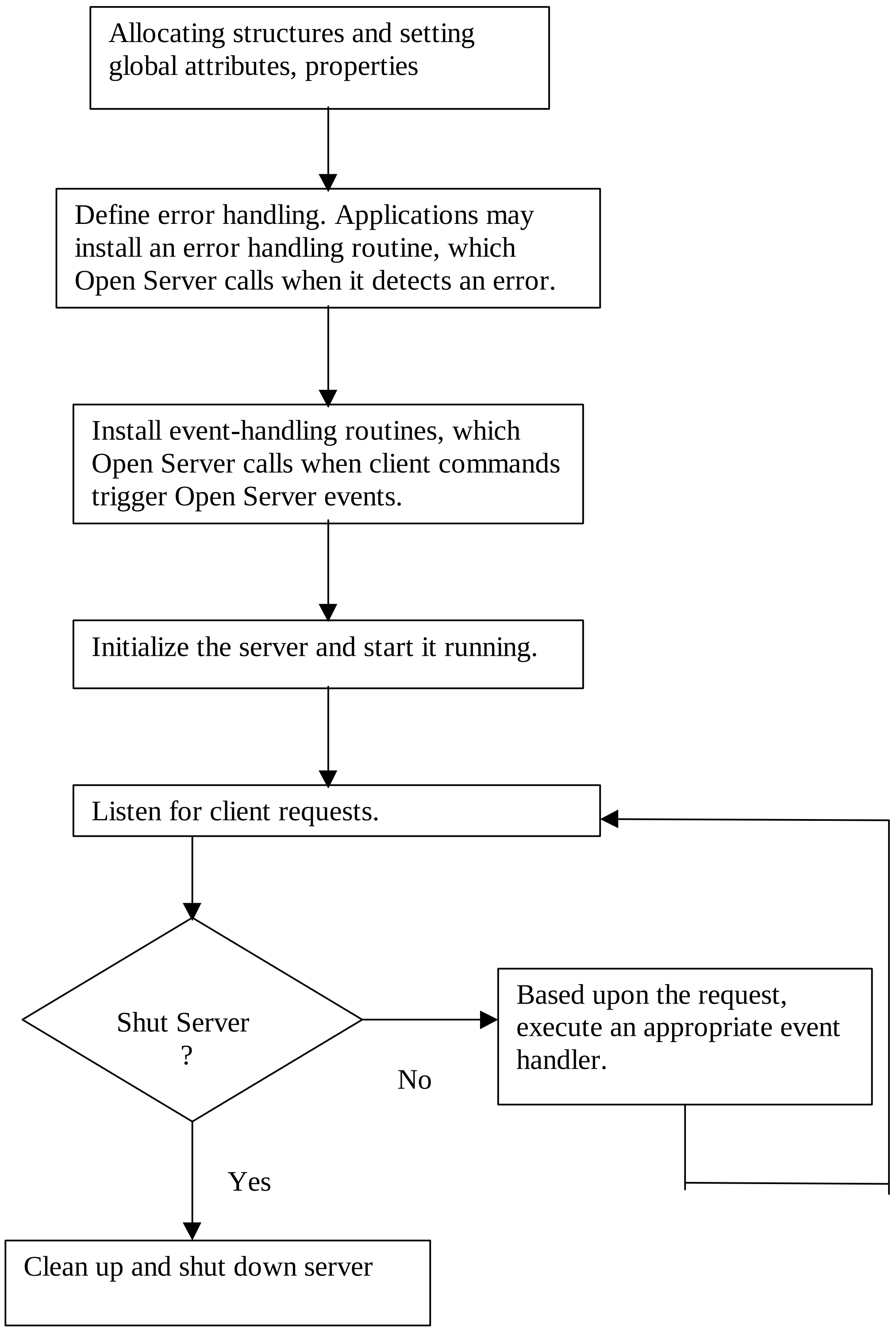

Allocating structures and setting global attributes, properties
Define error handling. Applications may install an error handling routine, which Open Server calls when it detects an error.
Install event-handling routines, which Open Server calls when client commands trigger Open Server events.
Initialize the server and start it running.
Listen for client requests.
Shut Server ?
No
Based upon the request, execute an appropriate event handler.
Yes
Clean up and shut down server

Figure 5.2: Implementation Gateway Open Server

is to shutdown server, it will clean up the environment and shut off the server. Otherwise, it will invoke different event handlers based upon the requirement.

The Gateway Open Server responds to requests from clients. When the Gateway Open Server triggers an event, it places the event in the active thread's event queue. The thread then executes a routine (event handler) that processes the event. There are ten event handlers in our Gateway Open Server. Figure 5.3 shows the event handlers we have defined.

Event	**Routine**	**Function**
SRV_ATTENTION	ActiveAttnHandler	Process client attention requests.
SRV_BULK	ActiveBulkHandler	Bulk data received from a client.
SRV_CONNECT	ActiveConnHandler	Process client connection requests.
SRV_CURSOR	ActiveCurHandler	Process client cursor requests.
SRV_DISCONNECT	ActiveDiscHandler	Process client disconnections.
SRV_DYNAMIC	ActiveDynHandler	Process client dynamic SQL requests.
SRV_LANGUAGE	ActiveLangHandler	Process client language requests.
SRV_RPC	ActiveRpcHandler	Process RPC requests.
SRV_OPTION	ActiveOptHandler	Process option commands.
SRV_START	ActiveStartHandler	Installs handlers for all events.

Figure 5.3 Event Handlers for the Gateway Open Server

When the Gateway Open Server starts, the SRV_START event is invoked and event handler ActiveStartHandler is called. ActiveStartHandler installs the other nine-event handlers that are used to respond to user's requests.

5.3 Implementation of Persistent Manager

Once the Gateway Open Server starts, it generates a thread that connects to the SQL Server directly by using Client-Library. The thread is a Persistent Manager that will control and manage all persistent ECA rules. The connection with the SQL Server should be granted high privilege, for example DBA, so that it can have higher privilege to create or delete system tables than ordinary users.

The Persistent Manager is responsible for the management of ECA rule database. Listed below are functions of the Persistent Manager:

1) Maintain ECA Agent system tables.
2) Recovery of ECA rules.
3) Persist ECA rules.
4) On system startup, restore all ECA rules.
5) Add or delete ECA rules from ECA Agent system tables.
6) Keep track of the occurrence of each event.

5.3.1 System Tables of the ECA Agent

To implement these functions, three system tables are created for the ECA Agent. Figure 5.4 shows the structure of table SysPrimitiveEvent which is used to store all primitive events. The structure of table SysCompositEvent is shown in Figure 5.5. This table is used to store all composite events. Table SysEcaTrigger whose structure is shown in Figure 5.6 stores all triggers.

Column_name	**Type**	**Length**	**Nulls**
dbName	varchar	30	NULL
userName	varchar	30	NULL
eventName	varchar	30	NULL
tableName	varchar	30	NULL
operation	varchar	20	NULL
timeStamp	datetime	8	NULL
vNo	int	4	NULL

Figure 5.4: Schema of SysPrimitiveEvent Table

Column_name	**Type**	**Length**	**Nulls**
dbName	varchar	30	NULL
userName	varchar	30	NULL
eventName	varchar	30	NULL
eventDescribe	text	text	NULL
timeStamp	datetime	8	NULL
coupling	char	10	NULL
context	char	10	NULL
priority	char	10	NULL

Figure 5.5: Schema of SysCompositEvent Table

Column_name	**Type**	**Length**	**Nulls**
dbName	varchar	30	NULL
userName	varchar	30	NULL
triggerName	varchar	30	NULL
triggerProc	text	text	NULL
timeStamp	datetime	8	NULL
eventName	varchar	30	NULL

Figure 5.6: Schema of SysEcaTrigger Table

Since system tables are created in the SYBASE database system, they are maintained in the SYBASE database system based upon SYBASE storage management.

Persistent Manager is responsible for managing the data in these tables and providing the persistent service to ECA Agent.

5.3.2 Implementation Details of Persistent Manager

The functions of Persistent Manager are discussed in the previous sections. Figure 5.7 shows how Persistent Manager works and how it is implemented.

