SUPPORT FOR COMPOSITE EVENTS AND RULES IN DISTRIBUTED HETEROGENEOUS ENVIRONMENTS

By

ROGER LE

A THESIS 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 family

ACKNOWLEDGMENTS

Firstly and foremost, I would like to express my sincere gratitude to my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this interesting topic and for providing me great guidance and support through the course if this research work. I am also grateful to Dr. Joachim Hammer and Dr. Eric Hanson, for graciously agreeing to serve on my committee, for teaching wonderful classes, which have greatly helped in the 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 for maintaining a well-administered research environment and being so helpful in times of need. Special thanks to Federico Zoufaly, Shiby Thomas, Hyoungjin Kim, and Seokwon 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.

Last, but not least, I would like to take this opportunity to thank my family for their endless love. My parents, my brother Francis and most of all, Hui, give me unlimited amounts of patience, support, and extraordinary encouragement to get through difficult moments.

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

SUPPORT FOR COMPOSITE EVENTS AND RULES IN DISTRIBUTED HETEROGENEOUS ENVIRONMENTS

By

Roger Le

December 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 the active database system are significant. Event-condition-action (ECA) rules are used to capture the active capability. As an example, the same capability can be used to support push/pull propagation of data in a distributed environment. As another example, workflow and E-commerce applications that are event-driven can be supported by extending the ECA rules concept to heterogeneous environments.

The utility and functionality of active capability (ECA rules) has been well established in the context of databases. Today, most of the commercial relational databases management systems (RDBMSs) have some form of ECA rule capability. In

addition, there are several research prototypes that have extended the ECA rule capability to object-oriented database management systems (OODBMSs).

In order for the active capability to be useful for a large class of advanced applications, it is necessary to go beyond what has been proposed/developed in the context of databases. Specifically, extensions beyond the current state of the art in active capability are needed along several dimensions: i) make the active capability available for non-database applications, in addition to database applications; ii) make the active capability available in a distributed environments; that is, in addition to specifying ECA rules within a system, it should be possible to specify them across applications; and iii) make the active capability available for heterogeneous sources of events (whether they are databases or not).

The objective of this thesis is to provide the best architecture and framework to support ECA rules that can run across distributed and heterogeneous systems. The design allows the user to specify rule condition and action at run-time, and will integrate a mechanism for composite event detection based on an event tree. As we go along with the design and implementation, the different alternatives will be discussed.

CHAPTER 1
INTRODUCTION

The utility and functionality of active capability (event-condition-action or ECA rules) has been well established in the context of databases. Today, most of the commercial relational databases management systems (RDBMSs) have some form of ECA rule capability. In addition, there are several research prototypes that have extended the ECA rule capability to object-oriented database management systems (OODBMSs). **Sentinel**, developed at the University of Florida ([ANW93], [BAD93], [CHA94a], [CHA94b], [CHA95]) is one such prototype that supports an expressive composite event specification language (termed Snoop), efficient event detection (by using pre-processor generated wrappers), conditions and actions (as a combination of OQL and C++), multiple and cascaded rule processing (using a rule scheduler and nested transaction model), a visualization tool, and a rule editor for dynamic creation and management of rules. Some of the above results will be relevant for object-relational DBMSs that are currently being developed by the industry.

Although the ECA rule concept was developed in HiPAC ([CHA89]) for providing a uniform framework for supporting many *ad hoc* functions (such as integrity constraints, separation of rules/policies from application code, access control, incremental view management) in the context of databases, their utility seems to be more universal than envisioned by its developers. While some types of applications can run in stand-alone mode, other applications need to react to external events in order to resume or even start their execution. Event propagation from a source to a consumer application can be used for many purposes in a distributed environment. For example, this capability is relevant in terms of propagation of changes for heterogeneous sources in the context of data warehousing. In that case, the frequency of data transmission can be very high so that you may want to reduce it by using composite events to filter the flow of data. That means support of push/pull propagation of data in a distributed environment has never been more relevant. As another example, workflow and E-commerce applications that are event-driven can be supported by extending the ECA rules concept to heterogeneous environments.

Scalability and high performance are facilitated by the fact that events are detected and rules processed asynchronously, separately from the initial updates to the data. When rules are executed locally to the events that trigger them, it may make sense to perform synchronous processing, but especially when it comes to separating the event detection and rule execution in a distributed environment, asynchronous processing allows faster testing of sophisticated conditions without slowing down the updates.

Scalability is an important factor for consideration when designing software, and the distributed nature of an application can be accounted for it. Because of the demand for distributed computing capabilities generated by the Web, an event detection system seems to be truly scalable if it can be hooked to the Internet and is able to handle hundreds or even thousands of event notifications and rule executions.

Making the applications work in a distributed fashion can increase the fault-tolerance of the whole system. When the latter does not perform properly anymore, it is easier to locate and replace the faulty component by another available one that performs identical tasks, in the case of separated functional modules instead of a single conglomerated application. So, for example, if a machine breaks down in a production line, the manufacturer can replace it with another one before the whole production is affected. When your application is distributed, replacing a component instead of the whole system is less expensive, and decreases the risk of having a single point of failure.

The performance of the system is also affected by whether it is distributed or centralized in one application. Processing the entire amount of data in one place should be done if the processing time is short, otherwise it could be faster to broadcast the data across multiple machines for distributed computation and gather the results back to the source. There is obviously a trade-off between the data transport overhead and the speed-up gained by distributing the computing. The best solution is to run applications on multi-processor machines, but not everybody can afford to replace their existing hardware. That is the reason why distributing applications on available machine can increase performance with scalability. To refer again to the production line, the throughput is improved when multiple machines in line constitute a pipeline better than a single equivalent machine that is doing all the work.

In order for the active capability to be useful for a large class of advanced applications, it is necessary to go beyond what has been proposed/developed in the context of databases. Specifically, the extensions beyond the current state-of-the-art in active capability are needed along several dimensions:

1) Make the active capability available in a distributed environment; that is, in addition to specifying ECA rules within a system, it should be possible to specify them across applications, and associate them with any of the various

available sources of information. That will constitute a framework for distributed computing by taking advantage of all the resources available on the network and alleviating the workload on a particular machine. The ECA rules execution can be brought to another level of distributed computing. In fact, there can be an advantage of evaluating the rule condition by one application and executing the corresponding action by another application. Since one of our concerns is to take advantage all the available distributed resources, it would be very cost-efficient to reuse legacy implementation to be used for the rule conditions and actions. Categorizing applications as containers of available conditions or actions to be made available to the users may be the right approach for component reusability. For example, a database can be used to store conditions in the form of structured query language (SQL) statements that can be used to trigger as many rules as the user wants, and the same concept can be applied for a library of rules actions.

2) Make the active capability available for non-database applications, in addition to database applications. The system should support connections to heterogeneous data sources, including general application programs, any types of databases, Web engines producing hypertext markup language (HTML) information and legacy systems.

3) Support specification of events and rules *dynamically*. The dependencies between data exchange as well as execution of methods/procedures may need to be established for the overall operation of the distributed application. For example, if the availability of a critical component is a problem, that

information needs to be sent to the designers to substitute an alternative *available* component so that the production line does not idle affecting the product shipment. If event-based rules can be specified across applications in a dynamic manner, the above can be specified and handled without having to change existing systems. The same is true in large enterprises having heterogeneous systems that need to coordinate and cooperate together for the overall functioning of the enterprise.

Going to a distributed architecture raises a number of interesting issues, and also makes the life easier for the user in several ways:

1. Usability has been improved because dynamic specification and rule execution are well supported. The capability to change a rule specification on the fly is much more flexible than having to recompile the rule implementation.
2. Application and rule specifications are simplified provided that the conditions and actions are already implemented. Consequently, managing the rules becomes easy.

1.1 Related Work

There has been some work in the detection of events in a distributed environment ([SCH96], [JAE97], [LIA97], [SU95], [SU96], [LAM97]). In Schwiderski [SCH96], the main emphasis is on the detection of events and the problems associated with it due to clock synchronization and communication (delays in delivery of events) problems. In Jaeger [JAE97], the emphasis is on processing a global event history that is gathered from individual event histories propagated by participants of a loosely coupled distributed environment.

1.1.1 Sentinel

In Liao [LIA97], a global event detector (GED) has been developed as a server essentially to provide support for rules using events (both primitive and composite) generated in other applications. The global event detector provides asynchronous event notification to its clients as well as propagates parameters of the events (primitive or composite) for use in condition and action evaluated in the client application address space. This functionality is satisfying in a once-for-all-defined environment. Since the ECA rules are hard-coded in the client programs, they cannot be modified at run-time, unless they are recompiled.

Another limitation in the Sentinel design is that it does not address the problem of how heterogeneous sources can be integrated into the system. The field of interest constituted by data warehousing encourages us to look further at the *extensity* of our active system, by including heterogeneous active or non-active systems to be integrated in a global event detection and rule execution system. For example, a RDBMS like Sybase can benefit from our system to propagate changes of tables to subscriber applications.

