EXECUTION AND VISUALIZATION OF RULES IN AN ACTIVE OBJECT ORIENTED DATABASE MANAGEMENT SYSTEM

By

TAMIZUDDIN ZIAUDDIN

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

1994

Dedicated to my
Parents

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser, Dr. Sharma Chakravarthy, for his guidance and support, and for giving me an opportunity to work on the challenging *Sentinel* project. I am extremely grateful to Dr. Stanley Su and Dr. Herman Lam for serving on my committee and for their comments to improve the manuscript.

I would like to thank Sharon Grant for maintaining a well administered research environment with her indefatigable spirit and commitment to work.

I will also take this opportunity to thank all the graduate students at this center for their help and friendship.

Last, but not the least, I thank my parents and family for their love. Without their encouragement and endurance, this work would not have been possible.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iii

LIST OF FIGURES v

ABSTRACT vi

CHAPTERS 1

1 INTRODUCTION 1

2 MOTIVATION 6

3 EXISTING SYSTEM 12

3.1 Object Oriented Databases: 12
3.2 OpenOODB 13
3.2.1 Functional Features 13
3.2.2 Operational Features 14
3.2.3 Differences between Zeitgiest and Open OODB 17

4 RULE EXECUTION IN SENTINEL 19

4.1 Limitations of the Existing System 21
4.2 Our Approach 26
4.3 Implementation Details 28

5 DESIGN AND IMPLEMENTATION OF THE RULE DEBUGGER 38

5.1 Introduction 38
5.2 Design choices 39
5.3 Overall Architecture 42
5.4 Implementation 45
5.5 Functionality 55

6 CONCLUSIONS AND FUTURE WORK 58

REFERENCES 59

BIOGRAPHICAL SKETCH 61

LIST OF FIGURES

1.1 Rule Execution model continuum 3

3.1 Functional class lattice of OpenOODB 15

4.1 Sentinel Architecture 20
4.2 Exodus Architecture 23
4.3 Extensions 27
4.4 Class lattice and Extensions 28
4.5 Anchored Hash Table 33

5.1 Overall Architecture 43
5.2 Functional Modules 46
5.3 Rule execution trace 56

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

EXECUTION AND VISUALIZATION OF RULES
IN AN ACTIVE OBJECT ORIENTED DATABASE MANAGEMENT SYSTEM

By

Tamizuddin Ziauddin

August 1994

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Sciences

During the last decade, database management systems (DBMSs) have evolved considerably to meet the diverging requirements of the application domains. New generation databases require active capability for meeting application requirements. This implies that a DBMS (as opposed to user/application) would continuously monitor situations to initiate appropriate actions in response to certain events (database, external) automatically. The active feature can be incorporated into DBMSs by ECA (event-condition-action) rule abstraction. The enhancement of a passive DBMS into an active DBMS requires a mechanism for event detection and rule execution. In order to execute rules in a DBMS we need a framework/model to guarantee the semantics of rule execution. This thesis deals with i) the implementation of the rule execution model and ii) design and implementation of a Rule debugger. We use the nested transaction model to support concurrent execution of ECA rules. This implies that rules are executed as subtransactions of the triggering parent transaction. When there is more than one rule eligible for execution, there is a need for "conflict resolution." Rules can be executed in a prioritized order, concurrently, or using a

combination of both. In order to establish an order in rule execution we use priority classes. Concurrent execution can be achieved by assigning the same priority class to each rule and a total serial execution can be achieved by assigning each rule to a different priority class. Hence conceptually the execution model can be viewed as a continuum wherein at one end we have prioritized serial execution with each rule having a different priority and at the other end we have a total concurrent execution with each rule having the same priority. The nested transaction model which we have adopted ensures sibling concurrency. This implies that all rules of the same priority and executing as siblings can execute in a concurrent manner. The parent transaction is suspended until the child transactions have completed. Therefore the model which we have adopted ensures a total serial execution (each rule having a different priority) or sibling concurrency wherein all rules (with the same priority) executing as siblings do so in a concurrent fashion. In our implementation, rules are executed as light weight processes for obvious benefits.