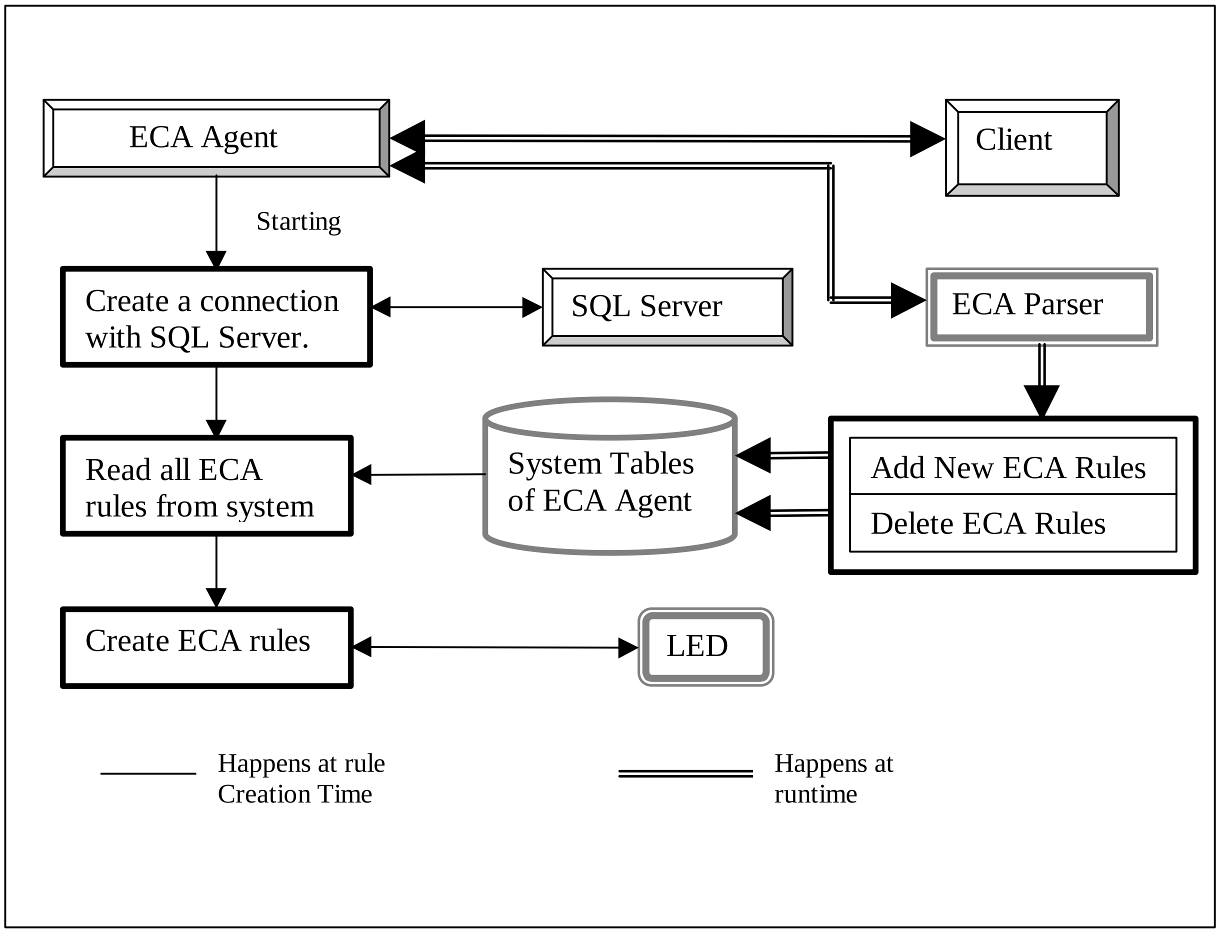

ECA Agent
Client
Starting
Create a connection with SQL Server.
SQL Server
ECA Parser
Read all ECA rules from system
System Tables of ECA Agent
Add New ECA Rules
Delete ECA Rules
Create ECA rules
LED
Happens at rule Creation Time
Happens at runtime

Figure 5.7: Implementation of Persistent Manager

When the ECA Agent starts, it creates a thread that connects to the SQL Server by using functions in the Client-Library. The Persistent Manager runs in this thread (dark square in Figure 5.7). It creates ECA rules from the system tables of the ECA Agent.

After the ECA Agent starts, the Persistent Manager waits for the requirement from ECA Parser. If there is an add or delete ECA Rules requirement, Persistent Manager will persist the ECA Rules to the system tables or delete them from system tables.

CHAPTER 6
IMPLEMENTATION OF PRIMITIVE EVENTS

Though the SQL Server provides triggers, it only supports a limited way of defining events, conditions, and actions. That is sufficient for many applications that require full-fledged active capability. The ECA Agent uses SQL Server's primitive trigger mechanism and extends it significantly. Implementation details of primitive event detection are discussed in this chapter.

6.1 Trigger Extensions of the ECA Agent

While triggers work well in the SYBASE SQL Server, there are a number of limitations. For example, triggers must be written in Transact-SQL whose limitations make it difficult for supporting ECA form of rule evaluation. Following is a list of restrictions of SYBASE trigger mechanism (SQL 92 compliant). Most of these restrictions are true for other SQL 92 compliant relational DBMSs as well.

- Complex data types is not allowed.
- There is no direct access to C, to other programs, or to the underlying operating system.
- Only atomic values (and not tables) may be passed as parameters to stored procedures.
- A trigger cannot be applied to more than one table.
- Each new trigger on a table for the same operation (insert, update, or delete)

overwrites the previous one. No warning message is given before the overwrite occurs.

Some of the restrictions make the system inflexible. For example, a trigger cannot be applied to more than one table, or each new trigger on a table for the same operation overwrites the previous one, etc. The ECA Agent, described in this thesis, overcomes these restrictions so that it can provide the SQL Server with a complete Active Database capability. Below, we list the functionality added by the ECA Agent:

- Primitive Events: If an event is defined on a base table for a certain operation, this event is called a Primitive Event. If two events are defined on the <u>same</u> table for the <u>same</u> operation, we define these two events to be the same. We will assign the same name in the system.
- Composite events: A composite event is built using primitive events, event operators, and previously defined composite events.
- Triggers can be defined on any events.
- Additional triggers can be defined on the same table for the same operation. Once the event occurs, all triggers defined on that event will be invoked.
- A trigger can be applied to more than one table and more than one operation through the use of composite events.
- Triggers can be dropped. When there are no other triggers defined on an event, the event is deleted by system.

6.2 Syntax of Primitive Events

To provide user transparency, the syntax of an ECA Rule definition proposed by the ECA Agent is kept the same as that of a trigger definition in Sybase except for the naming of an event. Figure 6.1 shows the syntax of the Primitive Event definition. The only difference here is the keyword **event**.

```
create trigger [owner.] trigger_name
on [owner.] table_name
for operation
 [event event_name [coupling_ mode] [parameter_context] [priority]]
as SQL_statements

operation := insert | delete | update
parameter_context := RECENT | CHRONICLE | CONTINUOUS
                      | CUMULATIVE
coupling_mode := IMMEDIATE | DEFERED | DETACHED
priority := positive integer
```

Figure 6.1: Syntax of a Primitive Event

A primitive event is defined on a table for an operation (delete, update, insert). A trigger can be defined on any event. The default coupling mode is **IMMEDIATE**, and the default parameter context is **RECENT**. The action function is written as SQL code, using the keyword “**as**”. The action is invoked in the SQL Server.

Once an event is defined, the user can define additional triggers on the event. Figure 6.2 shows the Syntax of defining triggers on a previously defined event.

6.3 Naming Mechanism

A user can assign a name for an object (a trigger or an event) in the system. Since SYBASE supports a multi-user, multi-database environment, we cannot use the name that the user assigns as the internal system wide identifier. All user defined names are changed into an internal name that is unique across user and database-names. A user need only be concerned with the names assigned by him/her. This naming scheme is consistent

create trigger *[owner.] trigger_name*
event *event_name [coupling_ mode] [parameter_context] [priority]*
as *SQL_statements*

operation := insert | delete | update
parameter_context := RECENT | CHRONICLE | CONTINUOUS
| CUMULATIVE
coupling_mode := IMMEDIATE | DEFERED | DETACHED
priority := positive integer

Figure 6.2: Syntax of Defining a Trigger on Existing Event

With the way Sybase expands user-defined object names. If a user assigns an object ***objectName,*** this name will be changed into the following system-wide internal name:

DatabaseName.userName.objectName

6.4 Language Filter

When a client forwards a language request command such as an SQL statement to the Gateway Open Server, an SRV_LANGUAGE interrupt occurs, and the event handler ActiveLangHandler is invoked. The Language Filter is called by this event handler and to process the incoming command. The input is analyzed as shown in Figure 6.3. If the command is not an ECA related command, the original command will be returned to the

Gateway Open Server to pass it on to the SQL Server. Otherwise, the handler will invoke a different function for each of the five different ECA requests:

- Primitive Event: Primitive Event Parser will be called.
- Composite Event: Composite Event Parser will be called.
- Drop Trigger: Drop trigger function will be called.
- Trigger created on existing Primitive Event: Existing Primitive Event Parser will be called.
- Trigger created on existing Composite Event: Existing Composite Event Parser will be called.