Remote procedure call (RPC) has been used to implement the communication protocol between the GED server and the client applications. But going to a full-fledged distributed environment support has several advantages, compared to the mere use of RPC and sockets:

1. Improve and simplify the communication between clients and GED server, by reducing the overhead inherent in the use of RPC communication style, especially in the case of passing complex information, like object instances (for example, *event graph* and list of event parameters [LIA97]).
2. Make the communication between clients and server more robust. When an application happens to be disconnected from its server, it should be aware of the cause (the server is down for example) and reconnect for subsequent requests in a transparent way if possible.
3. Be able to handle special cases like the possibility of running two client applications having the same name on the same machine.

We will reuse the module for composite event detection based on an event graph, which was the cornerstone of the GED server. While the event management system (subscription and notification) remains the same in essence, the communication protocol originally using RPC will be adapted for a distributed component-based environment [LIA97].

The processing model for the ECA rules is also changed in order to accommodate remote execution of conditions and actions.

1.1.2 TriggerMan

There are other attempts in using active capability in a distributed environment. TriggerMan [HAN97], for example, accepts and processes rules in a separate address space (TriggerMan server) that is connected to a number of information sources. However, TriggerMan does not support composite events. Beside, there is no support for rule execution at the client application side, as opposed to the Sentinel approach, although it is possible to propagate the events to the event consumer.

1.1.3 Component-Based Softwares

Apart from the above work related to active capability, a number of efforts in the commercial world have been addressing support for distributed components, notably Common object request broker architecture (CORBA) and object linking and embedding/distributed component object model (OLE/DCOM). Given that future distributed environments are likely to use these two component-based systems, it is imperative that we address the availability of services, such as composite events and rules, for these environments and provide support for them in a pragmatic framework.

1.2 Problem Statement

Based on our experience in developing active capability in **Sentinel** and for distributed database environments, we believe that the capability can be generalized along the three dimensions mentioned above, and supported using a component-based framework.

The general problem is to support event/rule specification dynamically, and their detection/execution for any number of systems. Our focus is on the specification, detection, and management of composite events as this aspect has not been addressed in the literature and we believe is important for a large class of real-life applications. We are also investigating rule processing in a distributed environment, where we also allow a consumer application to supply events at the same time.

Another issue raised by the distributed environment is the visibility at a global scope of all the components relevant to the event and rule specifications, including definition of the available primitive and composite events, the list of the client applications likely to provide actions and conditions for ECA rules, and the list of user-defined rules. From a user point of view, a tool for browsing the available events for example becomes anything but useless as the number of components to be managed increases.

In the following chapters, we will address how we are planning on addressing the above extensions using a combination of existing components and new functionality/services that are derived from our experience in designing and implementing Sentinel.

CHAPTER 2
ALTERNATIVES FOR INFRASTRUCTURE

In this chapter, we are presenting the alternatives for infrastructure. We have already seen that the traditional use of RPC/sockets has some drawbacks and limitations, and suffers from complexity in usage and development of servers and applications, when we presented the work of GED. The advantages/gains of going to a distributed environment support that handles objects which RPC cannot lead us to explore the features of newer infrastructures for the development of distributed applications such as distributed computing environment (DCE), CORBA, and DCOM, relevant to our project, and to discuss their limitations if any.

Because of the importance of bringing heterogeneous systems together, the choice for a framework for developing object-oriented, distributed applications will be certainly determined by the capability of cross-platform support, that is avoiding dependencies on the peculiarities of any one platform.

Another factor to be considered is cross-language support. While C and C++ are now used for a significant volume of software development, COBOL is still the most-used programming language (as the primary language of an estimated 3 million programmers, compared to 1.6 million using Visual Basic and 1.1 million using C and C++).

Nowadays, a system should provide some kind of connectivity with the World Wide Web to be considered viable. The importance of the Internet has already been stressed enough when it comes to use it as a general-purpose user interface. But there is demand for it to be associated with more distributed computing capabilities, and a major strength of the Web technology is its platform independence.

Scalability and high performance have fostered the need of distributing object computing. Thus, it is logical to think that such a framework should provide support to design software scalable across large networks, if not the Internet and its millions of online users.

Finally, all distributed computing necessarily involves communication. If this takes place over public computing networks then the authenticity of the data and its integrity while being transferred may be at risk. That is the reason why the security issue will be brought up into the discussion.

2.1 CORBA

The Object Management Group (OMG), a consortium of industry companies has been created in 1989 to share one consistent vision of an architecture for distributed, component-based object computing. The architecture is described in the Object Management Architecture Guide, first published in 1990, and has been incrementally populated with the specifications of the core inter-object communication component CORBA [OMG97].

Multiple platform support has always been OMG's highest priority, because it tries to avoid dependencies on the peculiarities of any one platform. Ironically, they even support a wider range of Microsoft platforms: CORBA-compliant products are available on MS-DOS and 16-bit Windows 3.x in addition to the 32-bit Microsoft platforms, almost every Unix, OS/2, OS/400, MacOS, VME, MVS, VMS, and a number of real-time operating systems.

CORBA’s language-neutral approach was designed to accommodate a lot of programming languages, including the most used, COBOL. OMG has adopted mappings for C, C++, Ada, Smalltalk, COBOL and Java, as well as FORTRAN.

Platform independence across heterogeneous systems and the Internet has been achieved with Java technology based on bytecodes that can be interpreted on every Java Virtual Machine, which in turn can communicate with the CORBA interface to access other distributed computing resources not available otherwise. This CORBA access can be provided via an “ORBlet”, itself written in Java, and downloadable into the browser. Such ORBlets are already available from many vendors including Sun, IONA, HP, Oracle, IBM and Visigenic. Furthermore, starting with Netscape Navigator 4.0, Java-enabled Web browsers will soon be available with CORBA support built-in, thus removing even the need to download the ORB into the browser.

CORBA was designed from the start with Internet-scale applications in mind. It supplies applications built on the Object Management Architecture (OMA) with a robust backbone for interoperability in local- and wide-area network environments. Though it works as well as within process and machine boundaries, support for interoperability across network boundaries was the primary design center from the very start. Moreover, CORBA provides interoperability over the Internet with its IIOP (Internet interoperability protocol) protocol.

CORBA specification provides services for secure communication with security safeguards to be usable in the real world of public networks, such as the Internet.

OMG has specified a wide range of security service for CORBA-based systems, which not only provides confidentiality and authentication, but also implements non-repudiation (making it usable for financial transactions, to ensure that the participants cannot later deny their commitments).

For the communication over the Internet, it seems that the state-of-the-art security model resembles more the approach of Java applet execution. This is because all Java applets run on a virtual machine which insulates them from direct contact with the host system. This so-called “sandbox” around the applet enforces restrictions that prevent it from interfering with the host.

Hence the OMG security service specifications address the real-world security issues necessarily for the use of distributed objects in building systems for electronic commerce, in addition to the confidentiality needed in other applications such as keeping medical records.

2.2 DCOM/ActiveX

The Windows-based strategy is from small and simple components to build blocks that can be assembled into more complex systems. From a single-machine communication protocol COM (component object model) for component-based software applications, DCOM was given birth to encompass communication between components on networked machines as well. Then to join the Internet trend, ActiveX controls were promoted as a way to support mobile code to compete with Java applets [OMG97].

ActiveX/DCOM implementations are available for Microsoft operating systems (Windows 95 and Windows NT), MacOS and Unix (by third parties) [CHO96].

Microsoft promotes ActiveX controls and DCOM as an alternative to Java mobile code and CORBA to encompass the Web and provide it with distributing computing capabilities. The drawback of the ActiveX control approach is the use of native x86 code and thus is platform-dependent. At present the only browsers understanding ActiveX are Microsoft’s own Internet Explorer 3.0 and 4.0, which are only available for Microsoft platforms, but not Unix. There is no word yet how ActiveX will be supported on other platforms, but Microsoft may simply dictate that all ActiveX controls should be recompiled, possibly into some sort of “fat binary” format with a separate code segment for each supported platforms.

DCOM seems not to be designed to be scalable across large-scale networks. As it uses reference counts of the number of clients to decide the life span of a computing object, the fault-tolerance of the whole system suffers from errors likely to occur in order to maintain exact information in a distributed environment. Mistakes can be insidiously easy to make by the developer, and does not guarantee error-free programs, because problems can also arise in the network. Then, DCOM has implemented a backup resource management mechanism, which is unfortunately even less scalable: keep-alive messages are sent at regular intervals to “ping” objects for their availability.

Communication security raises the issue of data authenticity and integrity. But DCOM/ActiveX communication style provides two unsafe mechanisms, one is based on the Remote Procedure Calls of DCOM, and the other is using ActiveX mobile native code to encompass communication over the Internet.

Although DCOM uses a variant of the DCE RPC mechanism for communication between remote objects, Microsoft did not use the accompanying DCE security mechanism originally developed at MIT. DCOM seems at this point to have no support at all for confidentiality or authentication of inter-object communication.

