EVENT HISTORY PROCESSING IN AN ACTIVE DISTRIBUTED OBJECTS SYSTEM

By

HARINI RAGHAVAN

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

1999

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Herman Lam, for giving me an opportunity to work on this challenging topic and for providing continuous guidance and feedback during the course of this work and thesis writing.

I wish to thank Dr. Stanley Su for his guidance and support during this work. Thanks are due to Dr. Joachim Hammer for agreeing to be on my committee.

I would also like to thank Sharon Grant for making the Database Center a great place to work and being a good friend to all of us here. I thank all my friends at the University of Florida for their support and love.

I would like to thank my parents and brother for their constant encouragement, love and guidance all the time.

TABLE OF CONTENTS

page

LIST OF FIGURES

Figure

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

EVENT HISTORY PROCESSING
IN AN ACTIVE DISTRIBUTED OBJECTS SYSTEM
By

Harini Raghavan

August 1999

Chairman: Dr. Herman Lam
Major Department: Computer and Information Science and Engineering

An Active Distributed Objects System (ADOS) has been developed which incorporates the "active" capabilities from the active database. In the same way that ECA rules can make a database system "active", the event-driven rule technology can make a distributed objects system active through rule-based interoperability.

This thesis describes the design and implementation of an Event History Processor to support an ADOS. The Event History Processor allows the specification and processing of composite events within an active distributed objects environment. Composite events are complex event expressions formed by combining primitive events using temporal and logical operators to specify meaningful preconditions for firing rules and taking actions. In this work, a grammar for composite event expressions has been defined. In the Event History Processor, these expressions are converted into *event graphs* for their evaluation. To support different types of requirements, the event graphs are evaluated based on different *parameter contexts*. To improve performance, the

relevant event graphs are evaluated, as different threads in parallel. Furthermore, the events are stored in a History Database and Logs for subsequent analysis and to enable recovery from system crashes.

CHAPTER 1
INTRODUCTION

Conventional database management systems (DBMS's) provide powerful capabilities to query and manipulate data to support the information needs of an enterprise. However, they are "passive" in the sense that they perform actions only in response to the explicit requests from embedded queries from application programs or ad-hoc queries from users.

1.1 Active Database Management Technology

Active database systems [6, 13, 24] allow users to specify actions to be taken automatically without user intervention when certain events and conditions arise. Typically, the active capability is implemented using ECA/ECAA (event-condition-action/alternative-action) rules. An event specifies the point in time when the DBMS has to react. It could be the state of the database or transition between states (from some INSERT/ UPDATE/ DELETE operation). When an event occurs, a set of rules associated with that event is triggered. The condition part of the ECA rule is evaluated. Based on the result of the condition, either the action or the alternative action is executed.

In many applications, however, the simple event-detection capabilities found in commercial systems are not sufficient. Complex sequences of events must instead be detected as the natural preconditions for taking actions and firing rules. A major thrust in current research in active databases focuses on allowing complex patterns of temporal

events to serve as preconditions for rule triggering [3, 8, 9, 12]. Logs of events can also be maintained to analyze the event patterns and their effect on various recorded observations.

1.2 Active Distributed Objects Technology

Many computer systems in operation today are distributed systems. A distributed system is a dispersed set of application programs communicating with one another over an infrastructure. A *distributed objects system* is a distributed system, which is based on an object-oriented paradigm. Component objects communicate with each other by means of sending remote messages via remote method invocation (e.g., Java RMI) or via some middleware. Middleware for distributed object systems includes CORBA ORB's [15, 17, 18], DCOM [35] and Information busses [32]

Currently, there is an on-going project in the Database Systems R&D Center at the University of Florida to incorporate the "active" capabilities (from the active database technology) into distributed objects systems [19, 20, 21, 22, 23, 26, 29, 30, 31]. An architecture using Event Servers and Rule Servers to support event-triggered rules in an ORB has been implemented using Iona's Orbix Web product. [16]. The Event Service is based on the CORBA Event Service. But it can support both asynchronous and synchronous events containing arbitrary data. In this framework distributed objects can post synchronous and asynchronous events leading to the triggering of rules managed in Rule Servers. Just as ECA rules have made a database system "active", the event-driven rule technology provides the rules-based interoperability to make a distributed objects system active.

1.3 Composite and Temporal Event Processing in ADOS

The idea of composite and temporal event processing in Active DBMS technology and the logging of events can also be adopted in Active Distributed Objects System (ADOS). Composite events are complex sequences of primitive events formed by the combination of various logical operators. Temporal events are primitive events associated with a time component. The detection of composite events makes it possible to have them also as the natural preconditions for taking actions and firing rules. The detection of a composite event entails detecting not only the time at which the composite event occurs, but also the specific constituent event occurrences that make the composite event occur. In addition the persistent store of the occurrences of primitive and composite events helps in the analysis of past data. An expressive event specification language, detection of composite events and logging can be used for analyzing event histories in applications, such as stock trading, trend/demographic profile computation, and auditing.

1.4 Event History Processor

In this thesis, we describe the design and implementation of an ***Event History Processor*** for an Active Distributed Objects System. In the ECA/ECAA paradigm, a trigger generally specifies the event(s) which activate the processing of a set of rules. To include composite and temporal event processing, the trigger now includes the *triggering* event(s) and a *composite event expression*. The occurrence of a triggering event would lead to the verification of the corresponding composite event and, if satisfied, activate the processing of a set of rules.

In the existing Active ORB architecture, a Rule Object Manager in the Rule Server subscribes to the Event Service. When a triggering event occurs, the Rule Object Manager would manage the triggering and execution of a structured set of rules. To include composite and temporal event processing, an Event History Processor component is included in the Rule Server, now renamed to ETR (Event-Trigger-Rule) Server. When a triggering event occurs, the Rule Object Manager queries the Event History Processor to verify the corresponding composite event expression. If the expression is satisfied, then the Rule Object Manager will trigger and manage the execution of a structured set of rules.

In this work, a grammar for the composite event expression has been defined. In the Event History Processor, these expressions are converted into *event graphs* for their evaluation. The event graphs are evaluated based on one of the *parameter contexts* that have been implemented [12]. The Event History Processor subscribes to all the events that are specified in the system's composite event expressions. When the Event History Processor receives notification of such an event, it is fed into the event graphs for processing. Furthermore, the event is stored in a database for subsequent analysis in applications and for support of recovery from system crashes.

1.5 Organization of Thesis

This thesis is organized as follows. The next chapter contains the survey of related work viz, ODE, SAMOS, Sentinel, Event Pattern Language and other composite event related work. Chapter 3 describes the semantics of composite events. Chapter 4 presents the general architecture of the Active Distributed Objects System, overview of the ETR

Server and the general architecture of the Event History Processor. In Chapter 5 the detailed design of the Event History Processor is given, describing the build time and the runtime activities. Chapter 6 describes the implementation details. The summary and conclusions are discussed in Chapter 7.

CHAPTER 2
SURVEY OF RELATED WORK

In this chapter, the work related to composite event processing is surveyed.

2.1 ODE

ODE is a database system and environment based on the object paradigm. The database is defined, queried and manipulated in the database programming language O++ that is based on C++. The constraint and trigger facilities [6, 7] in ODE makes it an active database. It provides facilities for object oriented databases that can be used to specify complex and higher level integrity constraints. Constraints are used for maintaining object consistency applicable to all instances of the class in which they are declared. Triggers monitor the database for some conditions. It consists of a condition and an action.

Ode defines an event occurrence as a tuple of the form (primitive event, event-identifier). Event identifiers (eid) are used to define a total ordering on event occurrences. An event history is a finite set of event occurrences in which no two event occurrences have the same event identifier. An event expression (a finite set of primitive or composite events) is formed using primitive events and several operators discussed in [8]. Ode implements event expressions using finite automata. The history in the context of which an event expression is evaluated provides the sequence of input symbols to the automaton

implementing the event expression. The automaton is fed as input the primitive event components from event occurrences in the history, one at a time, (in eid order) as they occur. If, after reading a primitive event, the automaton enters an accepting state then the event implemented by the automaton is said to take place at the primitive event just read. The history need not be known in its entirety a priori, and the automaton can be used, in "real time", as a triggering device.

Finite State Automaton (FSA) does not support variables, and thus unlike our system, they cannot detect parameterized events, which are essential in most real life applications. It is suggested in [8] that parameterized events can be handled by using relational automata. A relational automaton is a FSA, with the addition that each one of its states is augmented with data structures that maintain partial variable bindings for the composite event that the FSA implements (detects). However, the resulting model surrenders the initial simplicity and intuitive appeal of FSA, without providing fail proof formalism for the detection of composite events. For instance, the semantics of negation when attributes are involved remains a problem. In case of an event occurrence each constraint and trigger has to be evaluated, i.e., each finite automaton constructed has to be checked to see if there are any transitions. This leads to excessive checking. In our system when an event instance occurs, only those expressions containing that event type are processed. Hence it is different from Ode where each automaton has to be checked to see if there are any transitions.

2.2 Sentinel

The Sentinel architecture extends the passive Open OODB system. It supports the following features. i) Detection of primitive events, ii) Detection of composite events, iii) Parameter computation of composite events, and iv) Clear separation of composite event detection with application execution. For details of Sentinel architecture refer to [1].

SNOOP is the event specification language used in Sentinel. Events are classified as primitive and composite events. Primitive events are those that are predefined in the system. They include database events, temporal events and explicit events. A composite event is defined by applying an event operator to constituent events that are primitive or other composite events. Various event operators have been discussed in 4. Composite events are represented as a set of constituent event occurrences within which the order of event occurrences is preserved. The same constituent event occurrence can be used for more than one occurrence of a composite event. The time of occurrence of a composite event is the time of occurrence of the last constituent event.

A novelty offered by Snoop is the concept of parameter contexts. This feature provides the ability to interpret temporal sequences of events in different ways, and thus, it facilitates the precise modeling of a wide range of applications. We present a brief description of the various parameter contexts below.

Recent Context: In this context, the last occurrence of an event (simple or composite) subsumes the effect of all previous occurrences of the same event. Thus, at each point in time, only the most recent occurrences of the various event types of interest are maintained.

Chronicle Context: In the chronicle context, when a composite event is detected, its parameters are computed by using the oldest occurrence of each constituent event. This description implies that a particular simple event occurrence can participate in at most one instance of the composite event.

Continuous Context: Unlike the chronicle context, many instances of a composite event may occur at the same time point in the continuous context. However, at this point, every basic event that participated in one of these instances is deleted from storage, and thus, it cannot be part of any future instance of the composite event.

Cumulative Context: In this context, for each constituent event, all occurrences of the event are accumulated until the composite event is detected. Whenever a composite event is detected, all the constituent event occurrences that are used for the composite event are deleted. Unlike the continuous context, an event occurrence does not participate in two distinct occurrences of the same event in cumulative context.

The approach taken for that of composite event detection is that of Event Graphs. An event tree for each composite event is used and these trees are merged to form an event graph for detecting a set of composite events. Primitive event occurrences are injected at the leaves and flow upwards analogous to a data-flow detection. All event propogations are done in a bottom up fashion. The leaves contain the constituent events and the internal nodes the operators. The leaves of the graphs have no storage and pass the primitive events directly to the parent nodes. The operator nodes have separate storage for each of their children.

Our implementation follows the approach of event trees for composite event detection. We have also adopted the concept of parameter contexts to evaluate the composite event occurrences.

2.3 SAMOS

SAMOS (Swiss Active Mechanism-Based Object Oriented Database System) is an active object oriented database system. It defines a mechanism based on Petri nets for the modeling and detection of composite events. Samos [9] addresses event specification and detection in the context of active databases.

The most distinguishing feature of SAMOS is its event detection mechanism, which is based on colored Petri nets [11] and modified to so-called SAMOS Petri nets. Colored Petri nets were chosen as they could carry complex information through the Petri net. Hence, not only information about the occurrence of an event flows through the Petri net but also the event parameters.

A detailed presentation of the event language is given in [10]. Here we give a brief description of Samos composite event detection borrowed from [3]. Refer to [9] for a detailed description. For each primitive composite event construct, an appropriate Petri net structure has been defined. The Petri net for a complex event pattern can then be built by combining the Petri net structures of its component events. A place in a SAMOS Petri net models an event type, basic or composite. It may contain one or more tokens, which correspond to distinct occurrences of this event type. The token type for a place specifies the variable signature of its event type. A place is marked if it contains at least one token, and multiple tokens are stored in a FIFO queue. Input places are used to model the

component event patterns, while ouput places are used to model the composite event patterns.

Event composition is accomplished by connecting places via transitions. A transition connects its input places to its output place. When all input places of a transition t are marked, the transition fires and from each input place, the token with the oldest timestamp (at the head of the FIFO queue) is removed. All these tokens are then matched to compose a new output token, which is deposited into t's output place. The newly created token may in turn cause the firing of another transition, and so on. Thus, the arrival of a new token (event occurrence) in a Petri net could cause a sequence of transition firings. We can see that the event detection mechanism of Samos follows the c*hronicle* parameter context discussed in the previous section.