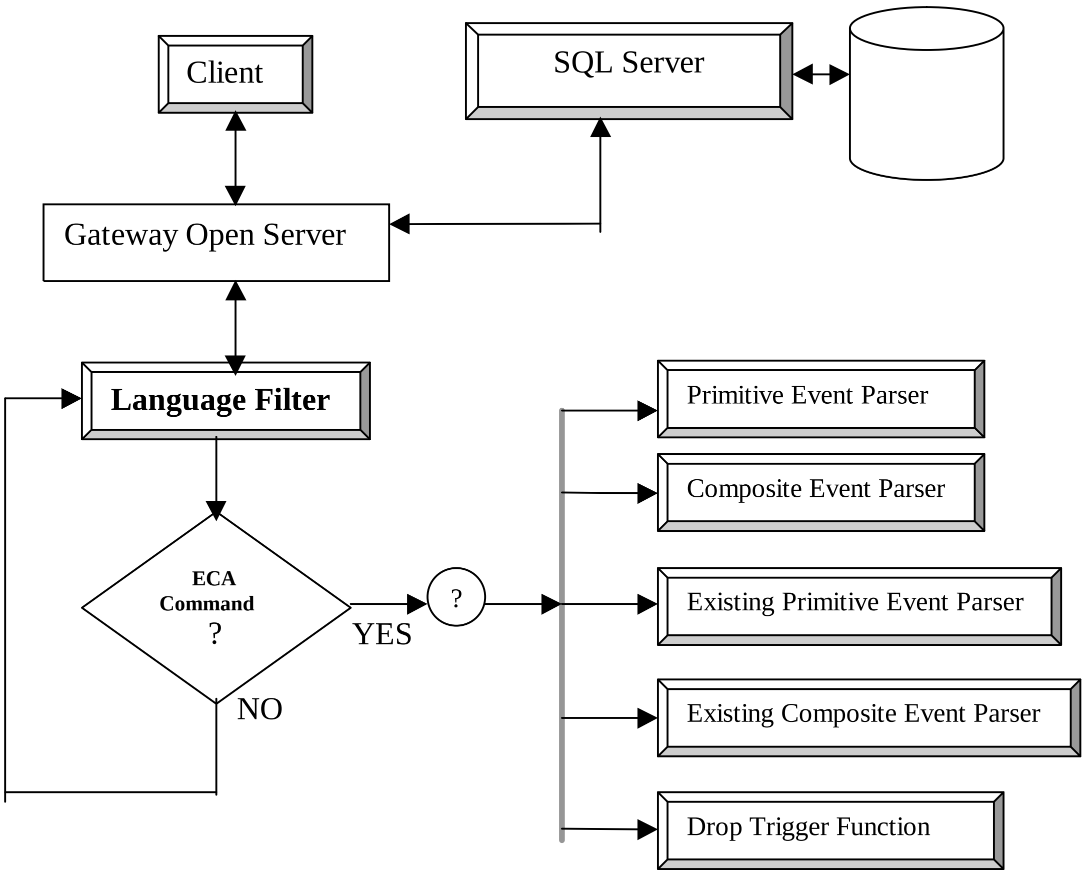


Figure 6.3: Language Filter

6.5 Parsing and Generating Primitive Event

The five steps to parse and generate a Primitive Event are shown in Figure 6.4.

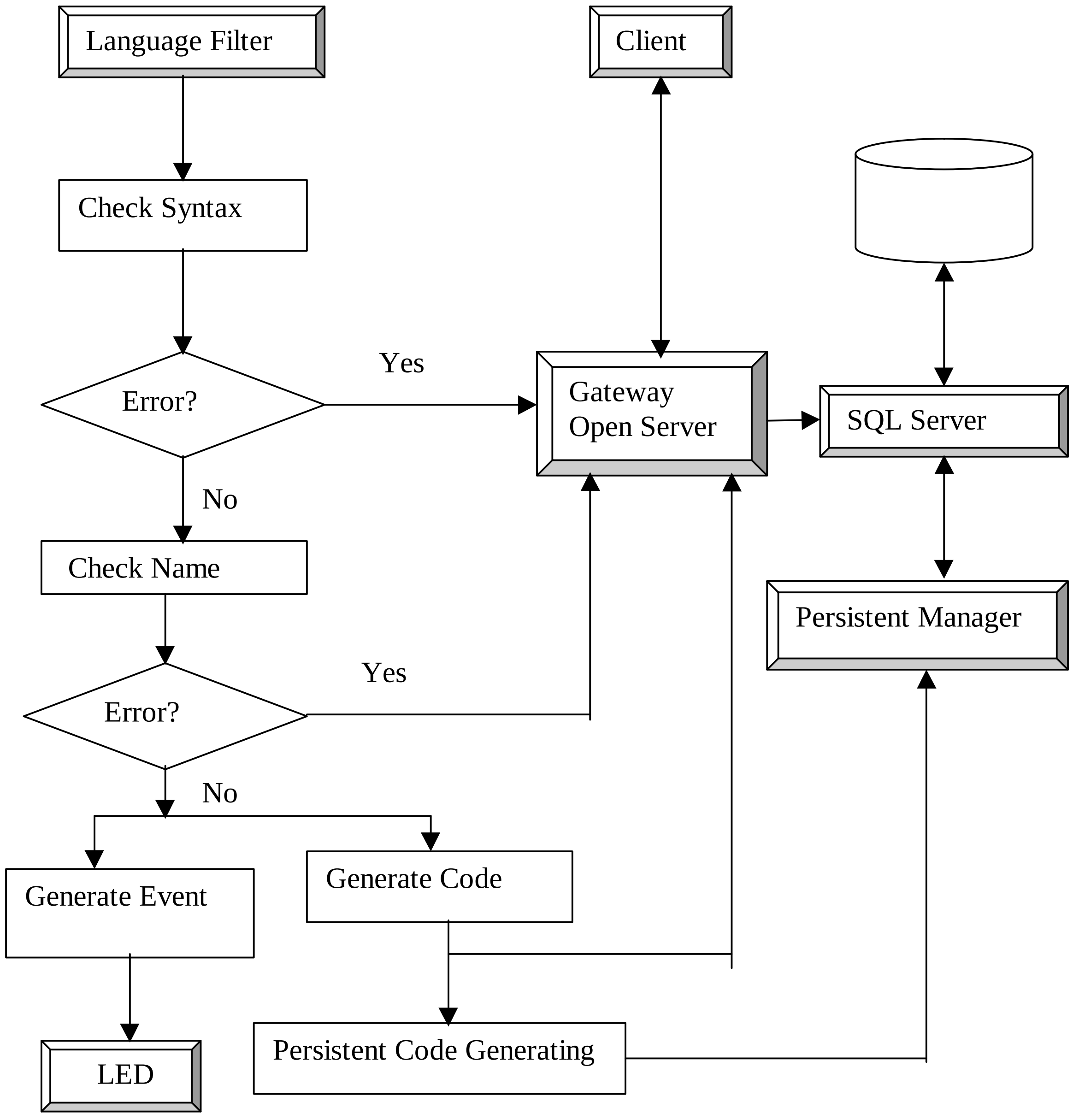


Figure 6.4: Primitive Event Parser

Here we give an example to explain how this works:

Example 6.1: Suppose the command is:

create trigger t_addStk on stock for insert

event addStk

as print " trigger t_addStk on primitive event addStk occurs"

select * from stock

and the contents of the stock table is shown in Figure 6.5

```
Column_name     Type          Length        Nulls
--------------  --------------  ------ ----      ----- -----

symbol          char            6             NULL
Co_name         char            20            NULL
price           money           8             NULL
time            datetime        8             NULL
```

Figure 6.5: Schema of Table stock

The Primitive Event Parser starts parsing this command:

1) Syntax checking: The command is scanned. If there is a syntax error in the command, an error message is returned to the Gateway Open Server, and the Gateway Open Server then returns an error message to the client. If there is no syntax error, all object names are replaced by the internal system names, e.g.:

   create trigger *sentineldb.sharma.t_addStk* on stock for insert

   event *sentineldb.sharma.addStk*

   as

   print " trigger t_addStk on primitive event addStk occurs"

   select * from stock

2) Duplicate object name checking: All object names (internal system name) including the trigger name and the event name are checked. If there is a duplicate name, an error message is returned to the Gateway Open Server.

3) ECA code generation: SQL code is generated. SQL code generated includes:

- Creation of two tables, if they do not already exist, for processing the parameter context. One table is created for storing the deleted tuples, whose internal system name is:

  DatabaseName.userName.tablename_deleted

  The other table is for storing the inserted tuples, whose internal system name is:

  DatabaseName.userName.tablename_inserted

  These two tables are created using the name of the table on which the primitive event is created. The schema is almost the same as the table with an additional attribute **vNo** (for recording the unique event occurrence value). The value of this attribute will be used for composing parameters for the parameter context specified.

  For the example 6.1, the following tables are created: ***sentineldb.sharma.stock_inserted*** and ***sentineldb.sharma.stock_deleted.*** The schema of these two tables is shown in Figure 6.6.

Column_name	**Type**	**Length**	**Nulls**
---------------	---------------	------ ----	----- -----
symbol	char	6	NULL
Co_name	char	20	NULL
price	money	8	NULL
time	datetime	8	NULL
vNo	int	4	NULL

Figure 6.6: Schema of two generating tables.

- Create a stored procedure to store the trigger action. The name of stored procedure will be:

DatabaseName.userName.TriggerName__Proc

- Get the occurrence number of the table from table **SysPrimitiveEvent** and put the number into table Version.
- Create a trigger for the primitive event. The main functions of this trigger are:

a) Transfer relevant tuples from **inserted** or **deleted** into two generating tables for the context.

b) Send notification to Event Notifier.

c) Invoke an action.

The code is generated for the example 6.1 is shown in Figure 6.7. This code will be sent to the Gateway Open Server, the Gateway Open Server, which will pass it on to the SQL Server, and the result will be returned to the client.

4) Generation of persistent code: SQL code is generated and persisted. The code includes the information written into tables created by the ECA agent: SysEcaTrigger and SysPrimitiveEvent. These SQL statements are sent to the Persistent Manager.

   Two **insert** statements are generated for the example 6.1:

   insert SysEcaTrigger values (“sharma”, “t_addStk”, “sentineldb.sharma.t_addStk__Proc”, getdate(), “addStk”)

   insert SysPrimitiveEvent values(“sharma”, “addStk”, “stock”, ”insert”, getdate(), 0)

5) Primitive Event Generation: A primitive event is created in LED.