The second DCOM/ActiveX communication is to copy ActiveX controls into a remote machine to be executed there. The problem is that ActiveX controls are native x86 code, compiled from C++ without any restrictions on what they may do. A complementary approach to mobile code is to certify the integrity and authorship of the code using the public key cryptography. By verifying the public key signature on downloaded mobile code, a browser can be sure that the code really was written by the apparent author, and that it hasn’t been tampered with since. However, this verification does not guarantee that the author has no malign intent, nor that the code does not contain honest but disruptive coding errors.

2.3 Our Choice

Generally speaking, we have chosen CORBA to be the component-based software for implementing our project, not because of its functionalities (DCE and DCOM have similar functionalities), but because of its deployment on a large range of operating systems, including Unix and Windows. Our motivation is to be able to access and use heterogeneous components to be integrated into our distributed system of event detection and rule execution.

Also CORBA and DCE have definitively a maturity of specifications and products that conform to them, compared to DCOM. There is a fundamental difference between DCE and CORBA, however, that we feel far overshadows either of these criteria as a basis for selecting a platform for distributed computing: the approach of DCE uses procedural programming like the C language, while CORBA follows the object-oriented methodology to interface with the computing components.

In fact, we do not completely discard the use of DCOM as infrastructure for supporting distributed event-driven applications, because of the popularity and overwhelming presence of Windows operating systems in the world of PCs. Eventually, we will consider to port our project in the Windows NT environment, and hopefully an interface will be specify to make the two systems communicate with each other.

CHAPTER 3
ALTERNATIVES FOR ARCHITECTURE AND DESIGN

CORBA has been chosen as the component-based software to implement this project. The design of our system of event detection and rule execution is based on its features. Allowing specification of events at run-time is an important feature in our system of event detection.

Then we will see that the event notification mechanism has been designed to match the event communication model, by making use of the non-blocking call capability provided by CORBA specification. The ORB is an important component of the CORBA framework. We will explain its role for achieving integration of heterogeneous systems.

Our goal is to take advantage of heterogeneous components for our system of event detection and rule execution. In order to do that, we come up with a flexible design for the execution of ECA rules. More precisely, we extend the scope of the condition evaluation and action execution, by allowing them to be specified as operations or services provided by other components in the network. Finally, we will discuss the question of how composite events are handled in our system.

3.1 The Nature of Objects in CORBA

Although CORBA objects are just standard software objects implemented in any supported programming language, including Java, C++ and Smalltalk, each of them has a clearly-defined interface, specified in the CORBA Interface Definition Language (IDL). The interface definition specifies what member functions are available to a client, without making any assumptions about the implementation of the object [ION97].

The separation between an object's interface and its implementation has several advantages. For example, it allows you to change the programming language in which an object is implemented without changing clients that access the object. This capability is used to bring together heterogeneous systems written in different languages.

3.2 The CORBA Services

The CORBA services are sets of objects defined by CORBA that provide useful services for some distributed applications [ION97]:

- The *Event Service*. This service allows objects to communicate using decoupled, event-based semantics, instead of the basic CORBA function call semantics.
- The *Naming Service*. Before using a CORBA object, a client program must get an identifier for the object, known as an *object reference*. This service allows a client to locate object references based on abstract programmer-defined object names.
- The *Trader Service*. This service allows a client to locate object references based on the desired properties of an object.
- The *Object Transaction Service*. This service allows CORBA programs to interact using transactional processing models.

- The *Security Service*. This service allows CORBA programs to interact using secure communications.

The Naming Service and Trader Service is useful to provide a visibility of the various components at a global scope. Since a user can specify the rule conditions and actions *dynamically* using existing operations implemented by some server applications on the network, he is certainly interested in having a list of such distributed resources.

In some cases, the rule execution is required to follow a transactional processing model, which is greatly facilitated by the Object Transaction Service. In other cases, the security issue becomes predominant. If the communication takes place over public computing networks then the authenticity of the data and its integrity while being transferred may be at risk. CORBA specification provides the Security Service for that purpose.

Finally, the Event Service has been provided to support an event communication model suitable for event detection and propagation. But we will see the drawbacks of using it for our project in the following subchapter.

3.3 The Event Service and Its Limitations.

The *supplier-consumer* communication model allows an object to communicate an important change in state, such as a disk running out of free space, to any other objects that might be interested in such an event [VIS97].

The Event Service provides a facility that decouples the communication between objects. It provides a supplier-consumer communication model that allows multiple supplier objects to send data asynchronously to multiple consumer objects through an *event channel*.

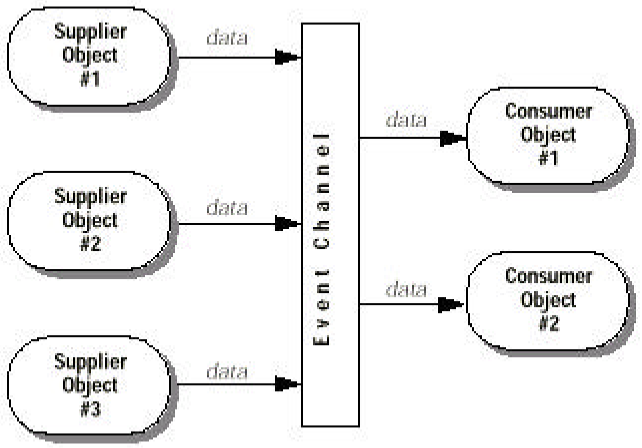

Figure 3.1: The supplier/consumer communication model

Figure 3.1 shows three supplier objects communicating through an event channel with two consumer objects. The flow of data into the event channel is handled by the supplier objects, while the flow of data out of the event channel is handled by the consumer objects. If the three suppliers shown in Figure 3.1 each send one message every second, then each consumer will receive three messages every second and the event channel will forward a total of six messages per second.

The data communicated between suppliers and consumers are represented by the **Any** class, allowing any CORBA type to be passed in a type safe manner. Supplier and consumer objects communicate through the event channel using standard CORBA requests.

The event service provides both a pull and push communication model for suppliers and consumers. In the push model, supplier objects control the flow of data by pushing it to consumers. In the pull model, consumer objects control the flow of data by pulling data from the supplier.

The event channel behaves like an event queue to regulate the flow of data to be processed by an application. The size of the buffers containing those events can be configured before starting the event channel but not at run-time.

The event channels provided by the CORBA Event Service seem to be the most appropriate mechanism to transmit events between distributed applications. But there are several reasons why we are not using them.

First, the availability of the Event Service depends on the CORBA vendor. At this point in time, the Event Service is not available for OrbixWeb. However, it is available for the C++ version of Orbix. We can also mention that Visigenic provides an Event Service package.

The event channel consumes too many system resources for what we want to achieve, that is, multicasting instead of broadcasting. The event channel is a black box that provides the users with an API allowing them to subscribe for, notify and push/pull events. Supplier and consumer objects communicate through the event channel using standard CORBA requests. Consumers and suppliers are completely decoupled from one another through the use of proxy objects. Instead of directly interacting with each other, they obtain a proxy object from the **event channel** and communicate with it. The event channel facilitates the data transfer between consumer and supplier proxy objects. Figure 3.2 shows how one supplier can distribute data to multiple consumers.

In our case, it would be a waste of resources to use event channels. Since there is only one consumer (the rule server) for several suppliers, there is no need of having proxy objects that only add delay in the event communication.

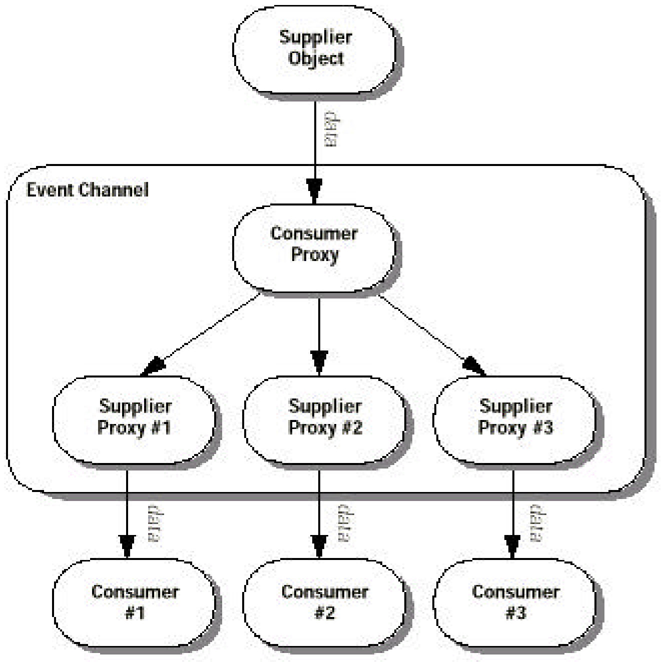

Figure 3.2: Consumer and supplier proxy objects

As we can see in Figure 3.3 multiple applications can be consumers of events, but they may not be interested in the same events. In order to achieve multicast communication, each consumer can filter the flow of data broadcasted through the event channel. Indeed, any consumer who subscribes for a particular event channel receives all the corresponding event notifications. But we do not want to overload the network with unnecessary data transmission.