2.4 The Event Pattern Language (EPL)

EPL is an active rule language designed and implemented at U.C.L.A., which supports the specification of composite events. [3] Composite events are constructed from simple events, using various constructs, such as sequence, repetition, conjunction, disjunction and negation. EPL has a rule-based syntax and has been designed as a portable front end to active relational databases supporting only simple event detection. It uses Datalog_{1S} [33] a temporal extension of Datalog, by allowing every query to have (at most) one temporal argument in addition to the usual data parameters. Hence it provides for a natural and powerful basis for defining the semantics of composite-event-triggered database rules. Two prototypes of EPL were developed, the first in LDL++ and the second in CLIPS.

EPL consists of basic events, that are simple database table modification commands. EPL can handle simultaneous events, i.e. multiple events with the same timestamp. Each basic event occurrence is time-stamped by the underlying database system. Composite events are patterns of simple events. A variety of constructs that are required for the formations of composite events have been provided. The user is referred to [3] for details.

Two different implementation approaches that facilitate the computation of the produced $Datalog_{1S}$ rules in an incremental and history-less manner are given. The first approach sees the detection of a composite event as proceeding through a sequence of states as in a Finite State Automaton. In this regard, $Datalog_{1S}$ rules can be viewed as defining transitions between two states. Every time a basic event occurs, a new stage is added to the system, simultaneous basic events create a single new stage. At every stage, the state of all satisfaction predicates is updated by using (a) the state of the satisfaction predicates in the previous stage, and (b) the new basic event(s).

More complicated patterns such as disjunctions, conjunctions and negated events can also be implemented by using additional state tables and propagating changes to composite event state tables as indicated by the corresponding $Datalog_{1S}$ rules. However, this approach requires a specialized implementation scheme for each construct supported by the language. Thus, while it can be highly optimized, it may also require significant development effort.

The second approach is probably simpler, and consists in implementing $Datalog_{1S}$ rules directly in an existing rule processor. Using a general-purpose deductive database system, such as LDL++, the $Datalog_{1S}$ rules can be translated almost verbatim, but for an

efficient implementation, the compiler must take advantage of the particular syntactic structure of these rules, so that their execution is optimized. For further details refer [3].

Like the EPL, our implementation for composite event processing can also handle *simultaneous* events. Each event occurrence that happens in the system is timestamped and sent.

2.5 Other Composite Event Related Work

Distributing ECA rules to different sites of a distributed system brings considerable difficulties in comparison to ”centralized” ECA rules. [16] looks at composite events in a distributed system. The assumption is that the constituent basic events of a composite event occur at different sites in the system. A certain site is responsible for monitoring a certain rule and therefore for detecting the corresponding composite event. This site is called the observer site. On the one hand the observer site detects relevant events locally and processes them and/or sends them to remote ”interested” sites; on the other hand it receives relevant events from remote sites. Events are first put into an event queue at the local rule monitor and then processed. One problem tackled is that the arrival order of events at the local rule monitor does not coincide with their occurrence order (in general). Another issue considered is that events on different sites can occur ”in parallel” (no global time in distributed systems; approximately synchronized clocks) and therefore cannot be totally ordered. Hence the local event monitor consists of a stabilizer, which sorts incoming events (topological sort) and an event detector, which detects composite events from the sorted stream. The topics

addressed in [16] are naming of basic events, construction of composite events, and detection of composite events.

CHAPTER 3
EVENT-TRIGGER-RULE TECHNOLOGY

Rules in active database systems are generally specified in the form of ECA (Event-condition-action) or ECAA (Event-condition-action-alternativeAction) rules. In our Active Distributed Objects System, the ECA/A paradigm is generalized into an event-trigger-rule (ETR) paradigm. The focus of this thesis is the event component of the ETR server. The remainder of this thesis describes the design and implementation of an Event History Processor, which processes composite and temporal events. Before we do so, let us first describe this ETR paradigm and compare it to the more traditional ECA/A paradigm. In particular, the semantics of composite events is discussed in detail.

3.1 Event-Condition-Action/Alternative Action Paradigm

The general syntax of a rule language for ECAA rules is shown in the Figure 3-1. The *Event* clause provides the specification of the triggering event. It includes a *coupling mode* and the *triggeringEvent*. *CouplingMode* specifies the time in which the rule is to be triggered relative to the triggering event. Example coupling modes include *Before, After, On, OnCommit, InsteadOf and Detached.* The *triggeringEvent* can be any legitimate event that is defined for the data model/language for which the rule language is associated.

When an event occurs, a set of rule bodies associated with that event is triggered and the corresponding code is executed to perform possible actions. A rule

specification consists of an optional *Condition* clause, and one or both of the *Action* and *AlternativeAction* clauses. The *Condition* clause is used to determine under what condition an action needs to be taken once the rule body has been triggered. It can be used to verify the values of incoming data/objects, the internal state of the object in question, or its relationship to other objects. In general, the *condExpression* can be any expression in the rule language that returns a TRUE or FALSE, including function or method calls. If the condition evaluates to TRUE, then the operations in the *Action* clause are performed. If the condition evaluates to FALSE, the operations in the *AlternativeAction* clause are performed. *Operations* can be a sequence of any legitimate operations, including function or method invocations.

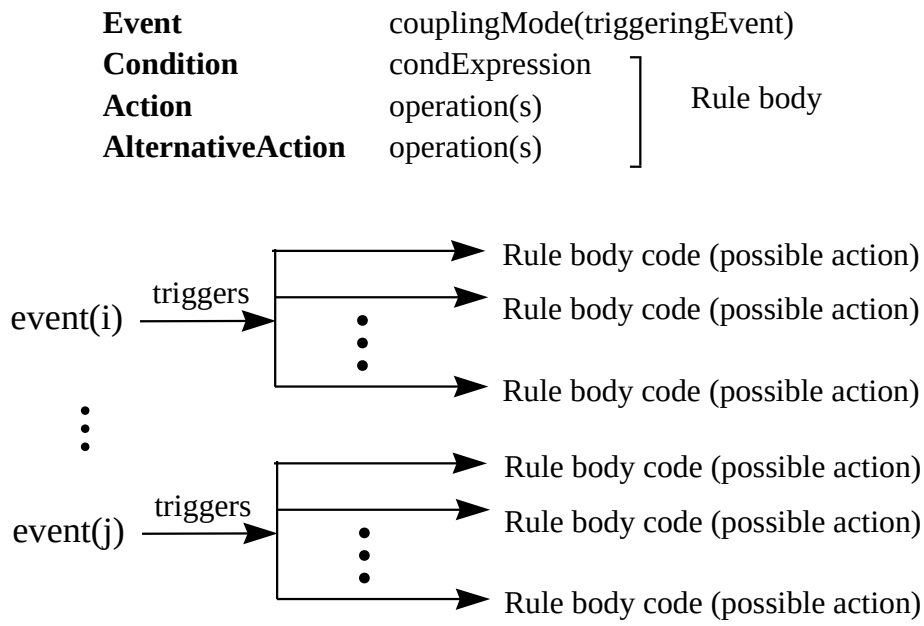

Figure 3-1 ECA/A Rules

3.2 Event-Trigger-Rule Paradigm

The Event-Trigger-Rule (ETR) paradigm is a generalization of the ECA/A paradigm. In the ECA/A paradigm, the event, the trigger, and the rule information are specified together as an ECAA rule. In the ETR paradigm, they are separated into three separate specifications:

- Event specification, which includes the event name and the list of event parameters.
- Rule specification, which includes the Condition, Action, and Alternative Action clauses found in ECAA rules.
- Trigger specification, which relates events with rules.

A trigger specifies how an event structure will trigger a structure of rules. The overall syntax for a trigger specification is as follows:

```
TRIGGER          trigger_name ( trigger_parameter_list )
TRIGGEREVENT     events connected by OR
COMPOSITEEVENT   event expression
RULESTRUC        structure of  rules
RETURNS          return_type : return_rule
```

The TRIGGER clause consists of a unique trigger name, and has a list of trigger parameters.

The TRIGGEREVENT clause specifies a number of alternative events, each of which when posted would trigger the evaluation of the composite event specified in the COMPOSITEEVENT clause. Each trigger event has a list of event parameters. The parameters should correspond to each of the trigger parameters defined in the TRIGGER

clause. In other words, each parameter list must have the same number of parameters as the trigger parameter list, and they must have the corresponding data types. For example assume that events E1 and E2 have the following parameter lists:

E1(int j1, objectX j2, String j3)

E2(int k1, objectX k2, String k3)

Then E1 and E2 can be specified as trigger events of the trigger ‘sample_trigger’. Here, v1, v2 and v3 correspond to j1, j2 and j3 of E1, and k1, k2 and k3 of E2, respectively.

TRIGGER sample_trigger (int v1, objectX v2, String v3)

TRIGGEREVENT E1 OR E2

The trigger events can only be connected with **OR** and is kept simple purposely as complicated event expressions can be specified in the composite event clause if required.

The COMPOSITEEVENT clause is a Composite Event expression formed by combining a set of primitive event expressions, using logic operators and temporal event operators. If the composite event evaluates to True, the structure of rules specified in the RULESTRUC clause is processed. Otherwise, it is not processed. A composite event expression states the inter-relationship of a number of events that have been posted. For example, we may want to trigger the rules if “E1 and E2 but not E3” have been posted or if “E5 occurred before E4 within a specified time window” have been posted.

The separation of TRIGGEREVENT and COMPOSITEEVENT provides a way to specifically name the events that will trigger the evaluation of more complex event expressions. This is different from the event specification of some existing ECA rule systems in which, when a composite event is specified all the events mentioned in the

composite event implicitly become the trigger events. In some application, one may want to specify that only the posting of E2 to trigger the evaluation of “E1 and E2 but not E3”. Similarly, one may want only the posting of E4 should trigger the evaluation of “E5 occurred before E4 within a specified time window”. The separation of TRIGGEREVENT and COMPOSITEEVENT allows more explicit specification of what triggers the evaluation of a composite event.

The TRIGGEREVENT or COMPOSITEEVENT can be omitted, but not both. If the TRIGGEREVENT is omitted, the default is the OR of all the events referenced in a COMPOSITEEVENT expression. Details of the composite event expression will be given in the subsequent sections of this chapter.

The RULESTRUC clause contains a structure of rules that is processed when the event specification is satisfied. It specifies the rule execution order and maps the parameters of the events to the individual rules. The rules can be executed sequentially, or in parallel. A number of rules together can implement a larger granule of logic and control. There are two basic operators for specifying the execution order. The first operator ‘>’ is used to specify a sequential order of rule execution and the second operator ‘,’ is used to specify a parallel execution. For example, the following expression means that rules R1, R2, R3 and R4 are to be executed sequentially following the specified order.

R1 > R2 > R3 > R4

Note that the ‘>’ operator can be cascaded. It is also straightforward to specify a parallel execution of rules. The following example shows that rules R1, R2, R3 and R4 are to be executed in parallel.

(R1, R2, R3, R4)

These two operators can be used in combination to specify a rule structure of arbitary complexity:

R1 > ((R2 > R3, R4), R5)

The RETURNS clause specifies which rule in the RULESTRUC would **return** an object after executing the rule structure. This returned object is returned to the event if the event was posted synchronously to cause the processing of the trigger.

A sample specification of a trigger is shown below.

TRIGGER	sample_trigger (int v1, int v2, objectX v3)
TRIGGEREVENT	E1 OR E2
COMPOSITEEVENT	BETWEEN [06 06 1999 9 20 0] [06 10 1999 0 0 0] (E5 > (E3 AND E6))
RULESTRUC	R1(v1,v2) > R2(v3,v1) > R3(v1,v2,v3)
RETURNS	int : R1

The trigger ‘sample_trigger’ contains three parameters. It is triggered when either E1 or E2 occurs. The occurrence of either triggering event will result in the evaluation of the composite event expression. The expression will evaluate to true, when an instance of E5 occurs before the occurrences of the instances of both E3 and E6 in the specified time interval. If the event history expression evaluates to true, the rule structure is processed. The rules R1, R2 and R3 are executed sequentially in that order. In the example, R1 uses event parameter v1 and v2, R2 uses v3 and v1, and R3 uses v1, v2 and v3. The original values of event parameters that are passed to the trigger are passed to the individual rules. The above trigger returns an integer value that is generated by R1.

3.3 Operator Semantics

We saw that the composite event expression is formed by combining primitive events with a set of event operators. Below we describe the operators and their semantics. We will use E (upper case alphabets) to represent an event expression as well as an event type and e (lower case alphabets) to represent an instance of the event E.

An event E (either primitive or composite) is a function from the time domain onto the boolean values, True and False.

$E(t) \rightarrow \{True, False\}$

given by

$$E(t) \begin{cases} = & T\text{(rue)} \quad \text{if an event of type E occurs at time point t} \\ & F\text{(alse) otherwise} \end{cases}$$

Any time component specified in these operators is represented in the format [yyyy mm dd hrs min sec].

The event operators and the semantics of composite events formed by these event operators are as follows:

1. OR (|): Disjunction of two events E1 and E2 denoted as E1 | E2 occurs when E1 occurs or E2 occurs.
2. AND (&): Conjunction of two events E1 and E2, denoted as E1 & E2 occurs when both E1 and E2 occur, irrespective of their order of occurrence.
3. ANY: The conjunction event, denoted by ANY{m, E1 , E2 , ... En } where m <= n, occurs when m different events out of the n distinct events specified occur, ignoring

the relative order of their occurrence. If any of the Ei's is a composite event, they need to be enclosed in brackets.