In example 6.1, the event name is **sentineldb.sharma.addStk.**

PRIMITIVE *eventPoint=new PRIMITIVE(sentineldb.sharma.addStk

“Sybase_Event","begin", sentineldb.sharma.addStk);

Details of all events are stored in the global link list **EventPrimitive** for use at a later time.

```
/* create two tables. */
select * into sentineldb.sharma.stock_inserted from stock where1=2
alter table sentineldb.sharma.stock_inserted add vNo int null

/*create stored procedure*/
create procedure sentineldb.sharma.t_addStk__Proc as
print "* trigger_addStk on primitive event addStk occurs"
select * from stock

/* create trigger*/
create trigger sentineldb.sharma.t_addStk
on stock
for insert
as
 insert sentineldb.sharma.stock_inserted
          select * from inserted,Version
 select syb_sendmsg("128.227.205.215", 10006, " sharmastockinsert
begin sentineldb.sharma.addStk") /* Notification */

/* Get and change occurence Number */
update SysPrimitiveEvent set vNo=vNo+1 where eventName
= “sentineldb.sharma.addStk “
delete Version insert Version select vNo from SysPrimitiveEvent
where eventName=” sentineldb.sharma.addStk”

/* action function */
execute sentineldb.sharma.t_addStk__Proc
```

Figure 6.7 Code Generation in Example 6.1

6.6 Creating Triggers on Existing Event

6.6.1 Primitive Event List

To provide users with the capability of defining many triggers on the same event, we keep track of each event and all triggers defined on that event. A list (termed **EventPrimitive** list) is used to store primitive event details and trigger functions. Figure 6.8 shows the structure of this list. When a new event is created, an event node is inserted into **EventPrimitive**, and a trigger node is inserted into **Triggers** of **EventPrimitive**.

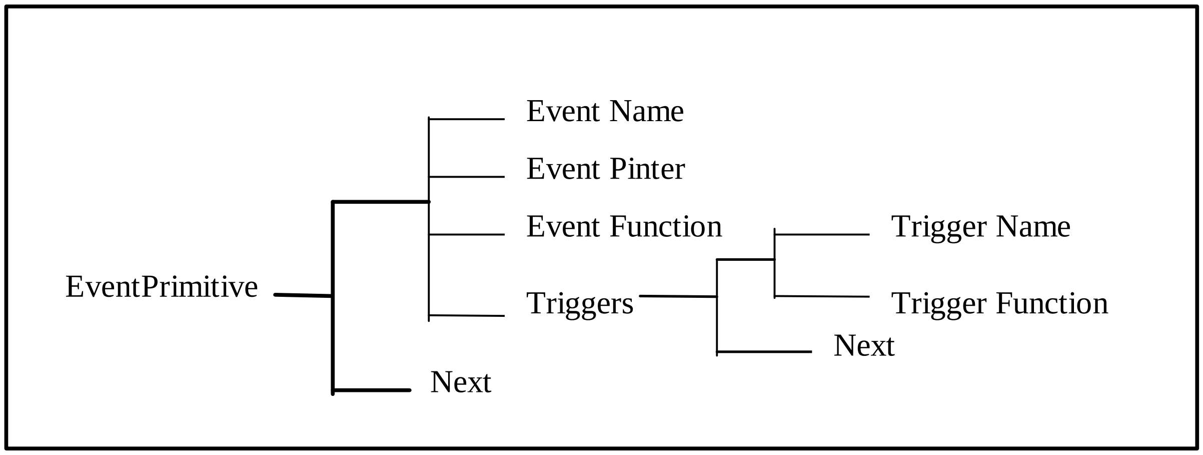


Figure 6.8: Structure of the list EventPrimitive

6.6.2 Implementation of Triggers on an Existing Event

As we store the details of an event in the Because an **EventPrimitive** list, when a new trigger is defined on an existing event, the following three steps are applied to the new trigger:

1) Syntax checking: If there is no syntax error, go to step 2, otherwise an error message is sent to the Gateway Open Server.
2) Duplicate object name checking: If there is no duplicate trigger name, go to step 3. Otherwise, an error message is sent to the Gateway Open Server.

3) The state of the new trigger is inserted into the **EventPrimitive** list for the existing event.
4) Code Generation: New code is generated and sent to the Gateway Open Server.
   - A store procedure is created for this trigger action. The name of the stored procedure is:

     DatabaseName.userName.TriggerName__Proc

   - A new trigger is created in the SQL Server. Code is generated by adding and executing the stored procedure statement.
5) Persistent code generation: SQL code is generated for persistence. The code includes the information written into the SysEcaTrigger table. It is sent to the Persistent Manager.

We illustrate the above with another example (example 6.2). Suppose the user defines another trigger on the event **addStk** that was created in example 6.1:

Example 6.2:

```
create trigger t1_addStk event addStk
as  print "* another trigger t1_addStk  on primitive
     event addStk :buy 1000 IBM stock for Sharma"
     insert PortFolio values("Sharma","IBM",1234,$320, getdate())
```

After parsing the code, since there is no syntax error, the code shown in Figure 6.9 is generated and sent to the Gateway Open Server.

```
/*create stored procedure*/
create procedure sentineldb.sharma.t1_addStk__Proc
as
         print "* another trigger t1_addStk  on primitive event addStk
:buy 1000 IBM stock for Sharma"
         insert PortFolio values("Sharma","IBM",1234,$320, getdate())

/* create trigger*/
create trigger sentineldb.sharma.t1_addStk
on stock
for insert
as
 insert sentineldb.sharma.stock_inserted
        select * from inserted,Version
 select syb_sendmsg("128.227.205.215", 10006, " sharmastockinsert
begin sentineldb.sharma.addStk") /* Notification */

/* Get and change occurence Number */
update SysPrimitiveEvent set vNo=vNo+1 where eventName
= "sentineldb.sharma.addStk "
delete Version insert Version select vNo from SysPrimitiveEvent where
eventName=" sentineldb.sharma.addStk"

/* action function */
execute sentineldb.sharma.t_addStk__Proc
execute sentineldb.sharma.t1_addStk__Proc
```

Figure 6.9: Code Generation in Example 6.2

6.7 Dropping a Trigger on a Primitive Event

6.7.1 Syntax of Removing a Trigger

The syntax of removing a trigger is same as that of SYBASE. Figure 6.10 shows the syntax which provides transparency for the user.

```
drop trigger trigger_name
```

Figure 6.10: Syntax of Removing a Trigger

6.7.2 Implementation Details

The client sends a drop trigger request to the ECA Agent. The command is checked in the Language Filter. Since it is a drop trigger command, it is sent to the Drop Trigger Function. The Drop Trigger Function will check if the 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" function. If the trigger is defined on a composite event, it invokes the "Drop Trigger on Composite" function. Otherwise, an error message will be sent to the user through the Gateway Open Server.

We discuss "Drop Trigger on Primitive" function in this section. "Drop Trigger on Composite" will be discussed in the next chapter.

To recall, the ECA agent performs the following actions a trigger is defined on a primitive event,:

1) Insert an item in **EventPrimitive.**
2) Create a stored procedure.
3) Create a trigger which adds a new stored procedure.

4) Insert tuples in the system tables to make the trigger persistent.

Therefore, to remove a trigger on a primitive event, we can reverse the above four steps. Also at least four additional steps need to be performed:

1) Delete an item in **EventPrimitive** list**.**
2) Delete a stored procedure.
3) Create a trigger which delete that stored procedure.
4) Delete tuples in the system tables.

After deleting this trigger and if there are no more triggers or events defined on the primitive event, the event itself should also be deleted. This includes:

1) Deleting the event item in **EventPrimitive** list**.**
2) Deleting the event details in the LED.
3) Drop a trigger.
4) Delete a tuple in the system table

Example 6.3 shows how to remove trigger t_addStk.

Example 6.3:

Assuming that we have the triggers created in example 6.1 and 6.2, the command:

drop trigger t_addStk

First, the system locates the trigger **t_addStk** defined on the primitive event **addStk**. Then the list **EventPrimitive** is scanned to find the event **addStk.** Once the event item is found, the item whose trigger name is **t_addStk** is deleted in this event item.

Second, code is generated to delete a stored procedure:

drop proc sentineldb.sharma.t_addStk__Proc

Third, code is generated to create a trigger in Figure 6.11.

Forth, code is generated to delete an item in the system table SysEcaTrigger.

delete from SysEcaTrigger

where userName = ”sharma” and triggerName= “t_addStk”

Since there is another trigger **t1_addStk** defined on primitive event **addStk,** event **addStk** cannot be deleted.

```
/* create trigger*/
create trigger sentineldb.sharma.t1_addStk
on stock
for insert
as
 insert sentineldb.sharma.stock_inserted
         select * from inserted,Version
 select syb_sendmsg("128.227.205.215", 10006, " sharmastockinsert begin
sentineldb.sharma.addStk") /* Notification */

/* Get and change occurence Number */
update SysPrimitiveEvent set vNo=vNo+1 where eventName
= “sentineldb.sharma.addStk “
delete Version insert Version select vNo from SysPrimitiveEvent where
eventName=” sentineldb.sharma.addStk”

/* action function */
execute sentineldb.sharma.t1_addStk__Proc
```

Figure 6.11: Code Generation for Drop Trigger

CHAPTER 7
IMPLEMENTATION OF COMPOSITE EVENTS

Composite events are supported in the ECA Agent. We discussed the implementation of primitive events and some basic components in chapter 6. In this chapter, we discuss –the implementation of composite events.

7.1 Syntax of Composite Events

The objective of the ECA Agent is to extend the SQL Server as well as provide user transparency. Our extensions need to be supported in the SQL Server and the users should be able to access the features through SQL extensions at the interface level. To provide a composite event mechanism for the SQL Server, the syntax of a trigger definition has been Extended.