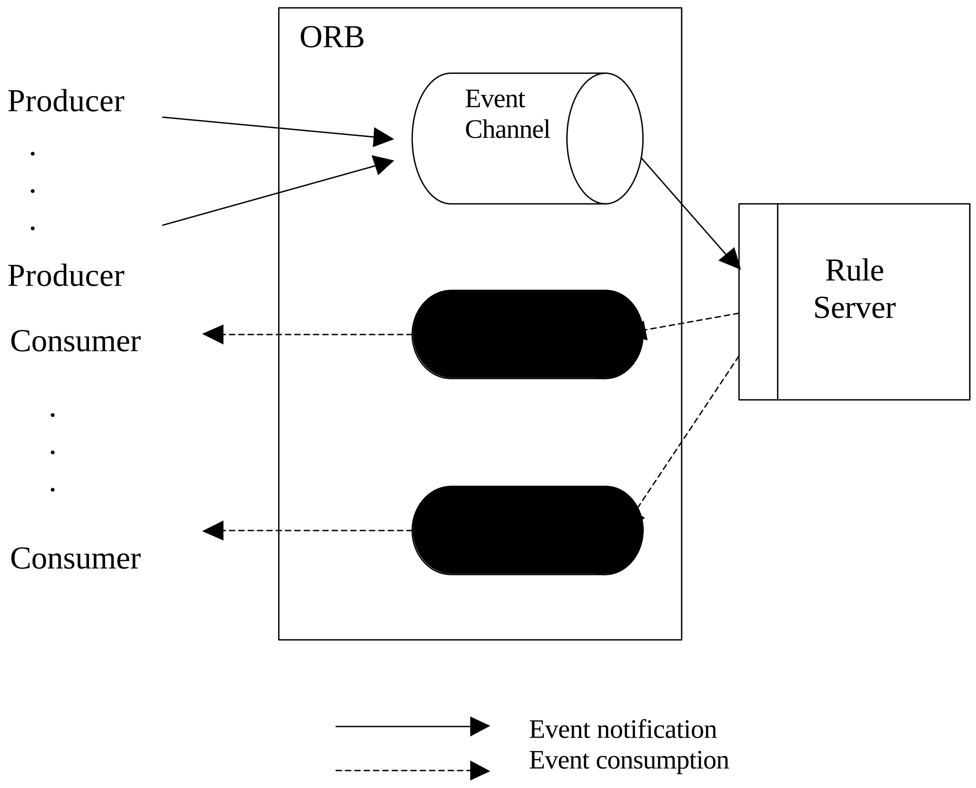

Figure 3.3: Architecture using event channels

Another alternative would be to associate one event channel with one and only one event type. It would also simplify the management of subscription/unsubscription to a particular event for each consumer. The number of event channels will grow with the number of user-defined events. Since it is a costly resource, the event channel turns out to be an inappropriate solution for scalability.

Finally, the design of our rule execution will definitively discard the last doubts about using event channels.

Our design simulates the model of event suppliers/consumers, without actually dispatching events to the consumer applications. In fact, the consumer application is a server that implements operations, some of which can be used as part of an ECA rule condition or action. Instead of being specified inside the applications, the ECA rules are stored in the rule server. The rule server is also responsible for the detection of primitive and composite events; it receives primitive event notification from suppliers and integrates an event graph for composite event detection. Events are not propagated to the applications; they are used in the address space of the rule server to trigger ECA rules, although the conditions and actions of those rules may be executed remotely.

3.4 Registering Events at Run-Time

The server design must take into account introduction of new event types into the system without having to be recompiled and restarted. This issue does not apply to the design of event suppliers, because their implementations are statically linked with the specification of the events they can raise. In other words, they cannot generate new event types during their lifetime.

Instead of having a single generic method call, we can implement a different method for each type of event. That would require recompiling the server whenever a new event type is introduced into the system. Another alternative for introducing new event types is the use of a template structure to specify new event types.

3.4.1 Java Reflection

Registering new events requires dynamic linking of event libraries. But there is an implementation issue about the dynamic linking mechanism provided by Java.

Java reflection allows to query information, including attribute value, at run-time on an instance of a class unknown at compile time, typically casted into an instance of class **Object**. It also allows to make method calls on that instance. We will use this Java facility to manipulate data types (events) introduced into the system at run-time.

Here is an example of Java code that makes a method call contained in a library loaded at run-time (the dynamic version of the example regarding the extraction from an instance of type *Any*):

```
import java.lang.reflect.*;


        // create formal parameters
Class formalParams[] = new Class[1];
formalParams[0] = Class.forName("org.omg.CORBA.Any");


        // create method stub
Method extractMethod =
helperClass.getDeclaredMethod("extract", formalParams);


        // create instance of helper class
Object helperObject = helperClass.newInstance();


        // params declaration
Object methodParams[] = new Object[1];
```

```
        // instanciate parameters
    methodParams[0] = eventAny;


        // invoke method
    Object event = extractMethod.invoke(
    eventRegistration.helperObject, methodParams);
        // to be casted in the proper type when using the event
```

This Java mechanism reminds of Dynamic Link Library (DLL), although it has an important limitation: once the library has been loaded, it cannot be reloaded again, without restarting the application. In other words, if the library had to be recompiled for schema evolution reasons, the changes will not be seen by subsequent calls to it within the application.

3.4.2 Alternative Solution: Event Template

Another alternative for introducing new event types is to agree about an event template, so that every event specification follows a model for defining the attributes. This solution may be appropriate if there is no dynamic linking supported by the language used to implement the system. But when the event structures become complex (large number of attributes, nested structures, user-defined types), this solution for creating an event instance is cumbersome to implement, especially if using a hierarchy of linked lists. In contrast, CORBA IDL can specify a large range of types, including nested structures and the generated Java classes are automatically used for the data marshalling and communication. Thus, if we make full use of the IDL expressiveness, it would simplify the implementation, as well as the view that a developer has of the definition of new event types. In fact, the Java code mapping to complex event structures is almost identical to the event structure originally specified, so that it would be more straightforward to insert parameters into and retrieve them in the supplier and server implementation, than if we had to manipulate linked lists of parameters.

3.5 Event Notification Using Non-Blocking Calls

3.5.1 Event Notification Using CORBA Method Calls

The mechanism of event passing can be implemented by using asynchronous method calls, as we are trying to avoid the overhead associated with the use of event channels. The event parameters can be the actual parameters of the request, or they can be encapsulated in a data type.

3.5.2 Non-Blocking Calls (“oneway”)

CORBA specifies a way to make non-blocking calls, by declaring operations as **oneway** in the IDL definition. The delivery semantics for an oneway requests are "best-effort" only; that is, a caller can invoke an oneway request and continue processing immediately, but will not be guaranteed that the request will arrive at the server.

An IDL operation may be declared as oneway only if it has no return value, out, or inout parameters. An oneway operation can only raise an exception if a local error occurs before an invocation is transmitted.

3.6 Communication and Connection Transparency Using CORBA

3.6.1 The Role of an Object Request Broker

CORBA defines a standard architecture for ORBs. An ORB is the software component that mediates the transfer of messages from a program to an object located on a remote network host. The role of the ORB is to hide the underlying complexity of network communications from the programmer.

An ORB allows you to create standard software objects whose member functions can be invoked by *client* programs located anywhere in your network. A program that contains instances of CORBA objects is often known as a *server*.

When a client invokes a member function on a CORBA object, the ORB intercepts the function call. As shown in Figure 3.4, the ORB redirects the function call across the network to the target object. The ORB then collects results from the function call and returns these to the client.

3.6.2 Interoperability between Object Request Brokers

The components of an ORB make the distribution of programs transparent to network programmers. To achieve this, the ORB components must communicate with each other across the network.

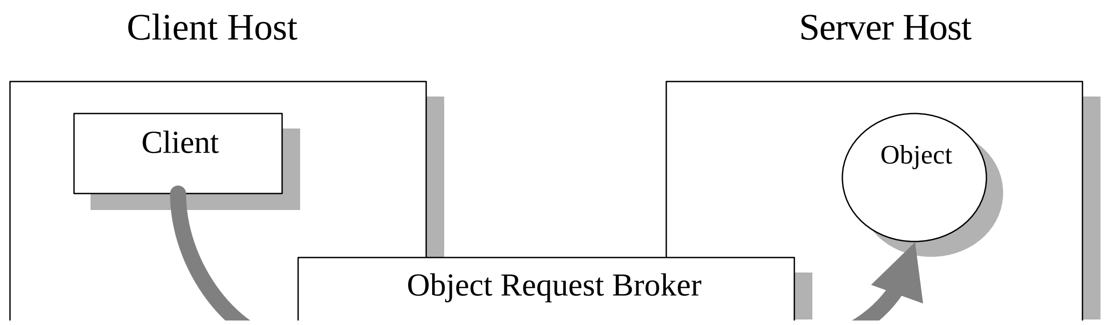

Figure 3.4: Object request broker

In many networks, several ORB implementations coexist and programs developed with one ORB implementation must communicate with those developed with another. To ensure that this happens, CORBA specifies that ORB components must communicate using the inter-communication protocol IIOP.