4. NOT (!): The negation operator is denoted as !E1. This is true when the event E1 does not occur.
5. SEQUENCE (>): Sequence of two events E1 and E2 denoted E1 > E2 occurs when E2 occurs provided E1 has already occurred. This implies that the time of occurrence of E1 is guaranteed to be less than or equal to the time of occurrence of E2. For this event to fire true, both the events need to occur. If E2 occurs before the occurrence of E1 then that event occurrence of E2 is ignored.
6. AT: The AT event, denoted AT [time] E1, occurs when the event E1 occurs or fires at the specified time.
7. BY: The BY event, denoted BY [time] E1, occurs when the event E1 occurs or fires by the specified time.
8. BETWEEN: The BETWEEN event, denoted BETWEEN [start time] [end time] E1, occurs when the event E1 occurs or fires in the interval specified between start and end time.

The Grammar for these operators, that specifies the precedent relationships between them has been designed as follows:

Expression	→	Expression > Term
	→	Term
Term	→	Term | Term1
	→	Term1
Term1	→	Term1 & Factor

	→	Factor
Factor	→	! Primary
	→	ANY ‘{‘<integer>’ ‘,’ Primary (‘,’ Primary)$^{+}$ ‘}’
	→	AT ‘[‘<date>‘]’ Primary
	→	BETWEEN ‘[‘<date>‘]’ ‘[‘<date>‘]’ Primary
	→	BY ‘[‘<date>‘]’ Primary
	→	Primary
Primary	→	‘(‘ Expression ‘)’
	→	Name
Name	→	<identifier>

It can be seen that the order of increasing operator precedence is SEQUENCE, OR, AND, {ANY, NOT, AT, BY, BETWEEN}. We believe that the above set of event operators define an event specification language that meets the requirements of a large class of applications.

Let us show some examples to see how composite expressions are formed using the above operators. We use a simplified syntax to make the events readable.

1. Check if any of the Toyota car models have been sold.

   Composite Event: ANY (1, Toyota_Camry, Toyota_Tercel, Toyota_Corolla)

2. Check if ticket reservation and car reservation have been confirmed

   Composite Event: CarReservationConfirmation AND TicketReservationConfirmation

3. Check if special flight fares will be notified by a certain date

   Composite Event: BY [07-14-1999] (SpecialFlightFareEvent)

3.4 Event History

So far we have defined the semantics of event operators. The detection of a composite event involves the determination of the specific event occurrences that make the composite event occur. Semantics of composite events are defined over an event history. The notion of event histories has been introduced for defining the computation of participating events for a composite event expression. Let us define the following:

- Global Event History: It is the ordered set of all primitive event occurrences
- Primitive Event History: It is the set of all occurrences of the primitive event type E present in the global history.
- Composite Event History: A set of primitive event occurrences, that form a composite event occurrence form a single component of the composite event history. A set of such components forms the composite event history.

For example consider the following expression (E1 & E2) | E3. An occurrence of an event type Ej is denoted as e_{ji} i.e., the i^{th} occurrence of the event Ej, where 'i' indicates the relative time with respect to other occurrences of the same event. Let us assume that the primitive event instances occur in the order e_{11}, e_{21}, e_{31}.

When e_{11} and e_{21} occur, (E1 & E2) evaluates to true. The expression flags true if either (E1 & E2) or E3 becomes true. So the expression A evaluates to true. The occurrence e_{31} will cause A to be true once more.

- Here the global event history is { e_{11}, e_{21}, e_{31} }

- There are three primitive event histories

  E1 - { e_{11} }

  E2 – { e_{21} }

  E3 – { e_{31} }

- The composite event history for this expression is { { e_{11}, e_{21} }, { e_{31} } }

3.5 Parameter Contexts

The event expressions associated with a trigger must be evaluated as event instances occur. Events requiring multiple event occurrences (either of the same type or of different types) for the detection of a composite event give rise to alternate ways of computing the expression, as the events are likely to occur multiple times over an interval. The use of parameter contexts [12] serves the purpose of detecting and computing the parameters of composite events in different ways to match the semantics of applications.

Parameter contexts essentially delimit the events detected, parameters computed, and accommodate a wide range of application requirements. The choice of a parameter context also suggests the complexity of event detection and storage requirements for a given application. The parameter contexts help in restricting the occurrences that make a composite event occur as well as for computing its parameters.

Two parameter contexts described in [12] have been adopted in our system, with slight modification.

- Most Recent
- Chronicle

In the Most Recent context, only the most recent occurrence of any primitive event is used. If another occurrence of the same event type occurs, before the composite event is fired, the previous instance is invalidated and the new occurrence is used for the evaluation of the expression. This semantics is useful in applications where the events are happening at a fast rate and multiple occurrences of the same type of event only refine the previous data value. In other words, the effect of the occurrence of several events of the same type is subsumed by the most recent occurrence. This is typical of sensor applications (e.g., hospital monitoring, global position tracking, multiple reminders for taking an action).

In the Chronicle Context, event occurrences are added to the event type list as they occur and do not replace an existing occurrence as the most recent semantics. The event occurrences are grouped together in chronological order and contribute towards the occurrence of a composite event. This context is useful for applications where there is a correspondence between different types of events and their occurrences and this correspondence needs to be maintained. In an expression it may be such that, one of the constituent events might occur many times during a certain interval and the other at another period. So we may need to match the earliest occurrence of one with the other.

3.6 Illustration of Parameter Contexts

In this section we will illustrate through an example how the constituent events of a composite event is computed for each of the parameter contexts. The expression used in this example is E1 > (E2 & E3). The global event history [Figure 3-2] indicates the

relative order of the primitive events with respect to their time of occurrences. Here, e_{ji} represents the i^{th} occurrence of the j^{th} event.

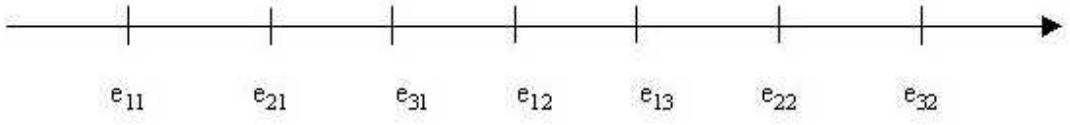

Figure 3-2 Global Event History

We illustrate the processing of each parameter context:

- Most Recent Context: The occurrence e_{11} makes the left-hand side operand of the SEQUENCE (>) operator true. The occurrence e_{21} followed by that of e_{31} makes the expression (E2 & E3) evaluate to true, which is the right side operand of SEQUENCE operator. Since E1 evaluates to true before (E2 & E3), the SEQUENCE operator evaluates to true and these three occurrences form an occurrence of the composite event. The second event instance e_{12} makes E1 true. The third event instance e_{13} replaces the previous occurrence and also evaluates E1 to true. The occurrences, e_{22} and e_{32} evaluate E2 & E3 to true. The occurrences, e_{13}, e_{22} and e_{32} fire the second occurrence of the composite event.
- Chronicle Context: Here the occurrences, e_{11}, e_{21}, e_{31} cause the trigger to fire just like the most recent semantics. Next when e_{12} occurs its added to the list of occurrences of E1. The occurrence e_{13} is added as a second member to the list of occurrences for E1. Subsequent occurrences of the same event are added and don't replace the previous occurrences like in most recent semantics. The occurrences e_{12}, e_{22}, e_{32} group together

to fire the second occurrence of the composite event. The occurrence e_{13}, will be part of the constituent events that will cause the next occurrence of the composite event.

From the above we see that the recent semantics require a fixed size buffer of one to store the truth-value and the timestamp corresponding to each node. It requires the minimum storage. For the chronicle semantics, the buffer size increases dynamically and a queue is required. The amount of storage needed is dependent upon the duration of the interval of the composite event and the frequency of event occurrences within that interval and also on the order of occurrence of the instances during that interval.

3.7 Summary

The advantages of an expressive event specification language and maintaining event histories can be summarized as follows:

- Multiple events can now be used for the activation of rules.

Previously the trigger specification had just one triggering event or a disjunction of events for the triggering of rules. For example:

Trigger: Car Reservation Confirmation ***OR*** Ticket Confirmation

Composite Event: Car Reservation Confirmation ***AND*** Ticket Confirmation

With the above trigger present the occurrence of the event will check the event history to see if the other event has occurred and trigger the rule to go ahead with reservations.

- A variety of operators have been provided for the specification of relationships among events.

Complex event expressions can be formed using the set of operators defined and can be used as the composite event expression for a triggering event.

- The Triggering Event and the composite expression part are separate, hence it is more flexible.

All the events of an event expression need not be responsible for the triggering of the composite event.

- The triggering of rules based on the evaluation of composite expression simplifies the rule logic.

Using the same example above if composite expression evaluation had not been present, the rule that is triggered needs to do the checking for the occurrences of the event. Assume that Car Reservation Confirmation event occurs first. The rule that it triggers must check if the other event has occurred and if it has not, it must note down the fact that the current event has occurred. So that the next time when the other event (Ticket Confirmation) occurs it knows that the previous event had occurred. So now the rule becomes messy and has to keep track of the events. The CONDITION clause of the rule will have additional logic that is not relevant to the rule.

CHAPTER 4
ARCHITECTURE OF AN ACTIVE DISTRIBUTED OBJECTS SYSTEM

4.1 General Architecture

In a distributed, heterogeneous computing environment, a number of computers are interconnected by a LAN or WAN. In a Distributed Objects System each computer may have a number of component systems that are modeled as distributed objects. These component systems communicate with one another via method calls through a common communication infrastructure (e.g., an ORB). In an Active Distributed Objects System (ADOS), in addition to direct method calls, components can communicate via events that can trigger rules to take action without intervention from the user or an application program. Shown in Figure 4-1 is the general architecture of an ADOS implemented in a CORBA environment. The components present in the system are:

- Component Systems: These are various objects present in the system that need to interact with one another. In event driven interactions, they form the suppliers and consumers of events.
- CORBA Event Server: It provides asynchronous event service. Events are posted by the suppliers to the event service and are passed on to the consumers who have subscribed to the event.

- Event-Trigger-Rule (ETR) Server: It is a consumer of all events involved in the triggering of rules. It processes the incoming events and manages the firing and execution of rules.

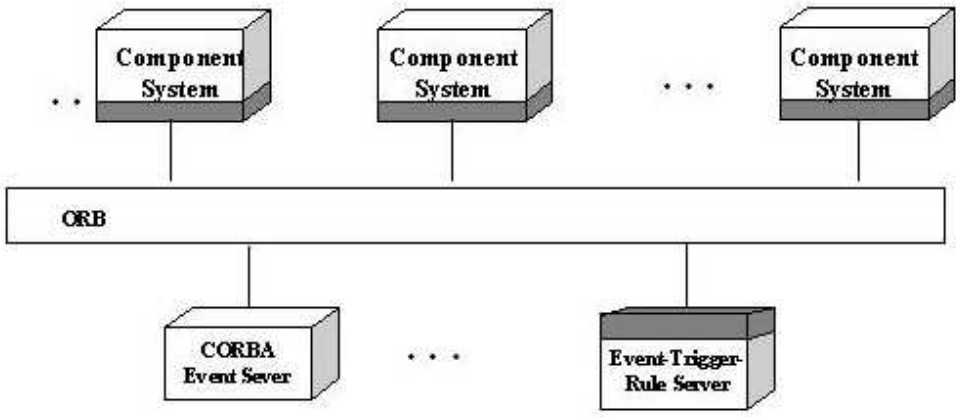

Figure 4-1 General Architecture of ADOS

In an ADOS the two types of events are distinguished:

- Explicitly Posted Events
- Method Triggered Events

Explicitly posted events represent enterprise events that occur within a component system that may be of interest to other component systems. The "supplier" of the event is a component system that is responsible for posting the event, but is not concerned with the handling of the event. Explicitly posted events can be synchronous or asynchronous. Shown in Figure 4-2 is the run-time interaction for an asynchronous explicitly posted event. Consumers interested in the occurrence of certain events have to register themselves with the CORBA Event Server. When a supplier (client) posts an event (label "1" in Figure 4-2), the responsibility of the CORBA Event Server is to notify all the

"consumers" that have been registered for that event, including the ETR Server as shown in Figure 4-1 (labeled "2"). The ETR Server triggers all the rules associated with that event and performs the required action (label "3")

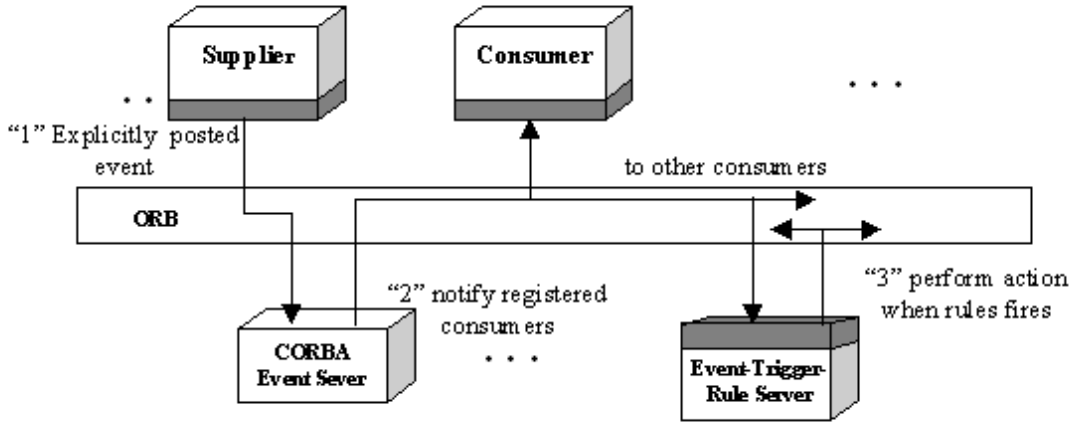

Figure 4-2 Explicitly Posted Event

Method triggered events are associated with the invocation of methods. Methods represent the functional behavior of a server object. Events can be defined for *before, after, detached, instead of,* etc. the execution of a method to provide "hooks", to trigger a set of rules that can be used to customize the behavior of a server object to satisfy some integrity constraints or business policy. Method triggered events can also be either asynchronous or synchronous. In the figure 4-3, a component system calls a method of another component system (label "1"). In this case a synchronous event is sent to the ETR Server to trigger a set of rules (label "2"). Again the ETR Server triggers all the rules associated with that event and performs the required actions (label "3"). In case of method-triggered events, the synchronous events can be used for *before, after, and insteadOf* coupling modes.