Figure 7.1 shows the syntax of a composite event definition. The event specification language, Snoop, is used to specify composite events to the ECA Agent. The keyword "**create trigger**" is the same as that of SQL. The action function is written in SQL and is invoked within the SQL Server. The result is returned to the client.

Also, we can define triggers on an existing composite event. Actually, the syntax is the same as that of defining triggers on existing primitive events. The syntax is shown in Figure 6.2. Currently, temporal events are not supported in the ECA Agent, and will Be addressed in the future.

```
create trigger [owner.] trigger_name
event event_name [= Snoop_Event_exp]
 [coupling_ mode] [parameter_context] [priority]
as SQL_statements

operation := insert | delete | update
coupling_mode := RECENT | CHRONICLE | CONTINUOUS
                 | CUMULATIVE
parameter_context := IMMEDIATE | DEFERED | DETACHED
priority := positive integer

     Snoop_Event_exp ::= E1
                    E1 ::= E1 OR E2 | E2
                    E2 ::= E2 AND E3 | E3
                    E3 ::= E3 SEQ E4 | E4
                    E4 ::= NOT (E1, E1, E1)
                         | A (E1, E1, E1)
                         | A* (E1, E1, E1)
                         | P (E1, [time string], E1)
                         | P (E1, [time string]: parameter, E1)
                         | P* (E1, [time string], E1)
                         | P* (E1, [time string]: parameter, E1)
                         | [time string]
                         | E1 PLUS [time string]
                         | (E1)
                         | event_name
          event_name::= name
```

Figure 7.1: Syntax of Composite Event Definition

7.2 Composite Event Parser

To recall, Figure 6.3 shows that the command from the client goes through the Gateway Open Server and is then forwarded to the Language Filter. If the command is a composite event definition command, it is sent to Composite Event Parser. The Composite Event Parser parses the command, creates the event tree for the composite event in the LED and generates appropriate SQL code. To parse a composite event definition command, there are four functions involved. The workflow is shown in Figure 7.2:

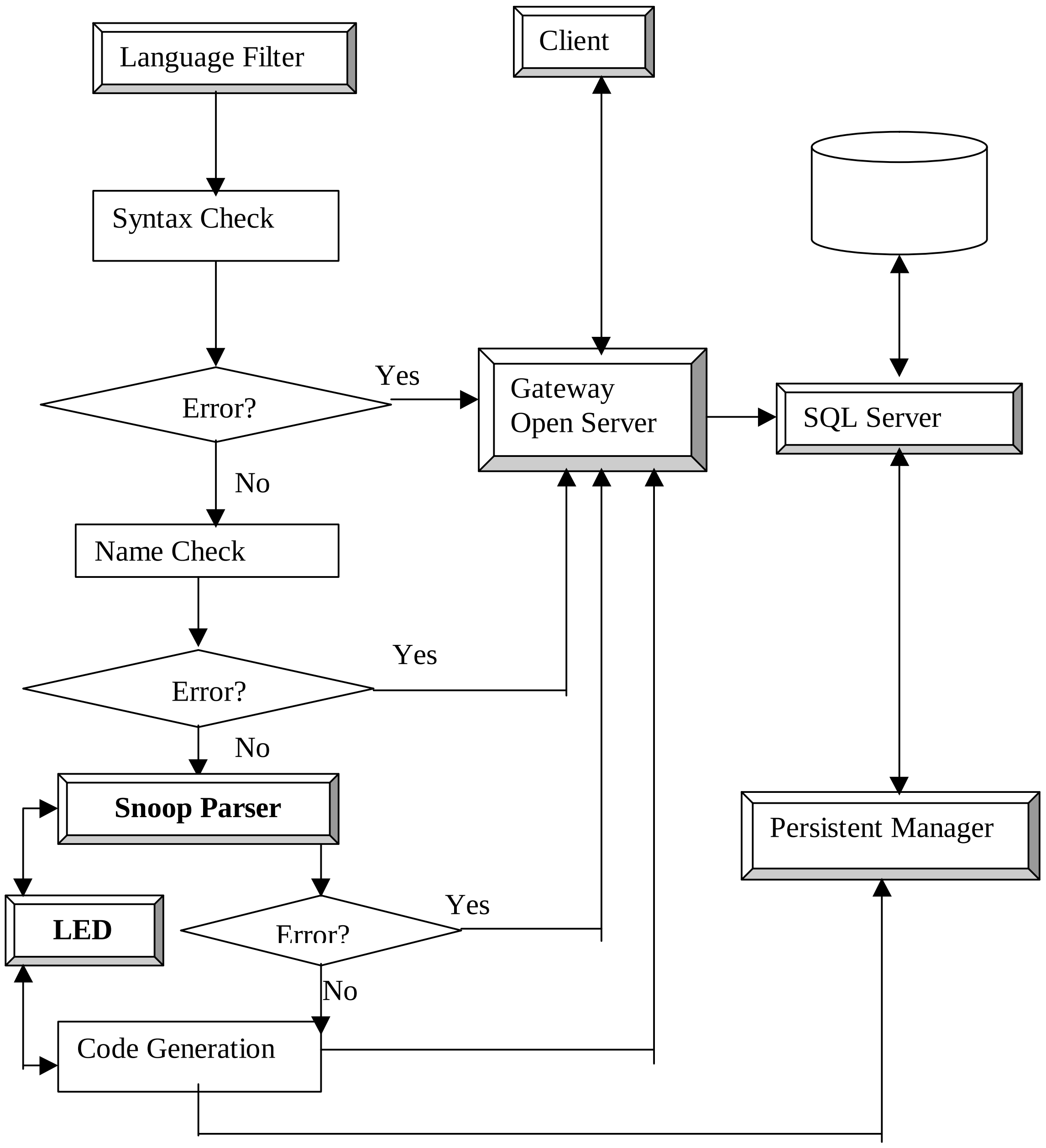


Figure 7.2: Composite Event Parser

Below, we illustrate with an example the creation of a composite event:

Example 7.1:

```
create trigger t_and
event addDel = delStk ^ addStk
RECENT
as
    print " trigger t_and on composite event addDel = delStk ^
          addStk"
   select symbol, price from stock.inserted
```

1) Syntax checking: The command is scanned to check syntax. If there is no syntax error, all object names are replaced with internal system names. Otherwise, an error message is returned to the Gateway Open Server.
   In example 7.1, since there is no syntax error, the command is changed to:

```
create trigger sentineldb.sharma.t_and
event sentineldb.sharma.addDel = sentineldb.sharma.delStk ^
                                     sentineldb.sharma.addStk
RECENT
as
    print " trigger t_and on composite event addDel = delStk ^
          addStk"
    select symbol, price from sentineldb.sharma.stock.inserted_tmp
```

2) Name checking: New object names should not be duplicates and all associated objects should exist. If these conditions are not satisfied, an error

message is returned to the Gateway Open Server. Otherwise, the event specification string is sent to the Snoop Parser.

For example 7.1, new object name ***sentineldb.sharma.t_and*** and ***sentineldb.sharma.addDel*** are not duplicates in the system and event ***sentineldb.sharma.delStk*** and ***sentineldb.sharma.addStk*** are currently defined, so string "***sentineldb.sharma.addDel* = *sentineldb.sharma.delStk* ^ *sentineldb.sharma.addStk***" is sent to the Snoop Parser.

3) Snoop Parser: Snoop Parser parses the event specification string. If there is no error in the string, it creates the composite event in the LED and generates a node in the **eventContext** list to process context.

   In example 7.1, event ***sentineldb.sharma.addDel*** is created in the LED using the constructor of the composite event operator:

   AND *sentineldb.sharma.addDel = new AND (sentineldb.sharma.delStk, sentineldb.sharma.addStk)

4) Code generation: The following code is generated:

   - A rule is created in LED. Once the rule is triggered, the action function is called, since a function can only be called in C++ in LED, function **SybaseAction** is defined as an interface of all SQL actions. In this function, the SQL action is sent to the Gateway Open Server to run the action. All information associated with the SQL action function is packed into a structure NotiStr (Figure 7.3) so that when **SybaseAction** is invoked, it can call the corresponding SQL function.

```
/* Notify Parameter Structure*/
struct NotiStr
{
   char store_proc[MAX_PARA_LENGTH];//stored procedure name
   char eventName[EVENT_NAME_LENGTH];// event name
  char context[CONTEXT_LEN];//context
   SRV_PROC      *spp; // Thread control structure for connection
}
```

Figure 7.3: Structure of NotiStr

- SQL code is generated to create a stored procedure in the SQL Server as an Action function. The code is sent to SQL Server through the Gateway Open Server.
- SQL code is generated to make the event and the rule object persistent.

In example 7.1, rule **sentineldb.sharma.t_and** is generated in the LED:

RULE * sentineldb.sharma.t_and = new RULE(sentineldb.sharma.t_and, sentineldb.sharma.addDel, condition, SybaseAction,(void *) ActionPara, RECENT);

ActionPara is a pointer of structure NotiStr:

ActionPara->store_proc = “sentineldb.sharma.t_and__Proc”;

ActionPara->eventName= “sentineldb.sharma.addDel”;

ActionPara->context=”RECENT”;

ActionPara->spp=spp;

// spp is a pointer of Thread Conrol Structure for client in Open Server

The code in Figure 7.4 is generated for action in the SQL Server and is sent to the SQL Server through the Gateway Open Server.