The inter-communication between ORBs is the typical way to bring together heterogeneous environments. Some CORBA applications can only connect to a particular ORB for various reasons: incompatibility with the ORB because of the programming language used to implement them, incompatibility when an application developed with one CORBA product is trying to connect directly with the ORB of another vendor.

Fortunately, the communication between two ORBs from different vendors has been made possible by CORBA IIOP protocol, now supported by most of the CORBA implementations. Heterogeneous systems can interact with each other thanks to the network of ORBs that constitute a gateway system in a distributed environment.

3.6.3 Transparency to Locate CORBA Applications on the Network

A client application connects to a server in a process called *binding*.

The parameter of host name can be indicated in the binding call to look for a server instance on a specific host. But one of our goals in the composite event detection and rule service (CEDAR) design is to avoid the client application to be concerned with the server location on a particular host of the network; if the CEDAR server were located on a remote network, then it would be necessary to indicate a specific host.

One solution is to provide the application with the host name by some means: hard-coding this information is not a flexible solution, but an option on the command line or a configuration file can be acceptable solutions. The best strategy is to mix those solutions with the use of the *locator* feature in Orbix as described below.

Most of the products that implement CORBA specification such as Orbix and Visibroker offer a convenient feature that allows the user to specify the name of the host where a server is running in a system configuration file, so that this information is not hard-coded in the client implementation, but instead can be shared by all the components of our system. In fact, it allows more than that: the user can indicate a list of host names where the server is *likely* to be running. Then the ORB will try to locate an instance of the server on one of them at run-time in a random order.

Visibroker actually goes one step further by using broadcasting within the local network in order to find a server, so that the user will not need to associate each server with a list of possible host names.

3.7 The Event Queue Model

An event queue is an important component of our architecture to regulate the flow of events coming to the server. When the events are not processed fast enough, they have to be temporarily stored in a structure.

In our design, we avoid implementing an event queue because it is already taken care of by the communication layer. In fact, this crucial component of the event push model can be mapped to the functioning of an iterative server. In the push model, events that are not processed yet by the application are stored in a queue. Similarly, when the flow of client requests arriving at the server is too big to be handled, the communication layer will store them in a queue. Later on, the server dequeues them one by one and processes them in order.

3.8 Rule Execution Using DII

Our motivation is to extend the concept of ECA rule, whose condition and action can be specified to map to existing and possibly remote method implementations provided by other systems and made visible through an application programming interface (API) defined in CORBA specification language IDL.

As we stressed it before, reusing the legacy codes is a cost-efficient solution to achieve the paradigm of distributed computing, provided that integrating those legacy systems only requires minor changes, which is usually not true. Fortunately, CORBA philosophy has been working to provide a clear interface specification that can easily map to the various underlying implementations. Once a program has defined its interface in

IDL, it is considered as a CORBA object and behaves like a server that implements a set of operations available to any CORBA client. Then, those operations can be used to specify any ECA rule condition or action in our rule server.

Our rule specification system is highly flexible, because it enables the user to define rule dynamically. In fact, a rule action can be modified on the fly to do something else; the user just needs to point it to another operation supported by some CORBA object.

Rules are not exactly executed in the address space of the CEDAR server, although their executions are initiated there. In fact, the condition evaluation and the action are performed in the CORBA servers that implement theirs conditions and actions, as if they are mere service requested by a client.

Our design defines an ECA rule as a data structure that stores the name of the event triggering the rule, the method calls for the condition and action to be carried out at runtime using the dynamic invocation interface (DII).

Another advantage for designing the rule execution that way is that we simplify the rule management by removing the notion of application subscribers. There are no consumer applications that subscribe for events of interest; instead they are replace by application containers of conditions and actions. In fact, the ECA rules are the only entities subscribing for events; the logic of their execution remains in the rule server and are not propagated to the client applications, which makes it easier to manage and control the rule execution.

3.8.1 The Structure of a Dynamic CORBA Application

One difficulty with normal CORBA programming is that you have to compile the IDL associated with your objects prior to using the generated Java code in your applications. This means that your client programs can only call member functions on objects whose interfaces are known at compile-time. If a client wishes to obtain information about an object's IDL interface at runtime, it needs an alternative, *dynamic* approach to CORBA programming.

The CORBA **Interface Repository** is a database that stores information about the IDL interfaces implemented by objects in your network. A client program can query this database at runtime to get information about those interfaces. The client can then call member functions on objects using a component of the ORB called the **Dynamic Invocation Interface** (DII), as shown in Figure 3.5, so that the method call can be chosen at runtime.

3.8.2 DII: Advantages and Drawback

CORBA specifies a powerful mechanism called DII to build requests (method calls) at run-time. Using DII on the client side does not require changing the implementation of the server; thus it does not create additional complexity when bringing together systems that are already implemented.

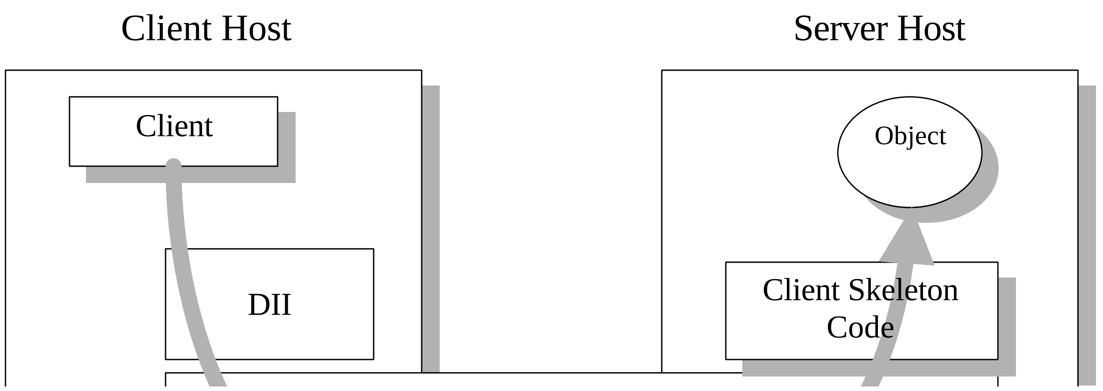

Figure 3.5: Invoking on a method using DII

The drawback of the DII is that it performs worse than static method invocation because of the overhead of building the request step by step at run-time. It put some complexity to implement a method call, as compared to the static method invocation but this is the only way to create generic method calls. The same comparisons can be drawn between dynamic SQL and static SQL.

3.8.3 An Alternative To Dynamic Method Invocation: Feature and Limitation

There is an alternative to dynamic method invocation: compiling the corresponding static calls at runtime.

In other words, the Java compiler will be used to create a library of calls for the condition and action whenever a new rule is specified. Compiling Java modules at runtime requires a mechanism to load them dynamically later on.

This approach can speed up the method invocations, although it introduces the overhead of compiling source code, which occurs whenever a rule has been defined for the first time or is modified. Besides it is not as meaningful for the system performance as the number of times a rule is executed.

Compiling code is an attractive solution for another reason. This mechanism is actually used to create user-defined conditions to be executed in the address space of the CEDAR server. For example, if a rule condition is simple enough to be evaluated locally in the rule server instead of being the result of a remote call, it can be compiled into a library call and reused for other rules. Thus we can have a unified mechanism for loading the rule libraries no matter if the rule conditions and actions are CORBA remote invocations or internal calls of methods coming from user libraries. This approach is attractive because it involves fewer changes to extend the Sentinel previous work.

The major drawback of the compilation approach is that it is difficult to run the Java compiler from within a Java program. In fact, Java does not allow to make system calls, like in the C language (*system("javac")*).

The number of files managed by the CEDAR system can become huge with the number of rules defined by the user because a library module has to be created for each rule.

In our implementation, rule execution is exclusively based on CORBA method invocations. But in future work, calls to Java libraries will be allowed, which will greatly enhance the scope of rule conditions and actions.

3.8.4 Extension of the Rule Execution Scope

Our rule execution model takes advantage of remote operations implemented by other CORBA servers in order to specify the condition and action of a rule. But in most cases, condition methods are likely to be simple and do not need to be implemented by a remote server to avoid the overhead of invoking them through the network. This is illustrated by the example of a CEDAR server receiving stock updates, where one rule has been defined to execute the action of buying thousands stocks if their price goes up beyond a certain threshold value. That straightforward condition can be processed in the same address space of the CEDAR server after being parsed or with a local call to a Java library. The user is the one to decide if the call should be made locally or remotely.

The execution of the methods used for conditions and actions is not bug-free. When an error occurs, it can raise an exception that needs to be caught, so that it would be easier for diagnosis. If the exception is not caught, it will be passed to the calling function, which may lead to an exit from the program.