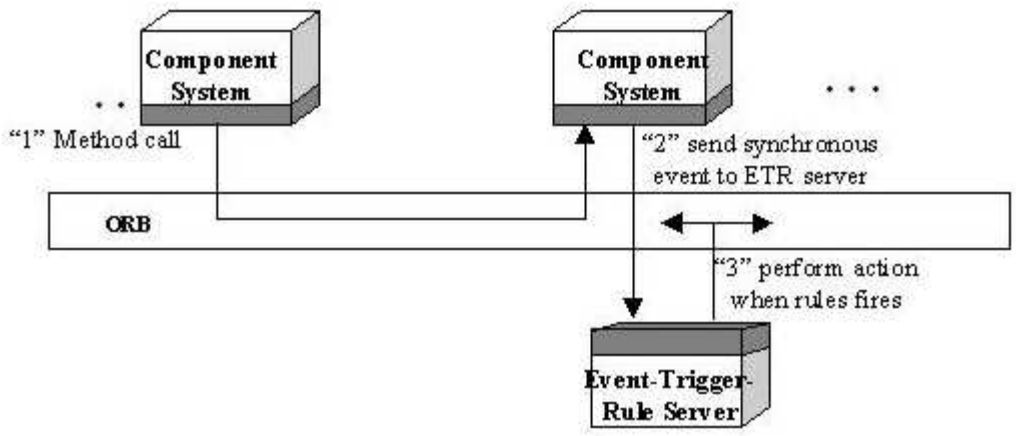

Figure 4-3 Method Triggered Event

4.2 ETR Server

The ETR Server is the key component in supporting an ADOS. It processes the incoming events (including composite and temporal events) and manages the firing and execution of the triggered rules. The general architecture of the ETR Server is shown in Figure 4-4. The following are the components of the ETR Server:

- ETR I/F: Interfaces the ETR Server with the outside. The ETR Server provides the APIs through which trigger definitions could be added/deleted/modified. It also has a key method, notifyEvent to receive the notification of event occurrences.
- Rule Object Manager: Oversees the interactions among the different components and is the controlling component of the system.
- Rule Group Manager: Manages the enabling/disabling of different groups of rules defined. Individual rules in a group can be enabled/disabled.

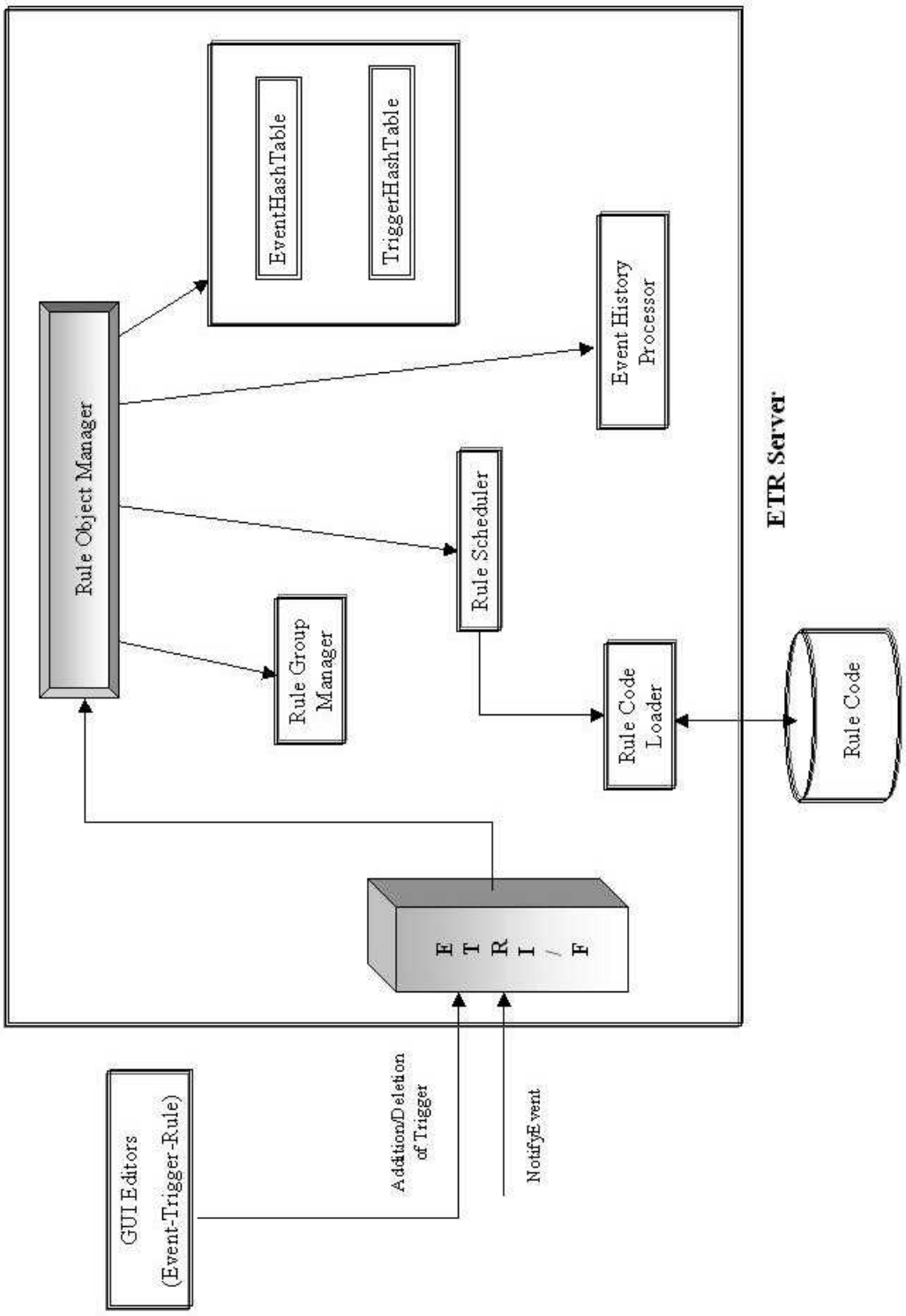

Figure 4-4 ETR Server Architecture Overview

- Rule Scheduler: Schedules the rules present in the rule structure of a trigger and executes them in parallel or serial based on the rule structure specification.
- Rule Code Loader: Loads the rules to be triggered into memory dynamically.
- Event History Processor: Evaluates the composite event expressions of triggers. This component is the main topic of this work and will be discussed in detail in the remainder of this thesis.

Two important tables are maintained to support the functionalities of the ETR Server.

- EventHashTable: Stores information about which events map to which trigger. An event may be a part of more than one trigger.
- TriggerHashTable: Maintains the mapping between the trigger and the corresponding trigger structure. The trigger structure contains the rule structure and the composite event expression.

During the addition or deletion of a trigger, the Rule Object Manager calls methods to update the EventHashTable and the TriggerHashTable to reflect the new information. It calls the Event History Processor to add or delete the composite event expression of the trigger, if an expression is present. The ETR I/f can receive additions and deletions of trigger definitions from any source. (E.g., An Event-Trigger-Rule GUI Editor)

Notification of an Event

An ETR Server registers with an event service for receiving notifications of events. When an event is posted, the ETR Server is notified by the event service with the

posted event instance. The posted event could either be a triggering event and/or an event present in a composite expression. The Rule Object Manager consults the EventHashTable and the TriggerHashTable to make that determination.

If the notified event is an event present in a composite event expression, then the Event History Processor is notified to evaluate the composite event expression. The details of the evaluation are described in the next section.

If it is a triggering event, the trigger structure is retrieved from the TriggerHashTable. If a composite event expression is present, the Event History Processor is queried about the status of the corresponding expression. If the composite event expression evaluated to true or if the triggering event does not trigger a composite expression, the Rule Scheduler is called. The Rule Group Manager is consulted to find out which rules in the rule structure are enabled. The Rule Code Loader loads all the enabled rules dynamically. Then the Rule Scheduler executes the rule structure.

If the triggering event is synchronous event that can trigger rules is posted, the event service is blocked until the evaluation of the rule structure by the ETR Server is complete. In this case, the ETR Server returns the object returned by the rule back to the event service. If the triggering event is asynchronous, the event service is not blocked and the ETR Server returns a null object. In this case, the processing of the expression and rule structure is done asynchronously.

4.3 Event History Processor

The Event History Processor (EHP) is a component of the ETR Server that handles the event processing part. It is responsible for:

- Processing and evaluation of composite and temporal events.
- Persistent storage of the occurrences of “registered” events and composite events. Registered events are primitive events that participate in a composite event expression. The composite events contain references to the constituent event occurrences that formed it.

The general architecture of the EHP is shown in Figure 4-5. The EHP consists of the following modules:

- Definition Processing Module
- Runtime Processing Module
- Startup/Recovery Module

The Definition Processing Module

The Definition Processing Module is responsible for processing the incoming trigger definition from the Rule Object Manager. A trigger definition consists of the trigger name, the composite event expression and the context in which the expression has to be evaluated. Definitions can be added dynamically during runtime.

The given expression is parsed and converted into an internal dynamic structure. The internal structure is dumped in the Log. The Log contains the internal data structures of the EHP used for the purpose of recovery. After the composite event definition has been processed, the Definition Processing Module returns the names of the primitive events present in the composite event expression to the Rule Object Manager. For those events not already registered, the Rule Object Manager subscribes to the event service to receive notifications of those events.

The Runtime Processing Module

The Runtime Processing Module is responsible for evaluating the composite event expressions corresponding to the incoming event instance and storing the primitive and composite event occurrences in the History Database. The History Database contains an Event Table for each primitive event type to store the primitive event occurrences and a Trigger Table for each trigger to store the composite event occurrences.

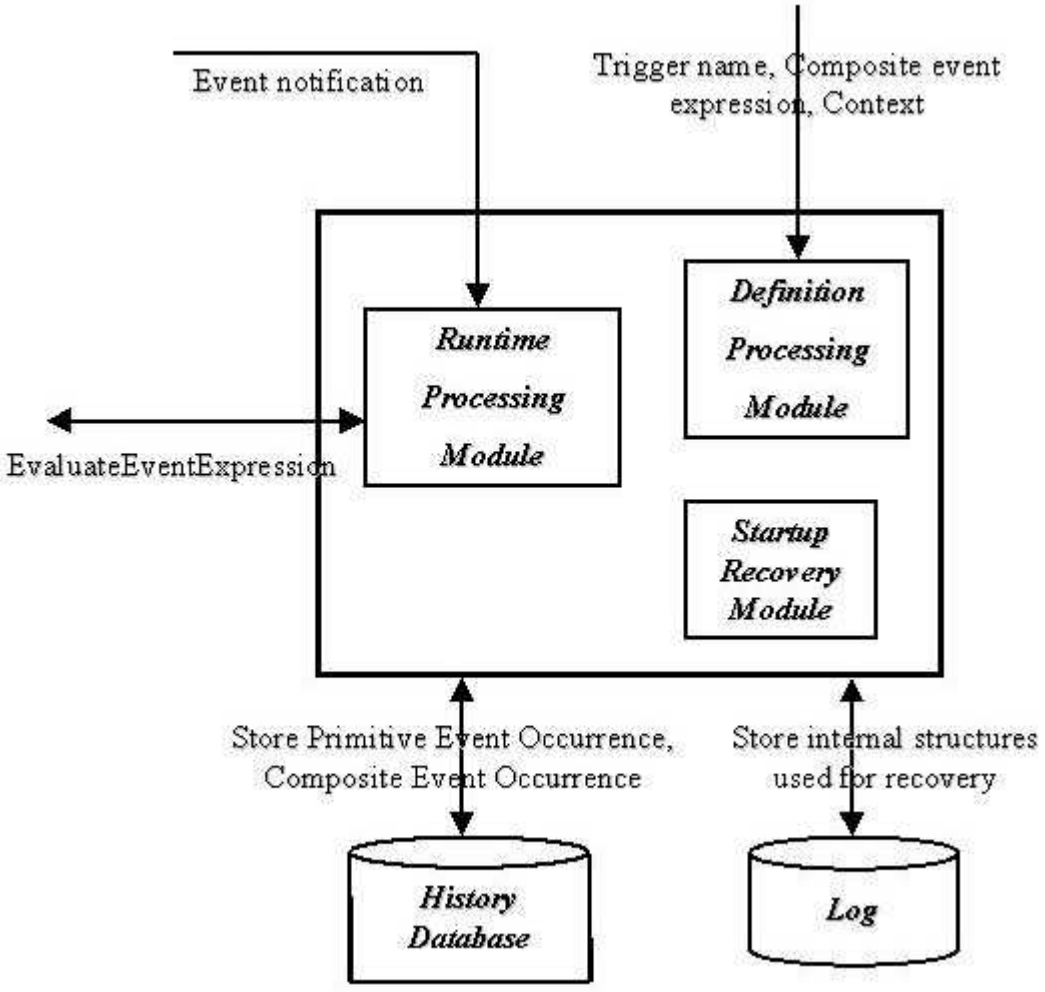

Figure 4-5 General Architecture of the EHP

On receiving an event notification from the Rule Object Manager of the ETR Server, the Runtime Processing Module stores the event occurrence in the corresponding Event Table in the History Database. The notified event occurrence is fed to all the expressions containing the event and these expressions are processed.