Also the following SQL code is generated for inserting tuples into the tables **SysCompositEvent** and **SysEcaTrigger:**

insert SysPrimitiveEvent values (“sharma”, “addDel”, “delStk ^ addStk”, getdate(), “IMMEDIATE”, “RECENT”, 0)

insert SysEcaTrigger values (“sharma”, “t_and”, “sentineldb.sharma.t_and__Proc”, getdate(), “addDel”)

The code is sent to the Persistent Manager.

```
create procedure sentineldb.sharma.t_and__Proc
as

/* context processing */
        delete sentineldb.sharma.stock_inserted_tmp
        insert sentineldb.sharma.stock_inserted_tmp
                select *
                        from sentineldb.sharma.stock_inserted, sysContext
                        where   sysContext.context=”RECENT”
                                and
                                sysContext.tableName=” sentineldb.sharma.stock”
                                and
                                sentineldb.sharma.stock_inserted.vNo=
                                 sysContext.vNo

/*action function*/
  print "trigger t_and on composite event addDel = addStk ^ delStk"
        select symbol, price from sentineldb.sharma.stock.inserted_tmp
```

Figure 7.4: Stored procedure for Example 7.1

7.3 Event Notifier

The Event Notifer is a Light Weight Thread which runs in the ECA Agent. It is generated when the ECA Agent starts. Once it is created, it will wait for notifications from SQL Server. If Event Notifier receives a notification from the SQL Server, it notifies the LED that an event has occurred.

7.3.1 server.config

A UDP socket is used to connect to SQL Server. The IP address and port number in the ECA Agent should match that in the SQL Server. The file **server.config** is used to store the IP address and port number of Event Notifer. Upon code generation, the Primitive Event Parser retrieves the address from **server.config** file and generates code to notify Event Notifier. In fact, the SQL Server will create a UDP socket to connect with the Event Notifier based upon the SQL code. An example of file server.config is shown in Figure 7.5:

```
128.227.205.215 10006

# IP address :eclipse.ufl.edu

# port_number :10006
```

Figure 7.5 An example of server.config

Also, when the ECA Agent is started, Event Notifier selects the IP address and the port number from the server.config file and generates a UDP socket for message passing from SQL Server.

7.3.2 Implementation of Event Notifier

There are two parts to the Event Notifier. The first is the Notification Listener which catches any notification from the SQL Server. The other is the Notifier which sends notifications to LED. Figure 7.6 shows the details of the Event Notifier.

Primitive Event Parser adds a built-in function call in the trigger definition. The built-in function uses the UDP protocol to send the message to the destination. When a primitive event occurs, a trigger is invoked in the SQL Server. The UDP socket is created and the message is sent to the Notification Listener in the Event Notifier. As soon as the Notification Listener receives notification, it calls the Event Notifer. At this point the Event Notifier unpacks the message and sends the notification to LED. After LED detects an event, it invokes an action interface "**SybaseAction**" which calls an action function (a stored procedure) stored in the SQL Server through the Gateway Open Server.

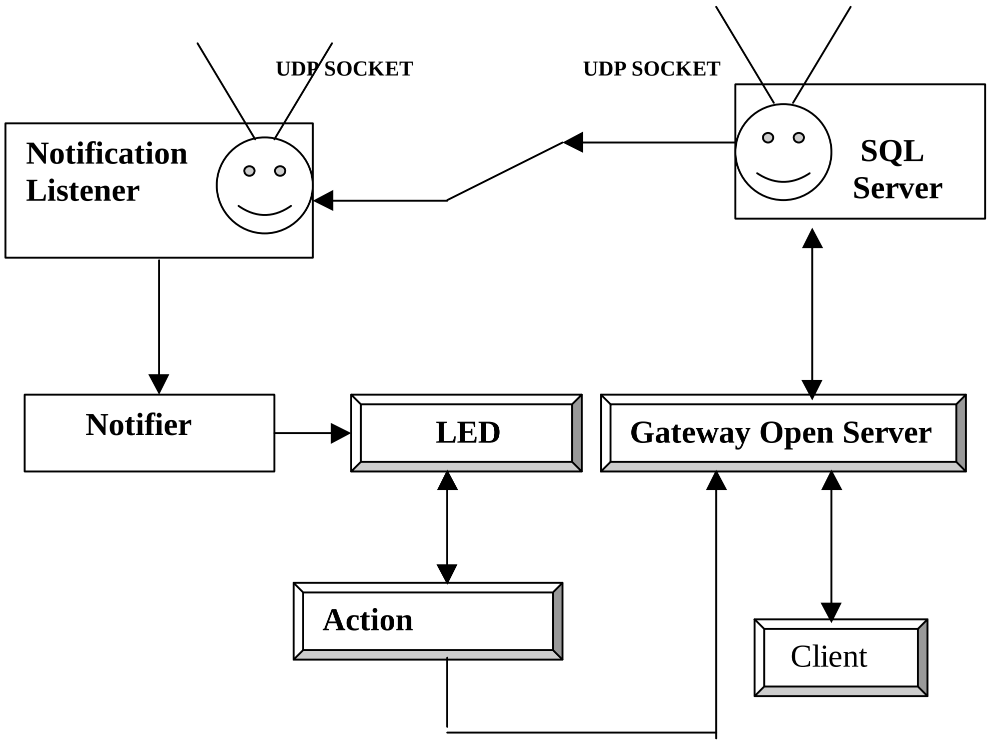


Figure 7.6 workflow of Event Notifier

7.4 Action Handler

Action Handler actually is a function in the ECA Agent. From the LED's point of view, it is a C++ function. From the point view of the SQL Server, it is an interface for an action in the SQL Server. Since many events may occur at the same time, and each action function should run independently, a new thread is generated for each call to **SybaseAction**. Each thread calls an action function within the SQL Server (stored procedure) through the Gateway Open Server. The final results are returned to the client through the Gateway Open Server. Figure 7.7 shows how Action Handler works.

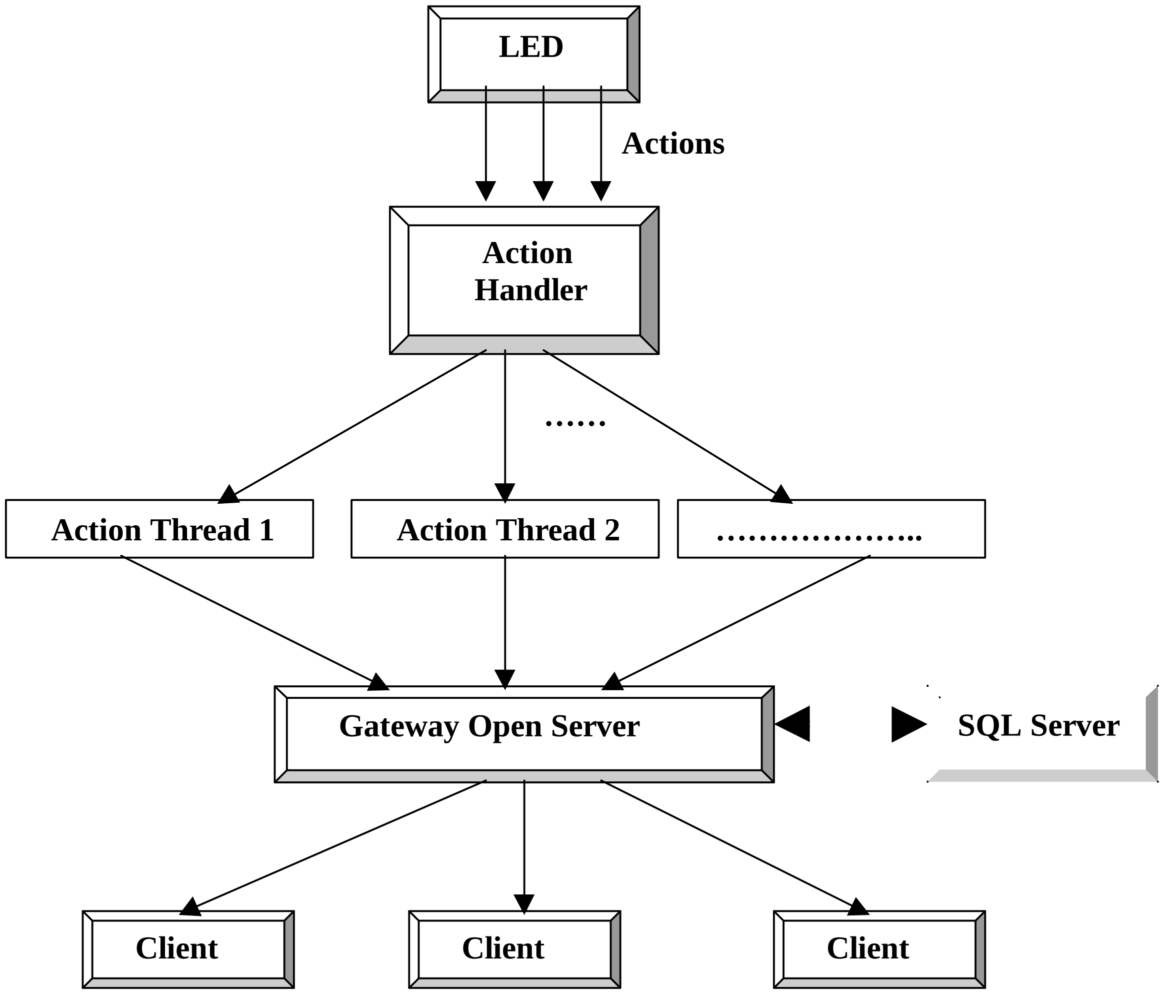


Figure 7.7 Action Handler

The interface of Action Handler is defined as following:

void SybaseAction(L_OF_L_LIST *n1_list, void *actionproc)

void *actionproc is a pointer of structure NotiStr. When SybaseAction is called by LED, SybaseAction generates a thread. In the thread, it unpacks the *actionproc and gets the stored procedure name and Thread Control structure which will be used to connect to the SQL Server in the Gateway Open Server.