3.9 Composite Event Detection

Composite events can be detected on the server side using of an event graph, which also detects primitive events.

Composite events [LIA97] are detected on the server using an event graph. An event tree is created for each composite event and these trees are merged to form an event graph for detecting a set of composite events. This avoids the detection of common sub-events multiple times thereby reducing storage requirements. The leaf nodes are made of primitive events, whereas the non-leaf nodes represent global composite events.

Whenever a primitive event is detected, it will propagate the event notification to its parent nodes. The parent nodes maintain the occurrence of its constituent events along with their parameter lists. If the composite event occurs by the last notification, it is detected and further propagated to its parent nodes.

CHAPTER 4
CEDAR: DESIGN AND IMPLEMENTATION

The CEDAR service is designed to provide primitive and composite event detection as well as rule processing in a distributed environment. This chapter presents the detailed design and implementation issues related to each functional module of the CEDAR system. The CORBA-compliant product we decided to use to implement this project is OrbixWeb, developed by IONA. This CORBA implementation allows you to build and integrate distributed applications written in Java. OrbixWeb is a full implementation of the OMG’s CORBA specification, and support features like DII and Interface Repository. We will first see a description of the CORBA-compliant components specific to the OrbixWeb product. Then we will see how the CEDAR system has been implemented to provide flexibility to the users, by allowing the introduction of new event types into the system at runtime. Finally, the system of composite event detection has been carried out by re-using a component of the Sentinel local event detector (LED), the event graph. It was translated in Java to the case of Sentinel applications written in this language instead of C++.

4.1 The Structure of a CORBA Application

The first step in developing a CORBA application is to define the interfaces to objects in your system, using CORBA IDL.

An IDL compiler generates files (in Java, for example), including *client stub code*, which allows you to develop client programs, and *server skeleton code*, which allows you to implement CORBA objects. As shown in Figure 4.1, when a client calls a member function on a CORBA object, the call is transferred through the client code to the ORB. The ORB then passes the function call through the server skeleton code to the target object.

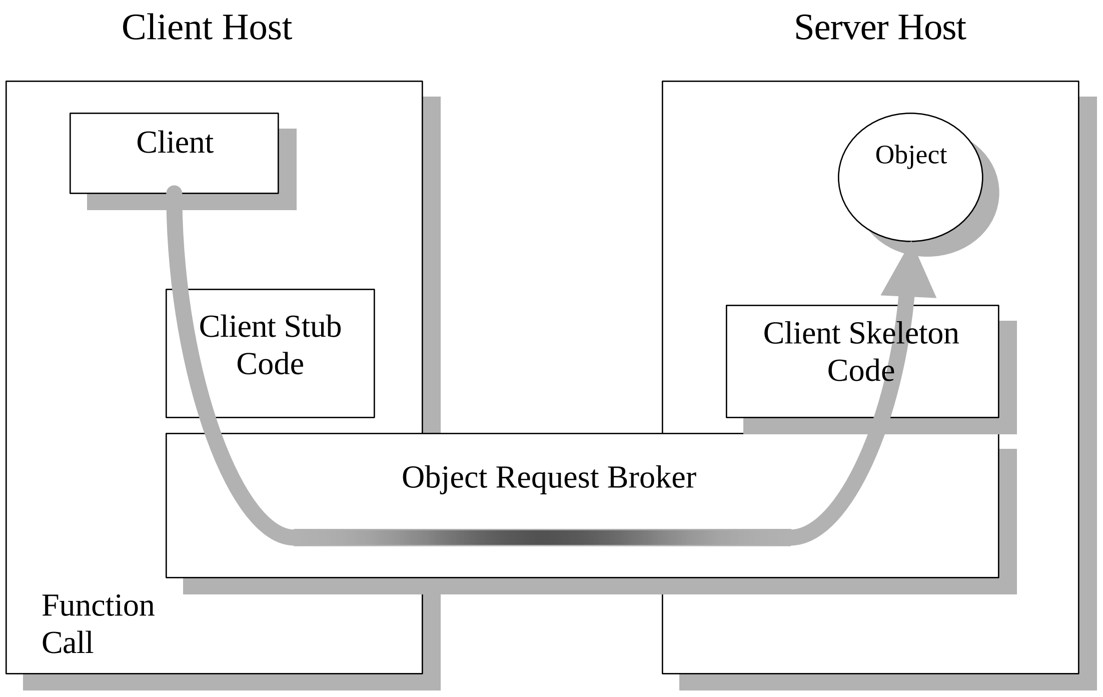

Figure 4.1: Invoking on a CORBA object

OrbixWeb is an ORB that fully implements the CORBA 2.0 specification. By default. All OrbixWeb components and applications communicate using the CORBA standard IIOP protocol.

The component of OrbixWeb are as follows:

- The *IDL compiler* parses IDL definitions and produces Java code that allows you to develop client and server programs.
- The *OrbixWeb runtime* is called by every OrbixWeb program and implements several components of the ORB, including the DII, the DSI, and the core ORB functionality.
- The *OrbixWeb daemon* is a process that runs on each server host and implements several ORB components, including the Implementation Repository. An all-Java counterpart to the daemon process is also included. This daemon process is also known as the Java Activator, also referred to as **orbixdj**.
- The *OrbixWeb Interface Repository server* is a process that implements the Interface Repository.

4.2 Registering Events

The CEDAR system is designed to provide flexibility to the users, by allowing the introduction of new event types into the system at runtime.

4.2.1 Syntax of Event Definition

The CORBA IDL is used to define interfaces to objects in the network. We also use the same mechanism it to specify event definition using the *struct* data type, which allows you to package a set of named members of various types. New event types are introduced to the system by compiling their IDL description so that the corresponding stubs can be generated in order to manipulate event instances in the implementation of the server and suppliers:

```
struct Stock {
   string symbol;
   float price; };
```

Figure 4.2: Example of event specification: Stock.idl

Once the Stock.idl file is compiled by an IDL compiler, the developer can manipulate an event instance in the implementation. Here is a Java example that explains how to load event libraries and how to use them to manipulate event instances:

```
import Stock;       //  event Stock
Stock stock = new Stock();
stock.symbol = “IBM”; stock.price = (float) 133.5;
```

Figure 4.3: Creation of an instance of type Stock

4.2.2 Dynamic Subscription Using Java Reflection

Client or server applications that use event structures in their implementation should be compiled using the corresponding stub libraries. This is typically done at compile time of the application, which means that after the application has started, it cannot accept subsequent event types. For example, the keyword *import* is used in Java to load required libraries when the application is started:

```
import Stock;      //  load the library class Stock
```

Figure 4.4: Example of static linking of library in Java

In order to go beyond the limitation of predefined event types, another mechanism has to be used to load event libraries at run-time. It is based on the Java feature of reflection. The following code example does the same thing as previously except that loads a library at run-time using Java reflection [AGA98]:

```
import java.lang.reflect.*;
Class helperClass = Class.forName("Stock");
```

Figure 4.5: Dynamic linking of library using Java reflection

4.3 Sending Event Notifications

In order to pass the event notification from the supplier application to the CEDAR server, we have chosen a simple generic CORBA method call at the interface of the server instead of using other mechanisms like the event channel or the generation of a particular method call for each event type.

The mechanism we choose to transmit any type of event, or more precisely type unknown at compile time, is based on a generic non-blocking method call *raiseEvent()*, which makes use of the type *Any* to pass safely the event instance as well as its parameters. Manipulation of the type *Any* will require access to the event libraries.

```
// raiseEvent():
   oneway void raiseEvent(in any event);
```

Figure 4.6: Non-blocking call for raising events (IDL specification)

The generic method call *raiseEvent()* takes the event instance as a parameter through the use of the type *Any*. Because the event type specifications differ from each other, we need to use a mechanism similar to *void ** like in C in order to pass parameters of type undetermined at compiled-time. Using the CORBA type *Any* is the safe way to pass different types of parameters as specified by CORBA specification. Technically speaking, the event instance is embedded into an instance of type *Any* through the *value* field, and the type information is encoded into the *type* field. In our case, the type is a Java class mapping to the event structure. In fact, the *value* field cannot be manipulated directly, but instead the event instance is inserted and extracted from the holder type *Any* using the methods *insert()* and *extract()* from the libraries generated by the IDL specification of the event types. Those libraries are constituted by adding the suffix *Helper* like in the following example:

```
import Stock;

import org.omg.CORBA.Any;

Stock stock = {“IBM”, 103.45};

Any event = new Any();  // or using ORB.init().create_any()

StockHelper.insert(event, stock);

RaiseEvent(event);  // notify server
```

Figure 4.7: Encapsulate event into an Any instance before notification

Without those libraries, it is not possible to extract from an instance of the type *Any*. That is the reason why we need a DLL-like mechanism to load those libraries at run-time, especially when specifications of new event types are compiled and enter the system after it has been started.