If an expression evaluates to true while processing, all the event occurrences that caused the firing of the trigger, as well as the timestamp of firing, are stored in the corresponding Trigger Table present in the database. Before the data is stored into the History Database, the trigger information is dumped into the Log. Note that the query for insertion of an event or trigger into the History Database is dynamically constructed and executed.

Subsequently when the Rule Object Manager of the ETR Server queries about the status of a composite event expression associated with a trigger, the Runtime Processing Module responds with a boolean value as to whether the expression has evaluated to true or false.

Startup / Recovery Module

The Startup / Recovery Module is responsible for retrieving the internal structures of composite event expressions from the Log at startup, and if recovering from system crash, it also restores the system to its previous stable state.

Recall that internal event expression structures of composite *events* are dumped in the Log at the time of definition and also each time when the event expression evaluates to true. For recovery, the internal structures of expressions defined previously are retrieved from the Log first. Then for each trigger, the Trigger Table in the History

Database is checked, and the composite event occurrences that have fired but have not yet been stored in the History Database are found. Such composite event occurrences are stored in the History Database.

Also recall that the internal structure of an expression for a *trigger* is dumped into the Log only when the trigger fires. Hence for recovery, the event tables of the constituent events of the trigger are queried and the event instances notified after the last firing of the trigger are retrieved from the History Database. These event instances are then re-posted locally and the expression is re-evaluated. This is done for each trigger and hence no data (i.e. event occurrence) is lost.

CHAPTER 5
DESIGN OF AN EVENT HISTORY PROCESSOR

In this chapter we describe the architecture of the Event History Processor in detail. Before we describe the run time activities and the design of the different modules of the EHP in sections 5.3, 5.4 and 5.5, we describe in Section 5.1, a set of build time activities that need to be performed before run time. Also in Section 5.2, we describe the approach taken for composite event detection in our work in using Event Graphs.

5.1 Build Time Activities

The purpose of the build time activities is to create the event and trigger tables [Figure 5-1] in the History Database. The Import Module [Figure 5-1] is used to perform the build time activities. The build time activities are:

- Event Import
- Trigger Import

Event Import

After the creation and generation of the event class for an event, the Import module is invoked passing the event name in order to create the event table in the History Database. The event table class is automatically generated. The event table class contains the following fields: an event identifier of type string, a timestamp field of type long, and the event object. The identifier is unique for an instance. The timestamp field indicates time of occurrence of the event instance. The event object holds the event occurrence.

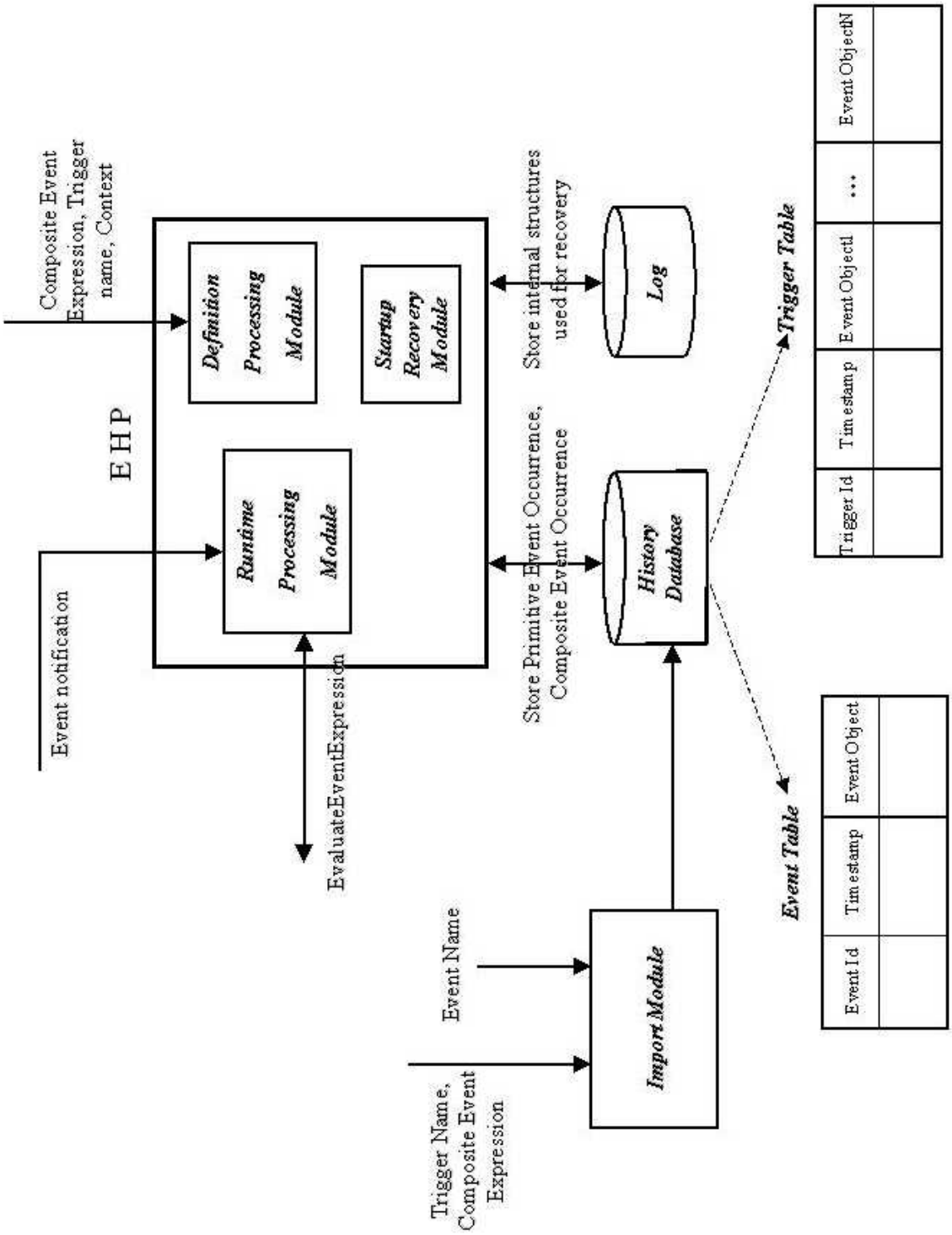

Figure 5-1 Overall View of EHP

The event table class is then compiled. The event class and the event table class are then imported to create the event table in the History Database.

Trigger Import

After the definition of a trigger, the Import Module must be called to create the trigger table in the History Database. The trigger name and the composite event expression are the inputs. A corresponding trigger table class is generated automatically. This class contains the following fields: a trigger identifier of type string, a timestamp field of type long, and an event object for each constituent event type. The identifier is unique for an occurrence of a composite event. The timestamp field indicates the occurrence of the composite event. The references of all event occurrences responsible for the occurrence of the composite events are stored in the corresponding event objects. The trigger table class is compiled and is then imported into the History Database, to create the trigger table in the database.

5.2 Event Graphs

The approach taken for composite event detection in this thesis is that of event graphs. In the following sub-sections, we describe event graphs and show how composite event expressions are detected in various parameter contexts using event graphs.

Event Graphs

An event graph is a binary tree. It is the internal structure that represents a composite event expression. The event types form the leaf nodes and the operators form the internal nodes. Each node contains pointers to two sub event graphs, representing the

operands of an operator. For a leaf node these are null. Each event graph node contains a list that will hold the truth-value(s) about the occurrence of the event instance or the event sub-expression, and also the timestamp.

Unary operators have only one child. Operators with more than two operands are still represented in the binary tree structure form, by introducing false parents. In Figure 5-2 we can see the operator ANY has more than two operands. A false internal node called ANYSUB is used.

The constituent primitive events of a composite event form the leaf nodes of an event graph. The primitive event occurrences are fed to the corresponding leaf nodes

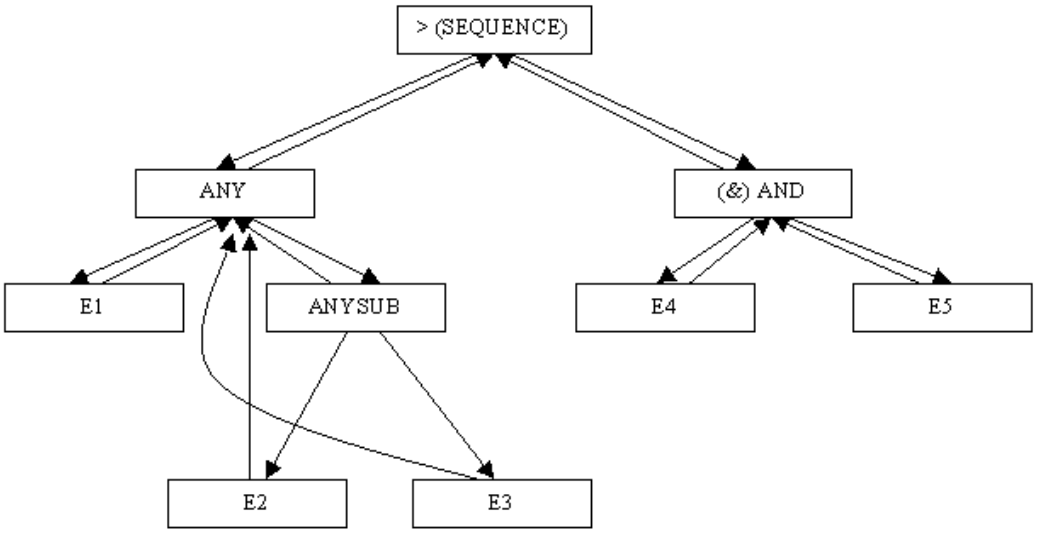

Figure 5-2 Sample Event Graph

followed by the evaluation of its parent node. The parent of each node henceforth is processed until the root node is reached. If the evaluation of an operator does not produce any result, the processing of higher level nodes is not done. For example in the graph

[Figure 5-2] if only an instance of E4 occurs, the AND operator is processed. However, the parent of AND (i.e., SEQUENCE) is not processed because E5 has not occurred, and no value is added to the AND node. Next if an instance of E5 occurs, it will trigger the evaluation of AND once more, and a value is added to the AND node. This results in the evaluation of AND's parent node (i.e., SEQUENCE). The internal nodes hold the truth-value obtained by applying the operator on its children and the timestamp at which the operator is evaluated. The internal nodes maintain a list for holding the evaluated values, up to that node. The leaf nodes also hold truth-values and the timestamp at which the event instances occur. When they occur, we add the boolean value true.

Illustration of Composite Event Detection

We will use an example event graph and discuss how we compute the constituent events of a composite event for each of the parameter contexts discussed in Chapter 3. The time line shown in Figure 5-3 indicates the relative order of the primitive events with respect to their time of occurrences. All event propagations are done in a bottom up fashion. The expression used is E1 > (E2 & E3).