The model of the traditional debuggers (of conventional programming languages) cannot be used in the context of active rules. The emphasis is on low level details such as program variables, subroutines, pointer referencing/dereferencing. The Rule debugger is meant to help understand the interaction among rules as well as between rules and the database. This is in contrast with the conventional debugging of programs. The interaction among rules refers to such details as nested triggering of rules, the events which cause the rule to fire or the context in which the rules are fired. The rules also affect the state of the database by way of modifying database objects and as a consequence acquiring locks. The rule debugger shows the rule-database interaction in terms of the locks acquired/released and the transactions in which the rules fire.

CHAPTER 1
INTRODUCTION

During the last decade, database management systems (DBMSs) have evolved considerably to meet the diverging requirements of the application domains. Conventional DBMSs have been passive which implies that the database changes its state only on user specified operations, a query or an update operation. New generation databases require active capability for meeting application requirements. This implies that a DBMS (as opposed to user/application program) would continuously monitor situations to initiate appropriate actions in response certain database updates, occurrence of particular states or transition of states automatically. The feature of active capability can be incorporated into DBMSs by ECA rule abstraction [Hsu88]. ECA Rules in the context of an active DBMS consists of three components: an event, a condition and an action. An event is an indication of an happening, mostly state changes that are produced by the database operations such as insert, delete and update. We can also have external/ explicit and temporal events. The condition can either be a simple or a complex query on the existing database states and the set of data objects, transitions between states of objects and even trends and historical data. Action is the set of operations to be performed when an event is detected and its associated condition evaluates to true.

Events are classified into i) primitive events-events that are predefined in the system (using primitive event expressions and event modifiers) and ii) composite events-events that are formed by applying a set of operators [snoop] to primitive and

composite events. Hence they are defined recursively. The mechanism for primitive event detection is assumed to be available. The detection of composite event may require the detection of one or more constituent events as well as one or more occurrences of a constituent event type. The occurrence of any composite event is marked by the occurrence of a constituent event that makes the constituent event occur. Recursive application of this definition will yield a primitive event that marks the end of a given composite event. This primitive event is termed the *terminator* of the composite event. Several primitive events can act as terminators but there is at least one terminator for a given complex event. Analogously, there is always a primitive event that initiates the occurrence of a composite event. This primitive event is termed as the *initiator*. The implementation of event detection has been discussed in citeKri94:thesis.

The semantics of rule of execution [Cha89b, Wid90] raises the following issues which need to be considered.

- Rule execution points: As discussed in HIPAC [Hsu88] three coupling modes for rule execution can be identified with respect to the triggering transaction. *Immediate coupling mode* rule is executed at the point where the event occurs, *deferred coupling mode* rule is executed at the end of the transaction. *detached coupling mode* Rule is executed as a separate transaction.

- Nested rules: When rule actions raise events which trigger other rules there is nested execution of rules. Rules can be nested to arbitrary levels. We need to adapt scheduling strategies (depth-first, breadth-first etc) for these rules.

- Multiple rules: In addition to the nested triggering of rules, an event can trigger several rules. When rules are triggered at the same time they form a conflict

set. There has to be some policy to order the execution of rules in the conflict set.

- Rule scheduling: As mentioned above when multiple rules are triggered at the sane time there has to be a policy for the order of rule execution. Instead of executing them in sequential fashion using some conflict resolution strategy we might have to execute them in a concurrent manner. Conceptually we can imagine a continuum wherein at one end we have priortized serial execution with each rule having a different priority. At the other end of the continuum we have concurrent execution of all eligible rules. Hence it is necessary to support a rule execution model that allows us to position ourselves anywhere along the continuum. This is illustrated in the figure 1.1 .

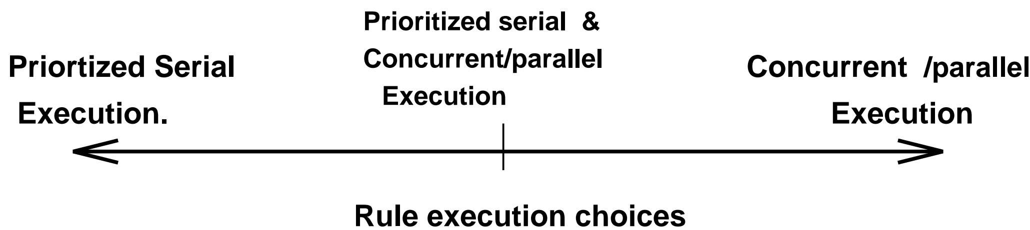

Figure 1.1. Rule Execution model continuum

- Rule management: This involves keeping track of activated and deactivated rules. Re-activating rules involves deciding whether the rule will get triggered by events that occurred prior to its activation. Based on the given priority, one can group a set of rules (e.g. integrity rules) and assign execution semantics automatically. For example, integrity rules need to be triggered in a deferred mode as the database state can be consistent within a transaction. Also, if rules are treated as shared objects (like any other shared data), then modification of rules need to be supported. This entails subjecting rules to the same concurrency control mechanism used for any other shared data. Otherwise, rules

have to be treated as meta-data whose manipulation is deemed different from shared data.