4.4 Connecting to the CEDAR Server

4.4.1 Configuration Files (*Orbix.hosts* and *Orbix.hostgroups*)

Locating CORBA objects in the network is made transparent: it is not necessary to explicitly specify the host name during the binding, except for locating an object on a particular machine. OrbixWeb provides the *locator* feature for this purpose. Specification of hosts for a particular application is done in a configuration file.

Using Orbix, here is how the configuration file *Orbix.hosts* can look like:

```
PED:lightning,manatee,coconut:
```

The format of each line is:

<server_name>:<list_of_hosts>:<host_group>

At this point in time, the OrbixWeb users may be unhappy because this configuration file feature is not available with the pure Java version of OrbixWeb. In other words, the version of ORB daemon that must be running in the background of every machine must be *orbixd* and not *orbixdj*. This is not a crucial limitation unless you want to integrate the CEDAR system with the World Wide Web by making the ORB (besides the application) an applet that can be downloaded by any JAVA-enabled browser.

Filling out the configuration file can be tedious when the same list of host names corresponds to several servers. That is the reason why another configuration file *Orbix.hostgroups* is provided in order to avoid repeating the same list of host names, as shown in the following example:

```
allNodes:host1,host2,host3,host4,host5,host6
```

and in the *Orbix.hosts* file, the format of each line is: <group_name>:<list_of_hosts>

4.4.2 Procedure Steps to Connect

All the steps to connect to the CEDAR server have been embedded in a single procedure *Connections.connect2PED()*. They are the same as indicated in the CORBA specification and the OrbixWeb manual.

First step to accomplish is to connect to a CORBA server, and in order to do that you need to make a binding call on a static class derived from the IDL specification of the server.

The procedure of connecting to the CEDAR server has been made easy to follow by embedding all the necessary method calls of CORBA API within a single method call.

The implementation details follow the typical steps of connecting to any CORBA servers, although they include the use of the locator feature, so that the binding call to the CEDAR server is made without providing host names.

4.5 Composite Event Detection

After presenting the API for the use of the event graph implemented in Java, we will explain how we integrate it with the CEDAR system.

4.5.1 API of LED Written in Java

We have translated the event graph module of the Sentinel project originally written in C++ in Java.

1. Initialize LED library:

```
globalLED.initializeLED();
LED aLED = globalLED.aLED;
```

2. Specify events to be detected: build the event graph with primitive/composite nodes:

```
EventNode node1 = aLED.createPrimitive("e1");
EventNode node2 = aLED.createPrimitive("e2");
aLED.createAnd("e3", node1, node2);
```

3. Create rule instances and associate (subscribe) them to events:

```
RehashingRule aRehashingRule = new RehashingRule();
```

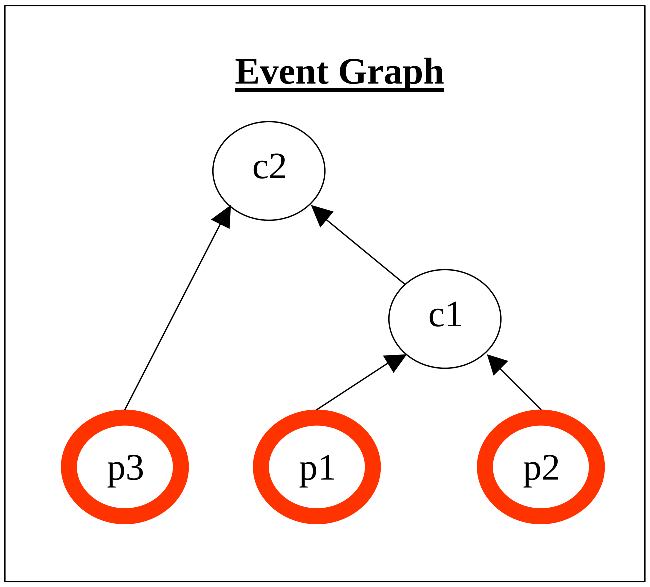

Figure 4.8: Creation of detection nodes for primitive events

```
aLED.addSubscriber("e3", aRehashingRule);
```

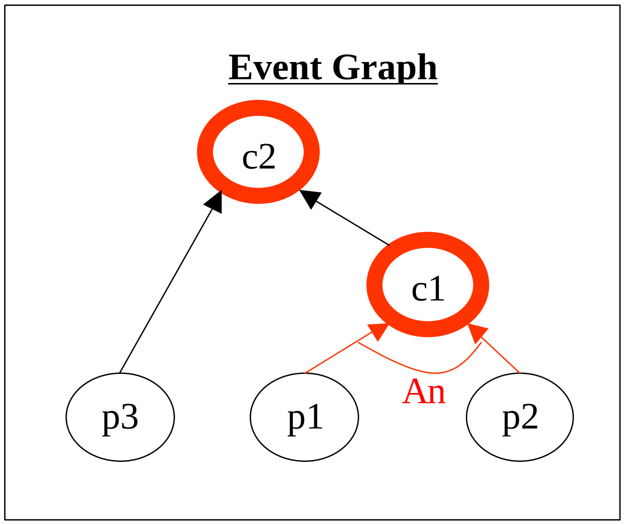

Figure 4.9: Creation of detection node for composite events

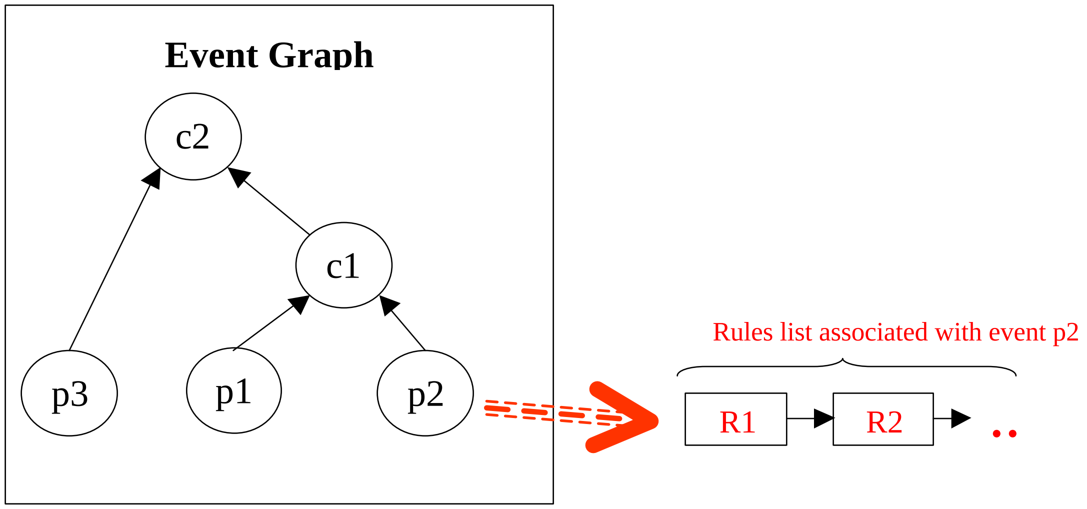

Figure 4.10: Event subscription by a list of rules

4. Raise events:

PrimitiveEventNotif.notify("e1");

PrimitiveEventNotif.notify("e2");

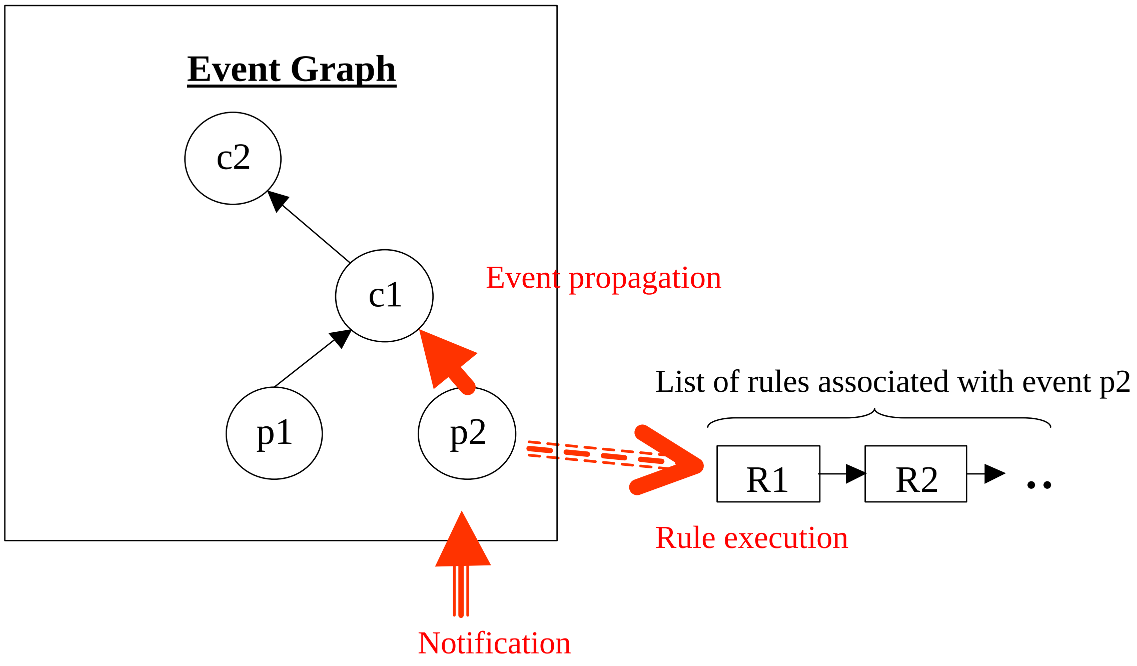

Figure 4.11: Raising primitive events

To create a rule, one needs to implement the interface *RuleObject* with respect to the condition and action of the rule.