Most Recent Context. As shown in Figure 5-3, e_{11} occurs first. It is added to the leaf node for E1. Its parent '>' (SEQUENCE) is evaluated. But since its right child has not yet occurred, no value is added to it. When event e_{21} occurs, it is added to E2. Its parent node '&' (AND) is processed. But since '&'s right child has not yet occurred, further processing is not done. e_{31} is then added to E3. It also triggers the evaluation of '&' node. The '&' node evaluates to true. Then '>' is processed. Since E1 has occurred before E2 and E3, the composite expression fires. At this point all the values are deleted from the event graph. The next occurrence e_{12} is added to E1. Another occurrence of E1,

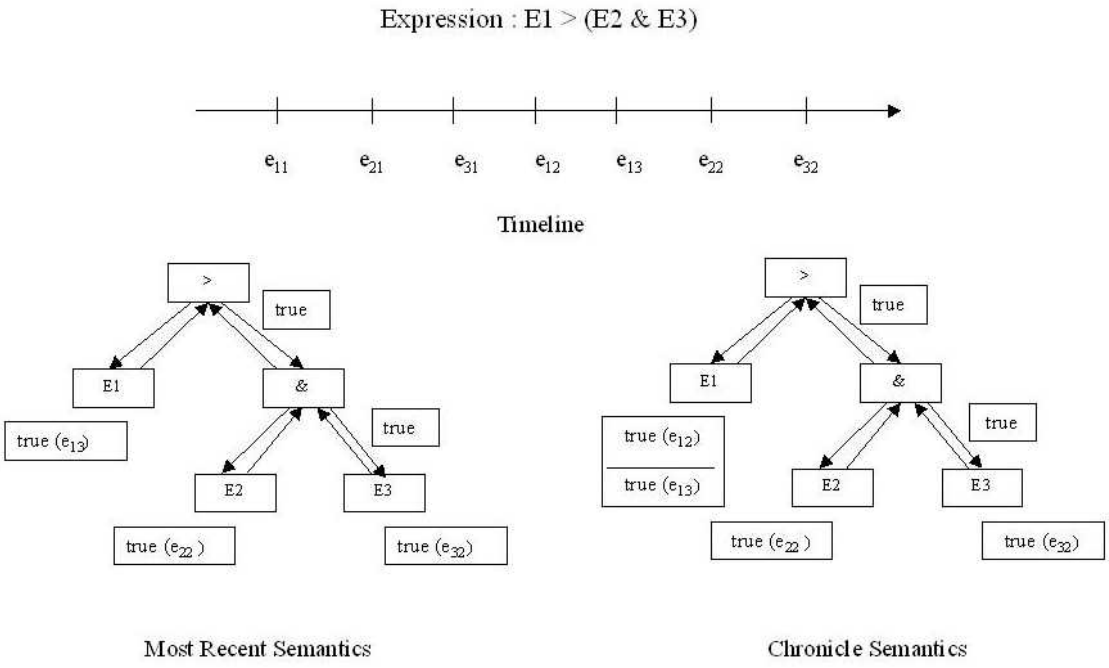

Figure 5-3 MRS & CS Event Graph

e_{13} occurs and this replaces e_{12}. e_{22} and e_{32} occur and the processing is similar to e_{21} and e_{31}. So e_{13}, e_{22}, e_{32} make the composite event true. The result is shown in the right of Figure 5-3.

Chronicle Context. The occurrences, e_{11}, e_{21}, e_{31} cause the trigger to fire just like the recent semantics. Next when e_{12} occurs, it is added to the list of occurrences for E1. Also its parent '>' is processed. As the right child is not yet populated, nothing happens. When e_{13} occurs, it is added to E1s list. However, it does not replace the previous value like the Most Recent context. It also triggers the processing of '>' node. But '>' node has no value for the right child. e_{22} occurs, followed by e_{32}. Now when the '&' node is processed, it evaluates to true and its parent '>' node is processed. The '>' node processes its first children i.e. the instances {e_{12}, e_{22}, e_{32}}. Since there is no second value for the

right child of '>,' (i.e., '&') only one value is computed and the composite event fires. The instance, e_{13} is an initiator for the next occurrence of the composite event.

Once a composite event occurrence is detected, the primitive event occurrences that participated in its firing are all logged and are not used for the processing of the next composite event occurrence.

5.3 Runtime Activities and EHP System Components

During runtime, any of the following activities can take place:

- Addition of a composite event expression.
- Notification of an event occurrence.
- Query as to whether a composite event expression has occurred.
- Startup and Recovery.

Before we describe the functionalities of the various modules of the EHP in detail, let us describe the data structures that are shared by all the modules:

- Event Graph: It is the internal structure representing a composite event, as described in the previous section. It contains methods to process the different operators.
- Event History Object: It is a data structure that holds the trigger name and the event graph of its composite event.
- Trigger Occurred Lookup Table: It is the table containing information about the occurrence of the trigger and the timestamp at which the trigger fired.

Definition Processing Module

The Definition Processing Module is responsible for:

- Adding the trigger and its composite event expression to its list of triggers.
- Converting the composite event expression into an event graph.

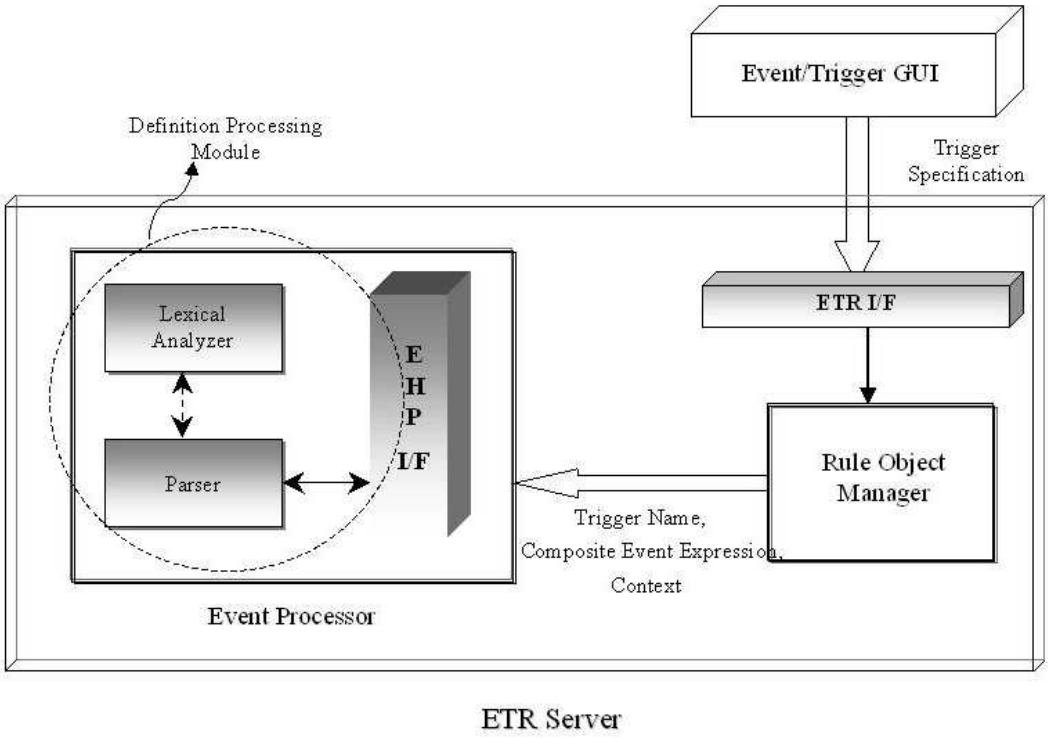

Figure 5-4 Definition Processing Module

The input to the Definition Processing Module is the trigger name, the composite event expression, and the context of evaluation of the composite event expression. The expression is sent to the Parser first. It parses the expression based on the grammar specified for the operators. The Lexical Analyzer supplies the tokens. The tokens are the different operators, the event identifier and the date. The expression is parsed and converted into an “event graph” of the specified context. The event graph of each parameter context is derived from a base event graph. During the construction of an event graph, the event graph nodes are dynamically instantiated, based on the specified parameter context. This facilitates the implementation of any new contexts without

requiring any change of the Event History Processor code. A new parameter context event graph can be implemented by deriving from the base event graph.

Once the event graph is constructed, an event history object is created. After the addition of an expression, the EHP returns a set of events for which it has to receive notifications.

Runtime Processing Module

The Runtime Processing Module is responsible for:

- Event storage.
- Event notification handling.
- Composite event expression status query.

Event Storage

The Runtime Processing module stores the primitive event occurrence and the composite event occurrence in the History Database in the corresponding event and trigger table. Object Query Language (OQL) is used to interact with the database. The required OQL statements are generated automatically by the Database Command Generator. The Database Connector interfaces with the Persistent Object Manager [34] and connects to the History Database and hence executes all the queries generated.

Event notification handling

The posted event is the input for the event notification process. The Event History Processor I/F receives the notification of an event. It finds the name of the event. The event occurrence along with the timestamp and event identifier is inserted into its event table in the History Database. The timestamp for the occurrence is the timestamp of the

notified event. The event identifier for an occurrence is generated automatically. It is formed by appending the event name with the number of rows in the corresponding event table. The number of rows is maintained in internal data structures and the database need not be queried.

If the expression contains the posted event occurrence type as one of its constituent events a thread is spawned using its Event History Object for evaluating the

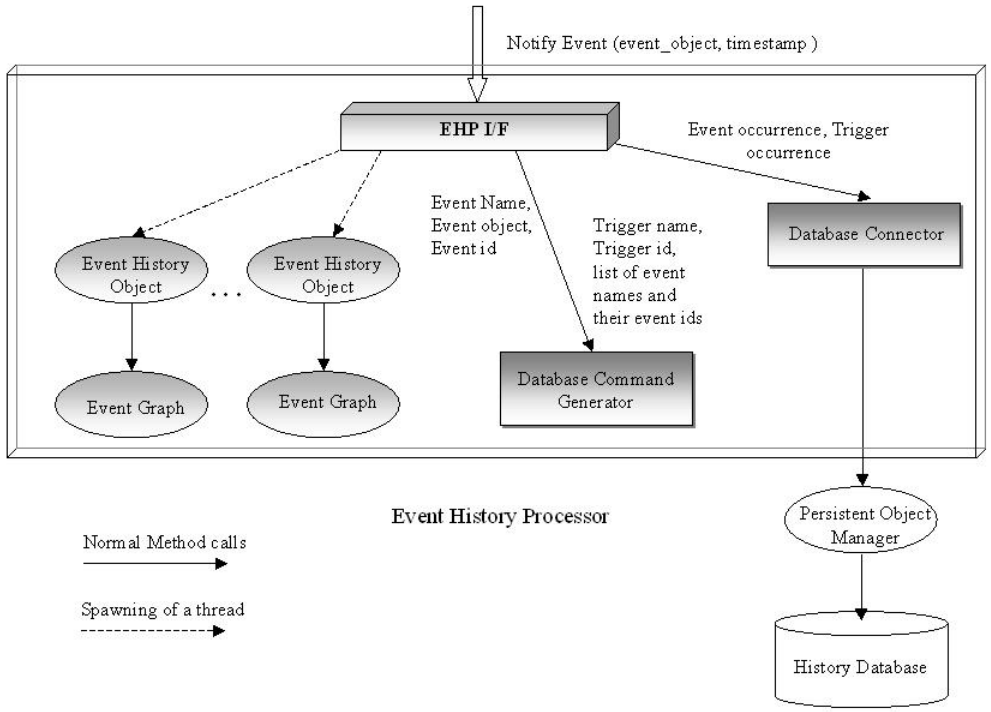

Figure 5-5 Runtime Processing Module

composite expression. The event graph of the Event History Object is evaluated in each spawned thread. The event occurrence is added to the leaf node(s) which is (are) of the posted event type. The event graph is processed bottom up starting from the

corresponding leaf node. The truth-values computed at each node are added to the list of values for the node, based on the specified context.

After all threads of control complete execution, the EHP runs through the list of expressions just evaluated and checks to see if any has fired. If a composite expression has fired, it inserts the composite event occurrence into the trigger table. The timestamp for the occurrence is the timestamp of the posted event. The trigger identifier is generated automatically. It is formed by appending the trigger name with the number of rows in the corresponding trigger table. The number of rows is maintained in internal data structures and the database need not be queried. For each event object of the trigger, an object reference to the event object is stored. The event identifier serves as the named object and is also the object reference id. The event ids are stored in the leaf nodes for each instance of the event. The corresponding event history object is also dumped into the Log. The Trigger Occurred Lookup Table is updated.

Only one event is processed at a time by the Runtime Processing Module. Any other events that arrive are queued and get in only if the processor is not processing anything currently. The event occurrences that make a composite event fire are not used for another occurrence of the composite event.

For the most recent context, an addition of a truth-value to the list of values in the nodes results in a replacement of the previous value if present. For the chronicle context, the list of values is a queue and truth-values are added to the end of list. The sets of values present in the head of a list group together for evaluation of an occurrence of the composite event. Similarly the values that are second in all lists group together. False

boolean values present in the list of head node of an event graph results in the deletion of the same and also the corresponding value in the list of the other nodes of the tree.

Composite event expression status query

When the Rule Object Manager receives the notification of a triggering event, it determines if there is a corresponding composite event expression. If an expression is present, it queries the Runtime Processing Module of the EHP about the evaluation of the composite event. When the EHP receives this query it refers the Trigger Occurred Lookup Table and responds with a boolean value as to whether the composite event has evaluated to true or false. If the composite event expression evaluated to true, the Rule Object Manager will execute the corresponding rule structure.

Startup / Recovery Module

The Startup / Recovery Module is responsible for:

- Retrieving the internal data structures and tables of the EHP.
- Restoring the system back to the condition before it had gone down.

At startup, the event graph structures of the composite expressions and tables maintained by the EHP are retrieved from the Log. In the recovered event graphs, the truth value list of the head node is checked to see if there are any composite event occurrences. If there are any composite event occurrences, the History Database is queried to check if the occurrence has been stored. If it has not been stored, the composite event occurrence is inserted into the corresponding trigger table. This handles the case of the system going down after logging but before storing in the History Database.

The event graph of a composite event expression is dumped into the Log only when the expression evaluates to true. Hence on restoring during startup, the event graph

will not be the latest one. It will not contain the primitive event occurrences that were posted between the time of last firing of the expression and the system going down. Hence, all occurrences of the constituent primitive events of the composite event expression that occurred after the last trigger fire timestamp are retrieved from the corresponding event tables in the History Database. They are arranged in the order of increasing timestamps and the event instances retrieved are re-posted and the event graphs are re-evaluated in the same way as that of during notification. This is re-done for each composite event expression to complete the recovery.