L_OF_L_LIST *n1_list is a linked list which is defijned in the LED for keeping track of the parameters of constituent events for the parameter context specified.

7.5 Parameter Context

The parameter contexts supported in the ECA agent are: *recent, continuous*, *cumulative* and *chronicle* [vk94, dke94, CHA94].

7.5.1 Syntax of getting context

To get the context of a table, a user should use the following syntax:

TableName.inserted or

TableName.deleted

The Composite Event Parser will turn it into an internal system name:

dbname.username.TableName.inserted_tmp or

dbname.username.TableName.deleted_tmp

In example 7.2, if the user wants the paramater context “CUMULATIVE” semantics when trigger t1_and is invoked, it is specified as follows:

Example 7.2:

create trigger t_com

event ComEve = addDel ; buyStk; (delStk ^ selStk)

CUMULATIVE

as

select symbol, price from stock.deleted

select * from PortFolio.deleted

7.5.2 System table for parameter context

System table sysContext is created to store the occurrence of the tables defined on

Sharma: please explain te occurrences of the tables clearly

certain events. The structure of table sysContext is shown in Figure 7.8.

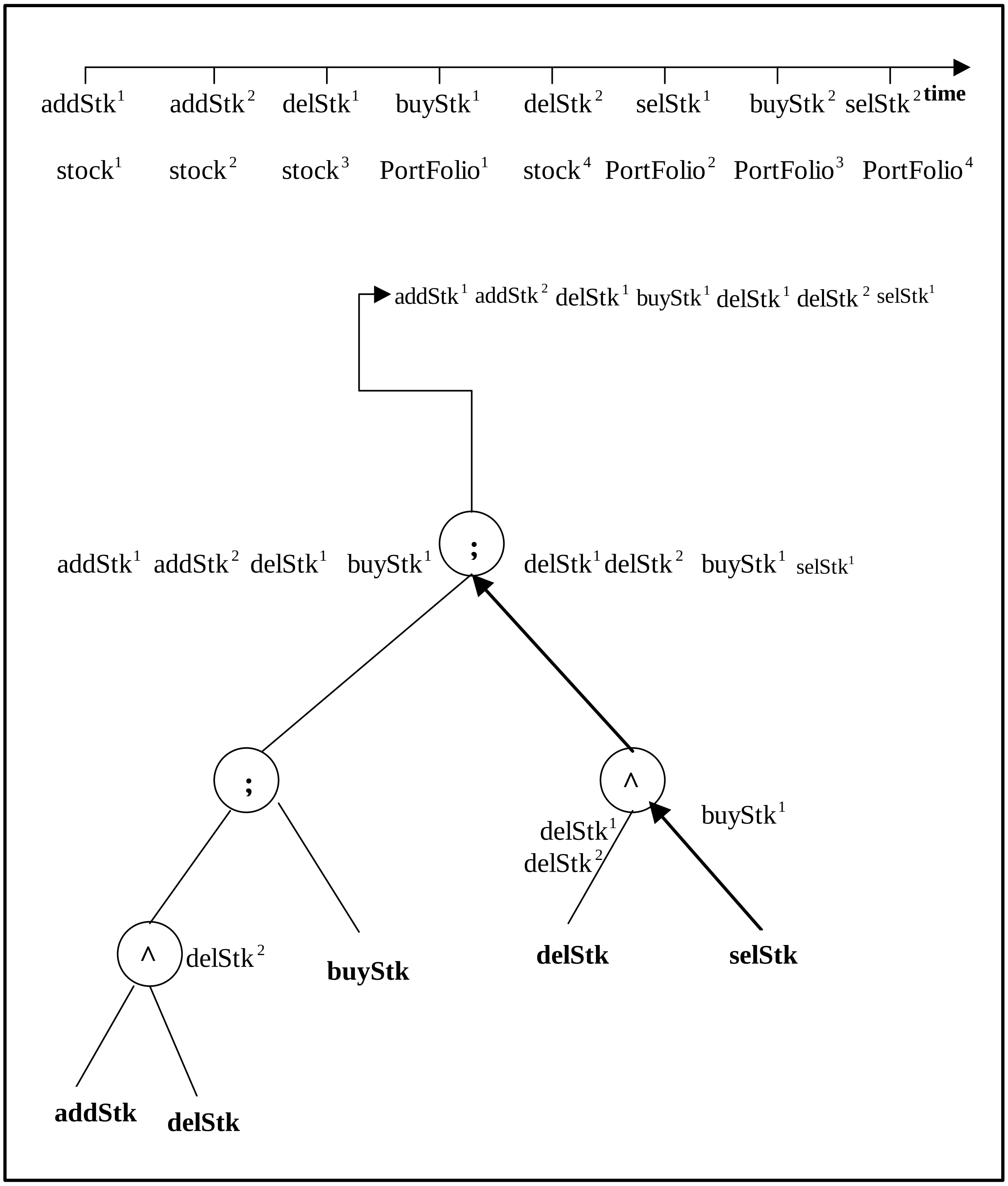
Column_name	Type	Length	Nulls
tableName	varchar	50	not null
context	varchar	12	not null
vNo	int	4	not null

Figure 7.8 Structure of table sysContext

Tuples are inserted into sysContext when the action is invoked. Each tuple is associated with an occurrence of a table. Since for one table there may be many occurrences in a composite event which is dependent on the parameter context, there are many tuples for the same table on the same parameter context. A list is generated after event detection. The list is derived from LED, changed into the list that records table occurrences, then SQL code is generated to insert tuples into sysContext table. The old tuples whose **tableName** and **context** is the same as that of new tuples are deleted before inserting new tuples.

Sharma: is not the above true only for the recent context?

Figure 7.10 shows the tuples that are inserted into sysContext in example 7.2, which depends on the event detection in Figure 7.9. SQL code generated is shown in Figure 7.10.



addStk1 addStk2 delStk1 buyStk1 delStk2 selStk1 buyStk2 selStk2 time
stock1 stock2 stock3 PortFolio1 stock4 PortFolio2 PortFolio3 PortFolio4
addStk1 addStk2 delStk1 buyStk1 delStk1 delStk2 selStk1
addStk1 addStk2 delStk1 buyStk1
;
delStk1 delStk2 buyStk1 selStk1
;
^
buyStk1
delStk1
delStk2
^
delStk2
buyStk
delStk
selStk
addStk
delStk

Figure 7.9: Event detection in example 7.2

tableName	context	vNo
sentineldb.sharma.stock	CUMULATIVE	1
sentineldb.sharma.stock	CUMULATIVE	2
sentineldb.sharma.stock	CUMULATIVE	3
sentineldb.sharma.PortFolio	CUMULATIVE	1
sentineldb.sharma.stock	CUMULATIVE	4
sentineldb.sharma.PortFolio	CUMULATIVE	2

Figure 7.10: Tuples inserted in sysContext

```
/* delete old tuples */
if exists
  (select * from sysContext where
          tableName= “sentineldb.sharma.stock “ and
          context = “CUMULATIVE”)
   delete from sysContext where
          tableName= “sentineldb.sharma.stock “ and
          context = “CUMULATIVE”)
if exists
  (select * from sysContext where
          tableName= “sentineldb.sharma.PortFolio”  and
          context = “CUMULATIVE”)
   delete from sysContext where
          tableName= “sentineldb.sharma.PortFolio”  and
          context = “CUMULATIVE”)
/* Insert tuples */
insert into sysContext values
        (“sentineldb.sharma.stock”,“CUMULATIVE”, 1)
insert into sysContext values
        (“sentineldb.sharma.stock”,“CUMULATIVE”, 2)
insert into sysContext values
        (“sentineldb.sharma.stock”,“CUMULATIVE”, 3)
insert into sysContext values
        (“sentineldb.sharma.PortFolio”,“CUMULATIVE”, 3)
insert into sysContext values
        (“sentineldb.sharma.stock”,“CUMULATIVE”, 4)
insert into sysContext values
        (“sentineldb.sharma.PortFolio”,“CUMULATIVE”, 2)
```

Figure 7.11: SQL code generated for inserting

7.5.3 Implementation of parameter context

All occurrences of a table associated context are kept in the system table sysContext. There are four steps to implement parameter context no matter what the parameter context is:

1) When a primitive event occurs, the occurrence of associated table is put into table **dbname.username.TableName.inserted** or **dbname.username.TableName. deleted.**
2) Retrieve the parameter context list from LED and generate the SQL code for inserting tuples.
3) Insert tuples into **sysContext** through the Gateway Open Server.
4) Join **sysContext** and **dbname.username.TableName.inserted** (or **dbname.username.TableName.deleted** to get the context.