4.5.2 Integration of the Event Graph With the CEDAR Implementation

The Java version of LED based on an event graph is can detect primitive events and some types of composite events. When implementing the detection of composite events within CEDAR framework, we retain the concept of event graph and rule subscription borrowed from LED.

Currently, CEDAR implements a straightforward system of primitive event detection and rule firing. The main data structure is the rule manager built on a hash table whose keys are the names of the primitive events registered through an API, while each of its buckets contains a linked list of rules to be triggered when the corresponding event has been raised.

The system of composite event detection is achieved by using the LED event graph and to create a particular rule between the event graph and the CEDAR primitive event detection. On one hand, we make the set of hash table keys include the names of composite events; each key can be either a primitive or a composite event because they are handled in the same way; the list of rules associated with a composite event works not differently from the case of primitive events. On the other hand, the detection of composite events is done using the event graph where all the rules subscribed to composite event nodes are the same particular rule called *'RehashingRule'* whose function is to "raise" or hash the composite event in the rule manager hash table.

The event graph has to be placed efficiently within the whole detection process. Its input can be connected to the output of the hash table, which means that each primitive event linked list of rules contains a particular rule that will call "notify" routine to propagate the event notification through the event graph. We can avoid this additional overhead by calling directly the "notify" routine within the CORBA call *raiseEvent()* instead of going through the hash table. But in any case, the composite event detecting process is sequential with the primitive event counterpart; it can be decided prior to or following the primitive event detection. In fact, from a conceptual point of view, it would be better to have both of them run concurrently. This can be implemented using threads.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

This dissertation presents an approach to provide the best architecture and framework to support ECA rules to be extended and integrated with distributed and heterogeneous systems.

CORBA has been chosen to be the component-based software for the implementation for our prototype against DCOM and the event service. Dynamic specification of events and rules has been one of our principal motivations as well as the expressiveness of composite events.

The Composite Event Detection And Rule Execution (CEDAR) service has been designed to provide primitive and composite event detection as well as rule processing in a distributed environment.

The CEDAR system succeeds to provide flexibility to the users, by allowing the introduction of new event types into the system at runtime.

Our design of ECA rules makes it easier to achieve distributed computing and integration of heterogeneous systems. Rule conditions and actions can be method calls on other CORBA objects and can be specified at run-time.

By integrating an event graph in our system, we address the aspects of specification, detection, and management of composite events, because we believe it is important for a large class of real-life applications.

5.2 Future Work

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

- Support active database semantics for the meaning of event notification: update, insert and delete on objects.
- Add a graphical user interface on top on the already existing API. Then it will be easier to register for new events and to specify rules at run-time.
- Extend rule execution besides the use of DII, by permitting calls to Java user-libraries.
- Enable an application to be a condition/action container and an event supplier at the same time. This may require the use of callbacks.
- Increase the expressiveness of the rule condition specification, by covering a large range of logical operators and allowing new data type to be registered into the system, much like in ORDBMS.
- Use threads for rule execution to increase the performance of the CEDAR server when running on a multi-processor machine.
- Implement a recovery system in case of server failure, which will require to persist events.

- The current implementation supports immediate coupling mode. We plan on extending this with detached and deferred coupling.

Obviously not all library methods can be used to implement conditions: only those methods returning a value that can be used in a comparison predicate. If the method returns a boolean value, it is not necessary to explicitly compare it with true or false; only the signature of the method call is parsed, because the entire condition is implemented as a single method returning a boolean value. However, it would be interesting to extend the condition syntax by supporting predicates for any primitive type like string, integer, etc. provided that the parser has been extended to accept the additional comparison operators. Then it would be possible to take any operation that returns a value and make it part of a condition predicate like in the following:

```
If ( Database.getName(id) == “Bob” ) then execute action…,
```

where *Database* is a CORBA object that maintains a database of correspondence of names with Ids.

Supporting predicates makes it more flexible to define the condition. If the user desires to change the value of a constant on the right side of an operator, he will not need to make any changes in the implementation of the condition. Furthermore, predicates increase the expressiveness as well as the readability of the condition definition. It would even be possible to make two remote calls within the same condition and compare their results:

```
If ( getName1(id) == getName2(id) ) then execute action…
```

The syntax of the predicates can be extended if the parser can recognize new abstract data types by registering them as well as their operators to the CEDAR server, which is similar with Object Relational DBMS to some extent.

LIST OF REFERENCES

[AGA98] Agarwal S., Event Management in an Active Object Request Broker. Master's thesis, University of Florida, Gainesville, May 1998.

[ANW93] Anwar E., Maugis L., and Chakravarthy S., A New Perspective on Rule Support for Object-Oriented Databases. In Proceedings, International Conference on Management of Data, pages 99—108, Washington, D.C., May 1993.

[BAD93] Badani R., Nested Transactions for Concurrent Execution of Rules: Design and Implementation. Master's thesis, University of Florida, Gainesville, December 1993.

[CHA89] Chakravarthy S., HiPAC: A Research Project in Active, Time-Constrained Database Management (Final Report). Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, August 1989.

[CHA94a] Chakravarty S., Krishnaprasad V., Anwar E., and Kim S.-K., Composite Events for Active Databases: Semantics Contexts and Detection. In Proceedings, 20th International Conference on Very Large Data Bases, pages 606—617, Santiago, Chile, August 1994.

[CHA94b] Chakravarthy S. and Mishra D., Snoop: An Expressive Event Specification Language for Active Databases. Data and Knowledge Science, 13(3), October 1994.

[CHA95] Chakravarthy S., Tamizuddin Z., and Zhou J., SIEVE: An Interactive Visualization and Explanation Tool for active Databases. In Proceedings, 2nd International Workshop on Rules in Database Systems (RIDS'95), pages 179—191, Athens, Greece, October 1995.

[CHO96] Choppell D., Understanding ActiveX and OLE, A guide for developers & managers. Microsoft Press, Seattle, WA, 1996.

[HAN97] Hanson E. and Khosla S., An Introduction to the TriggerMan Asynchronous Trigger Processor. In Proceedings, 3rd International Workshop on Rules in Database Systems (RIDS'97), pages 51—66, Skovde, Sweden, June 1997.

[ION97] IONA, The OrbixWeb 3.0 Programmer's Guide. http://www.iona.com/products/internet/orbixweb/fordevelopers.html, July 1997.

[JAE97] Jaeger U., Event Detection in Active Databases. Humboldt University of Berlin, Berlin, Germany, May 1997.

[LAM97] Lam H. and Su S. Y. W., ECAA Rules and Rule Services in CORBA. White paper published in the Object Management Group (OMG) web site, document number cf/97-01-09, http://www.omg.org, January 1997.

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

[OMG97] OMG, Comparing ActiveX and CORBA/IIOP. http://www.omg.org/news/activex.htm, February 1997.

[SCH96] Schwiderski S., Monitoring the Behavior of Distributed Systems. Ph.D thesis, University of Cambridge, London, 1996.

[SU95] Su S. Y. W., Lam H., An Extensible Knowledge Base Management System for Supporting Rule-based Interoperability among Heterogeneous Systems. Keynote Paper, Conference on Information and Knowledge Management, pages 1—10, Baltimore, MD, December 1995.

[SU96] Su S. Y. W., Lam H., NCL: A Common Language for Achieving Rule-Based Interoperability among Heterogeneous Systems. Journal of Intelligent Information Systems, Special Issue on Intelligent Integration of Information, Vol. 6, pages 171—198, 1996.

[VIS97] Visigenic, VisiBroker Naming and Event Services 3.0, Programmer's Guide, http://www.inprise.com/techpubs/visibroker/#NELink, July 1997.

BIOGRAPHICAL SKETCH

Roger Le was born on July 24, 1972, at Bourg-la-Reine, France. He received his Bachelor of Science degree in computer science from l'Ecole Nationale de l'Aviation Civile (ENAC), Toulouse, France, in July 1996. In the Spring of '97, he started his graduate studies in computer and information science and engineering at the University of Florida. He expects to receive his Master of Science degree in computer and information science and engineering from the University of Florida, Gainesville, Florida, in August 1998. His research interests include integrating applications with component-based object 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.

Eric N. Hanson
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.

Joachim Hammer
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.

December 1998

Winfred M. Phillips
Dean, College of Engineering

M. J. Ohanian
Dean, Graduate School