CHAPTER 6
IMPLEMENTATION

This chapter describes the implementation of the design described in the previous chapter for the *Event History Processor*. The ETR Server has been implemented as RMI (Remote Method Invocation) server in Java. The EHP, a component in the ETR Server, has also been implemented in Java. The EHP can also function as a separate server by itself in the form of a CORBA and a RMI EHP Server. In the next section, we describe the overall class diagram of the *Event History Processor*, which includes the class hierarchies and the interaction among various classes.

6.1 Class Diagram

There are two classes that are used during build time and which are used by the Import Module described in Chapter 5. They are described below:

ImportEventClass

This class contains only one public method that creates the event tables and imports them into the database.

- ImportEventClass (eventName, eventDirectory): This method is used to generate and compile event table code in the specified EventDirectory. It also imports the event class and the event table class into the History Database.

ImportTriggerClass

This class contains only one public method that creates the trigger table and imports it into the database.

- ImportTriggerClass (triggerName, triggerDirectory, compositeExpression): This method is used to generate and compile trigger table code in the specified triggerDirectory. It also imports the trigger table class into the History Database.

The overall class diagram of *Event History Processor* is shown in Figure 6-1. The implementation details of each class are given in this section.

EventHistoryProcessor

This is the main class and is the public interface class of the Event History Processor. It has the following attribute:

- eventHistoryObjectList: It is of type Vector and contains a list of elements of type eventHistoryObject, which holds all the information of a composite expression.

The eventHistoryObject contains the expression graph and has a method that triggers the processing of the event graph on an event occurrence. The details are given later in this section.

The EventHistoryProcessor has four public methods:

- addEHExprDefn (triggerName, mode, eventHistExpr): This method is used to add the composite expression corresponding to a trigger specification into the Event History Processor. The mode specifies the parameter context, either most recent or chronicle.

- notifyEvent (eventObject, timestamp): When an event is posted across the system the Event History Processor is notified and this method is invoked. It takes as parameters an event object and the timestamp of the event's occurrence. It performs all the actions described in the previous chapter concerning the notification of an event. A thread is spawned for each event history object whose event graph contains the posted event.
- boolean evalTriggerEventHistory (triggerName): This method is called by the Rule Object Manager of the ETR Server to find out if the composite expression corresponding to the trigger has fired. A boolean value is returned indicating if the trigger has fired or not.
- recoverOnStartup (): This method is used by the Event History Processor to initialize data structures and local re-posting of events not present in the log files.

To perform its functions, the EventHistoryProcessor class makes use of the following classes (and their subclasses): Parser, LexicalAnalyzer, EventHistoryObject, EventGraph, Recovery, and DBCmdGenr, and DatabaseConnector during run-time. The details of these classes are given below.

Parser

The Parser parses the event history expression based on the event history grammar and generates the event graph for the expression. It has the following method:

- parseEventHistExpr (modeString, inputExpression): This method is called by the EventHistoryProcessor to parse the input expression and convert it into the event graph structure when addEHExprDefn is invoked. This method takes the composite

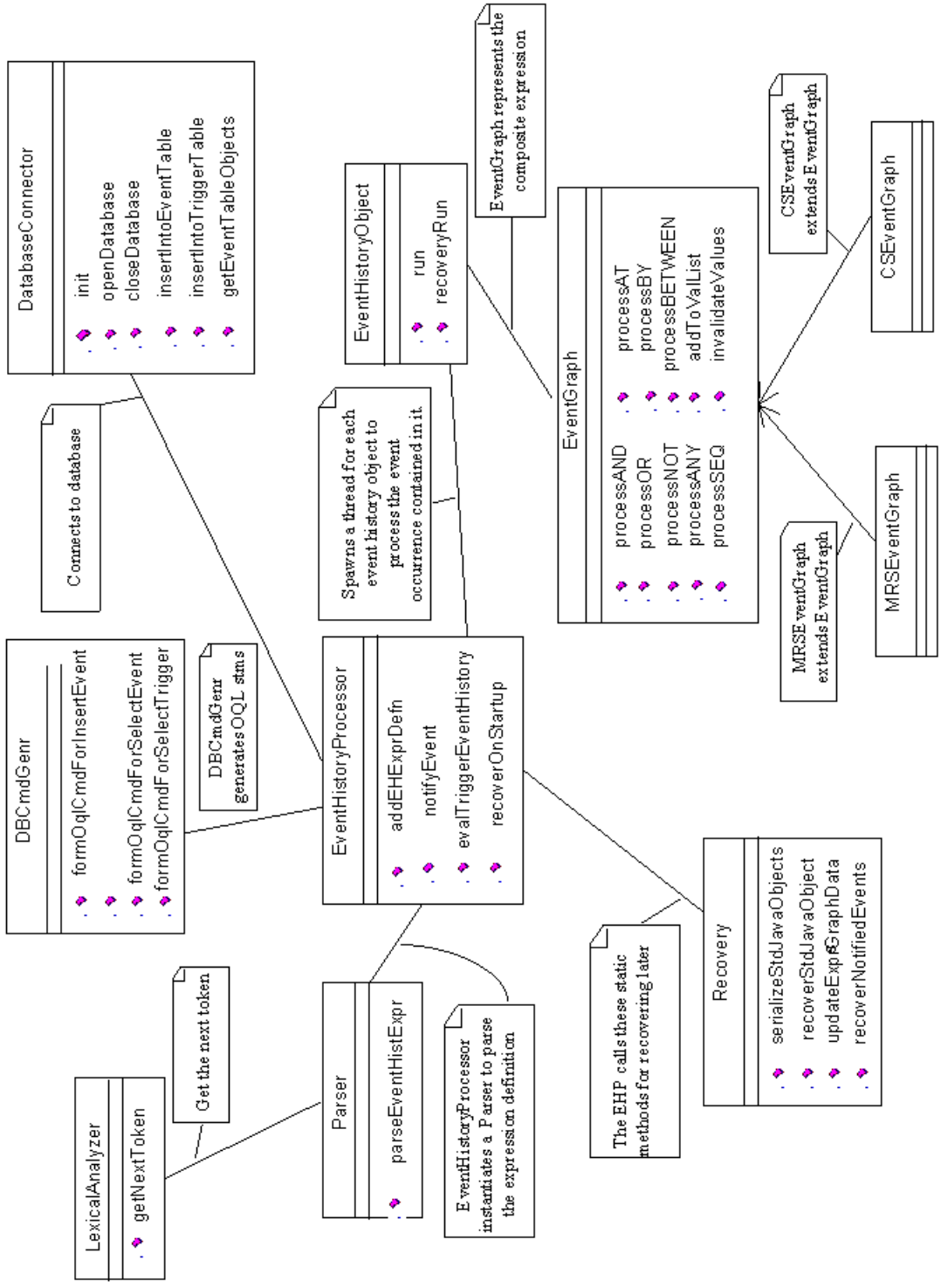

Figure 6-1 Overall Class Diagram

expression and the parameter context to construct the event graph. It parses the expression based on the specified grammar. While instantiating an event graph node to construct the graph, it uses the modeString and appends it with “EventGraph” and instantiates it dynamically. Any parameter context to be implemented has a name in the form of [PC]EventGraph where PC is the name of the parameter context.

LexicalAnalyzer

The Lexical Analyzer returns valid tokens to the parser for an input composite event expression, stored in form of a string. It has the following public method:

- Token getNextToken (): This method is used by the Parser to get the next valid token. Each token is classified as any of the event operators or event identifier or a date. If the lexical analyzer does not find a valid token it returns a dummy token to the parser.

EventGraph

This is an *abstract* class that defines the common functionalities of the various parameter context event graphs. It is the internal representation of the composite event expression. The data structure is in the form of tree, with left and right children (of type EventGraph’s) and a parent pointer for each node. Each node has a value list that holds a structure called TreeNodeObject containing the truth-value and the timestamp. For a leaf node, a true value is added if the event occurs and for an internal node the result of applying the operator to its operand is added. It has the following methods:

- addToValList (valListObject): It adds a TreeNodeObject to the value list maintained by the event graph node. This method is implemented in its derived classes, namely CSEventGraph and MRSEventGraph, according to the definition of the corresponding parameter context.

- invalidateValues (aEventGraphNode, index): To delete an item present in the specified index from the value lists maintained by the aEventGraphNode. This method is implemented in its derived classes. This method is called by the EventHistoryObject of the corresponding expression, when a trigger fires and it has been stored persistently.
- boolean processANY (parentEventGraphNode): This method is used to process the operator ANY. It takes as input the parent pointer of the node. It is called by the EventHistoryObject when it is processing an event occurrence. It is implemented accordingly in the derived classes.
- boolean processAND(): This method is used to process the operator AND. It is implemented accordingly in the base classes.

Similar to the processAND function, it has functions for all the event operators that are over-ridden in the derived classes. These methods are processOR, processNOT, processSEQ, processBY, processBETWEEN and processAT.

MRSEventGraph

This class extends the *class* EventGraph and implements the abstract methods of EventGraph based on the Most Recent semantics. It contains the following methods:

- addToValList (valListObject): It replaces the previous object, if present in the value list or adds the object as the first element. The maximum size of the list is one.
- invalidateValues (aEventGraphNode, index): It deletes an item present in the specified index from the value list maintained by the aEventGraphNode.

- boolean processAND(): This method is used to process the operator AND. It applies the boolean AND operator on its left and right child node values. There is only one value for each node, as the size of the value list in this semantics is one.

The MRSEventGraph class also implements the methods processANY, processOR, processNOT, processSEQ, processBY, processBETWEEN and processAT as mentioned in EventGraph class. These operators are implemented as discussed in Chapter 3 for the Most Recent semantics.

CSEventGraph

This class extends the *class* EventGraph and implements the abstract methods of event graphs based on the Chronicle semantics. It contains the following methods:

- addToValList (valListObject): It adds the valListObject to the end of the list, and does not replace as it is done in the MRSEventGraph. There is more than one object in the list. The values are processed in order by the parent nodes.
- invalidateValues (aEventGraphNode, index): It deletes an item present in the specified index from the value list maintained by the aEventGraphNode.
- boolean processAND(): This method is used to process the operator AND. It applies the boolean AND operator on its left and right child node values. It has to apply the operator to all the pair of values present in its right and left child nodes. There may be more than one value for each event graph node, as values are added to the list and there is no restriction to the size of the list.

The CSEventGraph also implements the methods processANY, processOR, processNOT, processSEQ, processBY, processBETWEEN and processAT as mentioned

in EventGraph class. These operators are implemented as discussed in Chapter 3 for the Chronicle semantics.

EventHistoryObject

This class contains the expression event graph. The EventHistoryProcessor spawns a thread using this object when it receives a notification. It has the following methods:

- run (): When the EventHistoryProcessor spawns a thread using this object, the 'run' method is invoked. It populates the graph with the event instance and processes the graph by calling the appropriate methods (e.g processAND, processOR etc) on EventGraph node.
- recoveryRun (): This is called by the EventHistoryProcessor at recoverOnStartup. It re-populates the event graphs, and re-evaluates them as events are recovered from the History Database during recovery at startup.

Recovery

This class handles all the recovery procedures required for the Event History Processor. The class EventHistoryProcessor uses it. It has the following methods:

- serializeStdJavaObjects (theObject, theObjectName): Serializes the object, theObject in the file of the name theObjectName.ser.
- recoverStdJavaObjects (theObject, theObjectName): Recovers the object, theObject from the file theObjectName.ser. The objects that are passed are the internal data structures and tables maintained.

- updateExprGraphData (eventHistObjList, dbCnector, triggerOccuredLookupTable): In this method, the recovered event graphs are checked to see if there are any composite expressions which have fired but not yet stored in the database. If any such composite expressions are there, then they are inserted into the database. For each object in the eventHistObjList, the fired triggers are checked if they exist in the database using the dbCnector. The changes are reflected in the hashtable called triggerOccuredLookupTable.
- recoverNotifiedEvents (leafNodePtrs, lastTriggerTimestamp, databaseConnector): It is called by the class EventHistoryProcessor. It returns a list of events notified after the last firing of a composite event expression at lastTriggerTimestamp corresponding to a trigger. The leafNodePtrs contain the constituent primitive events. It uses the databseConnector to connect to the persistent object store.

DBCmdGenr

This class generates the OQL commands automatically to execute a query on the persistent object store. It has the following methods:

- formOqlCmdForInsertEvent (eventName, eventObject, eventInstanceId, timestamp): It forms the OQL command to insert an element into the event table. It includes the event id, timestamp and the event object. The individual parameters of the event object are extracted and the statement is formed calling a private method, formOqlInsCmdForAClass. This method is recursively called until simple objects are reached for a complex object.
- formOqlCmdForInsertTrigger (triggerTableName, triggerId, leafNodePtrs, timestamp, index): It forms the OQL command to insert a trigger into the trigger table. It

includes the triggerId, timestamp and the event objects, which were responsible in causing the trigger to fire. The leafNodePtrs contain the event object reference ids of the constituent events in the value list at the specified index, for the specific occurrence of the composite event.

- formOqlCmdForSelectEvent (eventName, timestamp): It generates the OQL statement for selecting event occurrences from eventName table, that have occurred later than the specified timestamp.
- formOqlCmdForSelectTrigger (triggerName, triggerId): It forms the OQL statement for selecting the trigger with the specified triggerId from the triggertName table.