The environment/model into which ECA rules are incorporated has a bearing on some of the above. As described in [Anw93] event detection is considerably complex for an object-oriented environment and furthermore, compile and runtime issues need to addressed. The implementation of the event specification snoop, its detection and parameter computation in various context has already been discussed by Krishnaprasad [Kri94]. In this thesis we concentrate on the implementation of the rule execution model in an object-oriented database management system (Open OODB) based on the design proposed by Badani[Bad93]. The implementation involves extending the kernel functionality (transaction manager) of the existing system . Some of the issues involved are the implementation of coupling modes, nested execution, assignment of priorities of rules and scheduling of rules. The resulting system which includes event detection, rule specification and rule execution in various coupling modes is termed *Sentinel*.

Given the support for rule execution, an emerging need in the context of an active database system is to provide an environment for debugging and visualization of rule execution. The debugging of rules involves static analysis both static analysis and runtime analysis. Static analysis deals with features such as confluence. Runtime analysis deals with features such as the execution trace of rules, interaction between events and rules and rules and the database. The importance of this becomes even more clear in applications using a large number of rules.

The rest of the thesis is organized as follows. In chapter 2, we discuss the motivation for adopting the rule execution model for implementation and the need for a Rule Debugging/Visualization for active databases. In chapter 3 we describe the

existing system and its implementation, In chapter 4 we discuss our approach in implementation and the implementation details, Chapter 5 discusses the design and implementation of the Rule Debugger. Finally we present the conclusions in chapter 6.

CHAPTER 2
MOTIVATION

The motivation for this thesis is twofold. Firstly to incorporate active capability into a database management system we need to implement an execution model for rules. Secondly, there is an need in the active database field to provide for a debugging and explanation facility to understand the interaction among rules and the interaction between rules and database objects. Unlike an imperative programming environment, rules are triggered in the context of the transaction execution and hence both the order and the rules triggered vary from one transaction to another. Event-Condition-Action or ECA rules form the basis for supporting active capability [Cha89a]. HiPAC allows decoupling of the triggering event, condition evaluation and the triggered action with respect to the triggering transaction. Coupling modes are defined for rules, that determine when, relative to triggering event and transaction boundaries, the condition is evaluated and the action executed.

Support for rule execution in a database management system is driven by the following objective:

- The execution of a rule is a database operation and hence we have to maintain the correctness of this execution (ACID properties of a transaction). Also we need to provide a framework for rule execution which would preserve the semantics of rule execution. As mentioned previously multiple rules may be fired by an occurrence of an event. Hence we need to provide a conflict resolution strategy to order the execution of rules sequentially. It is also possible to execute rules in an concurrent manner or use a combination of both. As there

would be nested execution of rules and this needs to be accommodated by the execution model.

The natural step to take is to adopt the transaction framework for rule execution since we have a rigorous formalism already provided (ACID properties) to ensure correctness of the operation. This first step resolves the objective partly and in particular guarantees the correctness. However we also need to maintain the correctness in terms of the additional semantics of rule execution. In conventional DBMSs, transaction management is typically designed for using the a single level control structure. In accordance with the single level structure (flat transaction model), some strategies have been proposed for executing the triggered action to be a part of the triggering transaction. This essentially implies that some rules may be executed immediately when the triggering event occurs, and others may be delayed until the end of the transaction. These two execution strategies represent the immediate and deferred coupling mode. In either case, the triggered action is essentially executed as an in-line extension of the triggering transaction. This means that rules are executed as procedure calls within the scope of a single flat transaction. There are distinct disadvantages with this approach. Firstly we cannot achieve concurrent/parallel execution of rules since the procedure calls are executed sequentially within a transaction. Secondly if the action of any of the rules has to be undone for some reason, then we have to abort the entire transaction. In the process even successfully executed rules have to be undone since the whole transaction is the unit of recovery. Hence we need to have a finer grain of control to provide for parallelism and smaller unit of recovery.