Figure 7.12 shows the stored procedure in example 7.2

```
create procedure sentineldb.sharma.t1_com
as

/* for context processing*/
        /* context processing for table stock */
        delete sentineldb.sharma.stock_deleted_tmp
        insert sentineldb.sharma.stock_deleted_tmp
               select *
                       from sentineldb.sharma.stock_deleted, sysContext
                       where   sysContext.context=" CUMULATIVE"
                               and
                               sysContext.tableName="stock"
                               and
                               sentineldb.sharma.stock_deleted.vNo=
                                sysContext.vNo

        /* context processing for table PortFolio */
        delete sentineldb.sharma.PortFolio_deleted_tmp
        insert sentineldb.sharma.PortFolio_deleted_tmp
               select *
                       from sentineldb.sharma.PortFolio_deleted, sysContext
                       where   sysContext.context=" CUMULATIVE"
                               and
                               sysContext.tableName="PortFolio"
                               and
                               sentineldb.sharma.PortFolio_deleted.vNo=
                                sysContext.vNo


/*action function*/
select symbol, price from sentineldb.sharma.stock.deleted_tmp
 select * from sentineldb.sharma.PortFolio.deleted_tmp
```

Figure 7.12: Stored procedure generated for example 7.2

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

We have demonstrated in this thesis that by introducing and ECA Agent outside the SQL Server, full-fledged active functionality can be supported as a value added capability. Nearly the full range of active functionality can be supported without resorting to an integrated active database architecture. The ECA Agent makes the Sybase SQL Server more powerful and provides transparent active capability to database clients.

Although this thesis describes an architecture, design, and implementation of an ECA Agent for Sybase, we believe that the approach developed in thesis is general and can be used for any relational DBMS.The functionality includes:

1) Transparent interface for users to create primitive and composite events.
2) Persist user created events using the native DBMS capability.
3) Drop triggers and events as desired.
4) Detect primitive and composite events.
5) Invoke actions within the SQL Server.
6) Support for multiple parameter contexts.
7) Collecting and passing parameters to conditions and actions.

Considering the recent proliferation of commercial relational systems that supports primitive trigger capability, the prospect of seamlessly integrating an active component is very attractive indeed. For many users, this may be the only practical way to use production rules today. Additionally, because ECA Agent is external to Sybase, its power is not limited to what the database can provide, but by what the architecture supports.

8.2 Future Work

We plan on extending the current implementation with the following additional functionality:

- The current implementation supports immediate coupling mode. We plan on extending this with detached and deferred coupling
- We plan on supporting heterogeneous distributed active capability by using the extended LED (ELED) that interfaces with the global event detector (GED). We plan on doing this both in CORBA and RPC/Socket
- Optimization of the various components of the ECA Agent. Since the communication between ECA Agent and SQL Server is based on the socket, security and system efficiency will be affected. We plan on decreasing communication times to increase system efficiency and make the system more secure.
- Support active database semantics in the other RDBMS such as Informix, Oracle by using the idea of an ECA Agent.

LIST OF REFERENCES

[ACT96] ACT-NET Consortium (1996). The Active Database Management System Manifesto: A Rulebase of ADBMS Features. ACM SIGMOD Record, 25(3):40--49.

[BAD93] Badani, Rajesh. Nested Transactions for Concurrent Execution of Rules Design and Implemention. Master’s thesis, University of Florida, Gainesville, 1993.

[BER91] Berndtsson, M. ACOOD: an Approach to an Active Object Oriented DBMS. Master’s thesis, University of Skovde, September 1991.

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

[BER94] Berndtsson, M. (1994). 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

[CER96] Ceri, S., Fraternali, P., Paraboschi, S., and Branca, L. (1996). Active Rule Management in Chimera. In ActiveDatabase Systems - Triggers and Rules For Advanced Database Processing, chapter 6, pages 151--176. Morgan Kaufman.

[CHA89] Chakravarthy, S. (1989). Rule Management and Evaluation: An Active DBMS Perspective. ACM SIGMOD Record, 18(3):20--28.

[CHA90] Chakravarthy, S. and Nesson, S. (1990). Making an Object-Oriented DBMS Active: Design, Implementation and Evaluation of a Prototype. In Bancilhon, F., Thanos, C., and Tsichritzis, D., editors, Advances in DatabaseTechnology - EDBT'90. International Conference on Extending Database Technology, volume 416 of Lecture Notes in Computer Science, pages 393--406. Springer.

[CHA94] Chakravarthy, S., Anwar, E., Maugis, L., and Mishra, D. (1994). Design of Sentinel: An Object-Oriented DBMS with Event-Based Rules. Information and Software Technology, 36(9):559--568.

[CHA95] Chakravarthy, S., Krishnaprasad, V., Tamizuddin, Z., and Badani, R. (1995a). ECA Rule Integration into an OODBMS: Architecture and Implementation. In Proceedings of the 11th International Conference on Data Science, pages 341--348.

[CHA98] Chakravarthy, S., Le, R. (1998). ECA Rule Support for Distributed Heterogeneous Environments. In Proceedings of the 14th International Conference on Data Science, page 601.

[CHK94] Chakravarty, S., Krishnaprasad, V., Anwar, E., and Kim, S. K. (1994). Composite Events for Active Databases: Semantics Contexts and Detection. In Proceedings of the 20th International Conference on Very Large Data Bases, pages 606--617.

[CHM94] Chakravarthy, S. and Mishra, D. (1994). Snoop: An Expressive Event Specification Language for Active Databases. Data and Knowledge Science, 14(1):1--26.

[DAY88] Dayal, U., Blaustein, B., A. Buchmann, S. C., and et al. (1988a). The HiPAC project: Combining active databases and timing constraints. ACM SIGMOD Record, 17(1):51--70.

[DAY94] U.Dayal, E. N. Hanson, and J. Wisdom. Active Databasebase Systems. In Modern Database Systems: The Object Model, Interoperability, and BeyondAddison-Wesley, Reading, Massachusetts, 1994

[GEH92] Gehani, N., Jagadish, H. V., and Smueli, O. (1992b). Event specification in an active object-oriented database. InProceedings of the 1992 ACM SIGMOD International Conference on Management of Data, pages 81--90.

[HAN89] Hanson, E. N. (1989). An Initial Report on the Design of Ariel: A DBMS With an Integrated Production Rule System. ACM SIGMOD Record, 18(3):12--19.

[HAN92] Hanson, E. N. (1992). Rule Condition Testing and Action Execution in Ariel. In Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, pages 49--58.

[HAN93] Hanson, E. N. and Widom, J. (1993). An Overview of Production Rules in Database Systems. The Knowledge Science Review, 8(2):121--143.

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

[LIA97] Liao, H. Global Events in Sentinel: Design and Implementation of a Global Event Detector. Master's thesis, University of Florida, Gainesville, 1997.

[SAY96] Saygin, Y., Ulusoy, O. and Chakravarthy, S., (1996). Implementation of Parallel Nested Transactions for Nested Rule Execution in Active Databases.

[SHY91] Shyy, Yuh-Ming and Su, Stanley Y. W., "K: A High-Level Knowledge Base Programming Language forAdvanced Database Applications," SIGMOD '91, ACM, Denver, CO., May 29-31, 1991, pp. 338-347.

[STO87] Stonebraker, M., Hanson, E., and Hong, C. H. (1987). The Design of the Postgres Rule System. In Proceedings of the 3rd International IEEE Conference on Data Science, pages 365--374.

[STO88] Stonebraker, M., Hanson, M., and Potamianos, S. (1988). The POSTGRES rule manager. IEEE Transactions on Software Science, 14(7):897--907.

[STO92] Stonebraker, M. (1992). The Integration of Rule Systems and Database Systems. IEEE Transactions on Knowledge and Data Science, 4(5):415--423.

[SU93] Su, S. Y. W. and Chen, H.-H. M. (1993). Temporal Rule Specification and Management in Object-Oriented Knowledge Bases. In Proceedings of the 1st International Workshop on Rules in Database Systems, Workshops in Computing, pages 73--91. Springer.

[SU91] Su, Stanley Y. W. and Park, Jong H., "An Integrated System for Knowledge Sharing among Heterogeneous Knowledge Derivation Systems," International Journal of Applied Intelligence, Vol. 1, 1991, pp. 223-245.

[SYB96] Sybase. Sybase SQL Reference Manual: Volume 1. Sybase, Inc., 1996.

[VAN96] Vance, D. Supporting Active Database Semantics in Sybase. Master's thesis, University of Florida, Gainesville, 1996.

[WID94] Widom, J. and Chakravarthy, S., editors (1994). Proceedings of the 4th International Workshop on Research Issues in Data Science - Active Database Systems. IEEE-CS. ISBN 0-8186-5360-4.

[WID96] Widom, J. (1996). The Starburst Active Database Rule System. IEEE Transactions on Knowledge and Data Science, 8(4): 583--595.

BIOGRAPHICAL SKETCH

Lijuan Li was born on January 19, 1968, at Kunming, China. She received her Master of Science in Computer Science from Yunnan University, Kunming, China, in 1993, and her Bachelor of Science degree in Computer Science from Yunnan University, Kunming, China, in 1990. She expects to receive his Master of Science degree in computer and information science and engineering from the University of Florida, Gainesville, Florida in May 1998. She has been working as a software engineer since receiving her MS degree in 1993. Her current research interests include integrating databases with multi-tiered architectures, active and object-oriented databases.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Sharma Chakravarthy, Chairman
Associate Professor of Computer and
Information Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Stanley Y. W Su
Professor of Computer and Information
Science and Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Eric N. Hanson
Assistant Professor of Computer and
Information Science and Engineering

This thesis was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Master of Science.

August 1998

Winfred M. Phillips
Dean, College of Engineering

Karen A. Holbrook
Dean, Graduate School