DatabaseConnector

This is the class that is used to interface with the persistent object store manager. It calls the methods on the persistent object store manager, which execute a query. The persistent object manager connects to the MS Sql Server and stores the objects in the tables created. It has the following methods:

- init (): This method initializes the database. Before any operation is carried out in the database, it must be initialized.
- openDatabase (): Must be called after init (). It opens the database where all the events and triggers are stored.
- closeDatabase (): This is used to close the database and disconnect from the server.
- insertIntoEventTable (eventInstanceId, command): It binds the eventInstanceId to the insert event command specified by command. The command is the statement for inserting the event object into the event table.

- insertIntoTriggerTable (command): It executes the command. The command specifies inserting a trigger into its corresponding trigger table.
- getEventTableObjects (selectCmd): This method is used to execute the select query which selects all the event occurrences from an event table that have occurred after a given timestamp. It returns a list of event objects.
- getNoOfRowsOfTrigger (selectCmd): Gets the number of rows of a trigger with particular trigger id. The selectCmd is executed and the resulting number of rows is returned.

6.2 Event Notification Procedure

When the ETR Server is notified of an event occurrence, the following takes place:

- The ‘notifyEvent’ method of the EventHistoryProcessor is called.
- The event occurrence needs to be stored persistently. The ‘formOqlCmdForInsertEvent’ is called to generate the database command dynamically, followed by the ‘insertIntoEventTable’ of the DatabaseConnector.
- A thread is spawned for each EventHistoryObject which contains the posted event as its constituent event in its EventGraph expression. The ‘run’ method of the EventHistoryObject is called when the thread starts execution
- For each EventHistoryObject, the following takes place in the ‘run’ method:
  - The addToValList (eventObject) method is called on the leaf node EventGraph object(s) to populate the leaf node(s) of the event expression, with the posted event occurrence.

- ♦ The composite event expression graph is traversed bottom-up from the leaf node to which the event occurrence was added. The operator at each parent is processed. Based on the operator present in the parent node, any one of the methods processAND, processANY, processOR, processNOT, processSEQ, processBY, processBETWEEN or processAT is called on that EventGraph object. The methods are called until the root EventGraph node of the composite event expression is reached or until the evaluation of a node does not yield any results.
- ♦ The thread completes action after the processing of the expression.

- After all the threads join, the composite event expressions that have evaluated to true need to be dumped into the log and also stored persistently into the History Database. The EventHistoryObject is logged in the serialized file by calling the 'serializeStdJavaObjects' method of the Recovery class. Then the 'formOqlCmdForInsertTrigger' of DBCmdGEnr is called followed by the 'insertIntoTriggerTable' method of the DatabaseConnector to store the composite event occurrence into the History Database.

6.3 Recovery Procedure

When the event processor starts up, the 'recoverOnStartup' method of the EventHistoryProcessor is called. It does the following to initialize and recover.

- The 'recoverStdJavaObjects' method of the Recovery class is called for each of the data structures and tables used by the Event History Processor to retrieve them from the Log.

- The ‘updateExprGraphData’ method of the Recovery class is called to commit triggers that have just been logged on firing and not yet committed.
- For each composite event expression present in the list of EventHistoryObject’s:
  - The event instances of the constituent events of this expression which occured after the last trigger fired timestamp must be recovered. The ‘recoverNotifiedEvents’ method of the Recovery class is called to perform this step.
  - The event instances are arranged in timestamp order.
  - The recovered events are re-posted locally to redo the evaluation of event graphs with the event instances not logged before. The ‘recoveryRun’ method of the EventHistoryObject of this expression is called to re-evaluate the event graph.

CHAPTER 7
SUMMARY, CONCLUSIONS AND FUTURE WORK

This thesis describes the design and implementation of an Event History Processor for the support of an Active Distributed Objects System (ADOS). Just as ECA rules have made a database system active, the event-driven rule technology provides the rule-based interoperability to make a distributed objects system active. The Event-Trigger-Rule (ETR) Server, is the key component in supporting an ADOS. It processes the incoming events (including composite events) and manages the firing and execution of the triggered rules. The Event History Processor is a component of the ETR Server that handles the composite event processing.

To allow a composite event specification, the trigger specification language has been extended to include a variety of event operators to form the composite event expressions. A grammar for operator precedence and associativity has been defined. Also, different parameter contexts have been used to determine the manner in which composite event expressions are evaluated. In this work, we have implemented the Most Recent Context and the Chronicle Context.

In the Event History Processor, the approach taken to evaluate composite expressions is the use of event graphs. Each expression is parsed and converted into an "event graph" of the specified context. The event graph of each parameter context is derived from a base event graph. During the construction of an event graph, the event graph nodes are dynamically instantiated, based on the specified parameter context. This

facilitates the implementation of any new contexts without requiring any change of the Event History Processor code. A new parameter context event graph can be implemented by deriving from a base event graph. During the notification of an event, all event graphs containing that event type are evaluated and processed in parallel. The concurrent processing of event graphs provides efficient implementation in a multiprocessor environment.

Primitive event occurrences and composite event occurrences are stored persistently in the History Database. Storage of event data in a History Database is useful for the subsequent analysis of event patterns and extraction of event parameter data. For example if Stock_Price_Rise_Event is a primitive event, it may be useful to know the past data for the analysis of the increase in stock price trend.

Robust recovery from system failures is provided. The event graphs are serialized in files and are retrieved on starting the server. During recovery the composite event occurrences that have been logged but have not yet been stored in the History Database are inserted into the History Database. Reposting of constituent events of an expression, that occurred after the last firing of the expression is performed to redo the evaluation of its event graph once more. This is done for each expression making the recovery complete.

Future Work

- Implement the composite event specification language completely.

For example operators such as the periodic and the aperiodic operators have not been implemented. The operator '*', which indicates zero or more occurrences must also

be included. The ANY operator can be enhanced with features like ‘n’ occurrences of the same event.

- Implement additional parameter contexts.

Additional parameter contexts such as the cumulative context and continuos context can be implemented.

- Extend the trigger specification.

Currently, only parameters from triggering events can be passed to the rules. The trigger definition can be extended to pass parameters to rules. In that case, the current EHP has to be extended to extract the relevant event parameters for a composite expression when it fires and pass it onto the rules. This can be accomplished by extending the EHP to query the database and extract the required information.

- Maintain backup log files.

Logging structures in serialized files is assumed to be an atomic action. Having backup of files containing event graph structures will enable us to do partial recovery in the event of the system crashing during the writing of serialized files.

REFERENCES

[1] Chakravarthy, S., Anwar, E., Maugis, L., and Mishra, D., "Design of Sentinel: an Object-Oriented DBMS with Event-Based Rules," Information and Software Technology, Vol. 39 (9), London, Sept. 1994, pp. 555-568.

[2] Agarwal, S., "Event Management in an Active Object Request Broker," M.S. Thesis, Department of Computer and Information Science and Engineering, University of Florida, 1998.

[3] Iakovos Motakis and Carlo Zaniolo, "Formal Semantics for Composite Temporal Events in Active Database Rules," JOSI, 1997, pp 1 - 37

[4] Krishnaprasad, V., "Event Detection for supporting active capability in an OODBMS: Semantics, Architecture and Implementation," Department of Computer and Information Science and Engineering, University of Florida, 1994.

[5] Luckham, C., D., Frasca, B., "Complex Event Processing in Distributed Systems," Technical Report, Stanford University, CA, August 18th 1998.

[6] Gehani, N.H. and Jagadish, H.V., "ODE as an Active Database: Constraints and Triggers", In Proceedings, 17th International Conference on Very Large Databases, pages 327-336, Barcelona, Spetember 1991.

[7] Agarwal, R., and Gehani, N.H., "ODE (Object Database and Environment): The Language and the Data Model", SIGMOD Conference 1989, 36 - 45

[8] Gehani, N.H., Jagadish, H.V., Shmueli, O., "Composite Event Specification in Active Databases: Model & Implementation," Proceedings of the 18th VLDB Conference, 327-338

[9] Gatziu, S., and Dittrich, K.R., "Detecting Composite Events in Active Databases using Petri Nets," in Proceedings of the 4th International Workshop on Research Issues in Data Engineering, Active Database Systems, February 1994.

[10] David, R., "Petri Nets and Grafcet: Tools for modeling discrete event systems," Prentice Hall, New York, 1992.

[11] Jensen, K., “Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use,” Vol. 1, EATCS 1992

[12] Chakravarthy, S., Krishnaprsad, V., Anwar, E., Kim, S.K., “Composite Events for Active Databases: Semantics, Contexts and Detection,” Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.

[13] Buchman, A., Chakravarthy, S., Dittrich, K., “Active Database System,” Seminar Report 9412, Sclob Dagstuhl, March 21 – 25.

[14] Schwiderski, S., “Composite Events in Distributed Systems,” University of Cambridge, 1996

[15] Object Management Group. CORBA Services: Common Object Service Specification. Technical report, Object management Group, July 1998

[16] White Paper – OrbixTalk, IONA technologies PLC., http://www.iona.com, 1997.

[17] Object Management Group, Object Management Architecture Guide, John Wiley & Sons, Inc., New York, September 1992.

[18] Object Management Group, The Common Object Request Broker: Architecture and Specification, John Wiley & Sons, Inc., New York, 1992

[19] Shyy, Y.M., Arroyo, J., Su, S.Y.W., and Lam, H., “The Design and Implementation of K: A High-Level Knowledge-Base Programming Language of OSAM*.KBMS,” Very Large Data Base (VLDB) Journal, Vol. 5, No. 3, 1996, pp. 181-195.

[20] Shyy, Y.M. and Su, S.Y.W., “K: A High-level Knowledge Base Programming Language for Advanced Database Applications," Proc. ACM SIGMOD, Denver, CO, May 1991.

[21] Su, S.Y.W., Lam, H., Arroyo, J., Yu, T.F., and Yang, Z., “An Extensible Knowledge Base Management System for Supporting Rule-based Interoperability among Heterogeneous Systems,” Proc. of the Conference on Information and Knowledge Management (CIKM '95), Baltimore, MD, November 28 - December 2, 1995, pp. 1-10.

[22] Su, S.Y.W., Lam, H., Yu, T.F., Arroyo, J., Yang, Z., and Lee, S., “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, 1996, pp. 1-30.

[23] Su, S.Y.W., Lam, H., Arroyo, J. and Nyapathi, R., "Integration of Rules and Agents in the FAIME Architecture," CIIMPLEX design document, 1996.

[24] Dayal, U., Blaustein, B., Buchmann, U., Chakravarthy, U., Hsu, M., Ledin, R., McCarthy, D., Rosentahl, A., Sarin, S., Carey, M.J., Livny, M., and Jauhari, R., "The HiPAC Project: Combining Active Databases and Timing Constraints," ACM Sigmod Record, Vol. 17, No. 1, March 88.

[25] Schade, A., "An Event Framework for CORBA-Based Monitoring and Management Systems," IBM Research Division, Zurich Research Laboratory, Switzerland, 1997

[26] Paton, N.W., Diaz, O. and Barja, M.L., "Combining active rules and metaclasses for enhanced extensibility in object-oriented systems," Data & Knowledge Engineering, Vol. 10, 1993. pp. 45-63.

[27] Su, S.Y.W. and Alashqur, A., "A Pattern-based Constraint Specification for Object- oriented Databases," Proc. of COMPCON, Spring 1991.

[28] Su, S.Y.W., Guo, M., and Lam, H., "Association Algebra: A Mathematical Foundation for Object-oriented Databases," IEEE Transactions on Knowledge and Data Engineering, Vol. 5. No. 5, Oct. 1993, pp. 775-798.

[29] Su, S. Y. W., Krishnamurthy, V. and Lam, H., "An Object-oriented Semantic Association Model (OSAM*)," AI in Industrial Engineering and Manufacturing: Theoretical Issues and Applications, Kumara, S. and Kashyap, R.L. (eds.), American Institute of Industrial Engineering, 1989.

[30] Su, S.Y.W. and Lam, H., "An Object-Oriented Knowledge Base Management System for Supporting Advanced Applications," Proc. of the 4th. Int'l Hong Kong Computer Society Database Workshop, Dec. 12-13, 1992, pp. 3-21.

[31] Su, S.Y.W., Lam, H., "Enterprise Rule Management and Services," CIIMPLEX design document, 1996 .

[32] "Learning Technology Specification Architecture," A Base Document

for IEEE 1484.1, http://www.edutool.com/ltsa

[33] Baudinet, M., Chomicki, J., Wolper, P., "Temporal Deductive Databases," In Tanset et al, chapter 13, pages 294-320

[34] Wang, L., "The Design and Implemenatation of a Persistent Object Manager for Object-Oriented Applications," M.S. Thesis, Department of Computer and Information Science and Engineering, University of Florida, 1998.

[35] Horstmann, M., and Kirtland, M., "DCOM Architecture," http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomarch.htm

BIOGRAPHICAL SKETCH

Harini Raghavan was born on July 21, 1976, in Trichy, India. She received her Bachelor of Engineering degree from Anna University, Madras, in June 1997, majoring in Computer Science and Engineering.

She joined the University of Florida in January 1998 to pursue a master's degree in Computer and Information Science and Engineering. She worked as a Research Assistant during her master's degree at the Database Systems Research and Development Center.

Her research interests include Active Databases and Distributed Computing.

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.

_____

Herman Lam, 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, Cochairman
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.

August, 1999

_____

M. J. Ohanian
Dean, College of Engineering

_____

Winfred M. Phillips
Dean, Graduate School