The nested transaction model proposed by Moss [Mos81] overcomes some of the above drawbacks. The nested transaction model allows dynamic decomposition of a

transaction into a hierarchy of subtransactions preserving the properties of a transaction as a unit and assuring atomicity and isolated execution of individual subtransaction. Transactions can be an nested to arbitrary levels. In a nested transaction model, the execution of the parent transaction is suspended while its children execute concurrently. Subtransactions however lack the property of durability since the commit of the subtransaction is conditional upon its parent's commit. Essentially rules are executed as separate threads of execution in this model. In order to achieve concurrency among these threads of execution we have to maintain some concurrency control mechanism such as a lock table. Thus the concurrent execution of rules impacts on the efficiency and performance of the system. Firstly the response time is reduced since we have rules executing in parallel and the overhead of recovery is reduced since we have smaller units of recovery.

In this thesis we have implemented the nested transaction model for Open OODB [Wel92, Tex93] using the design proposed in [Bad93]. In our execution model, condition evaluation and triggered action are packaged into a different execution thread than that of the triggering transaction. Essentially the rules are executed as subtransactions of the triggering parent transaction. We can achieve sibling concurrency in our model. To increase concurrency we can model the condition evaluation of the rule as one subtransaction and action execution as another subtransaction. When there is concurrency achieved at this level we need to take additional precautions to ensure the correctness of rule execution. For example, when a substransaction executes the action of a rule, it cannot commit and update the database unless the condition of the rule is still true. The condition of the rule may be nullified (by the action of some other rule) before the subtransaction executing the action commits. The serializability criteria has to be augmented in terms of the interdependencies between the condition transaction and the action transaction. Although this approach would

increase the degree of concurrency, it may not necessarily improve performance (e.g. cost of aborts, overhead of enforcing the augmented serializability criteria). In our approach we allow the condition and action of the rule to execute in the same subtransaction. Since both are being executed in the same subtransaction the underlying locking scheme guarantees the correctness of rule execution. All sibling transactions with the same priority are executed in parallel.

The second major motivating factor for this thesis is that eventhough the active capability in terms of rules can be used beneficially in a wide range of applications, it is not a trivial task to understand, debug and maintain a large number of rules. The static analysis of database production rules has been discussed in [Wid90]. The static analysis methods are used for determining whether arbitrary sets of database production rules are guaranteed to terminate (*Termination*), produce a unique final state when multiple rules are triggered at the same time (*Confluence*),produce a unique stream of observable actions (*Observable determinism*). Termination is analyzed by constructing a directed triggered graph for a set of rules R. If there are no cycles then rules are guaranteed to terminate. Rules in R are said to be confluent if every execution graph of R has at most one final state. Also partial confluence is analyzed by analyzing confluence for a subset of rules in R. The algorithms presented in [Wid90] are conservative: they may not always detect when a rule set satisfies these properties. However, they isolate the responsible rules when a property is not satisfied. Runtime analysis is in terms of interaction among rules, between rules and events and between rules and database. Runtime analysis augments the static analysis and gives the user a better understanding of the system. In addition to static analysis it is also necessary to provide a runtime analysis of rules for the following reasons.

- We need to have an expressive rule language to accommodate a diversity of applications. However the expressiveness adds complexity. One solution is to find an appropriate declarative rule language which would allow detection of rule inconsistencies at compile time. This is static analysis. Although these languages are appropriate for specifying conditions over database states, some problems arise when trying to describe reactions declaratively. For example, reactions could be arbitrary operations other than database operations. It may be difficult to formalize these operations in order to do static analysis. Hence the expressiveness of the rule language can be limited to focus on a particular application (e.g.integrity rules which have restricted reactions only to update/restore the state of a database once a violation occurs). However we need a general purpose rule language. But in this case, as the number of rules increases it becomes difficult for the administrator to foresee possible interactions between rules. A rule debugger would be of immense help in this regard.

- . The nature of rule execution is dynamic. The rules are fired in response to a certain event. Due to the event based nature of rule execution the user does not know in advance when a rule would be fired. Hence there ought to be some sort of explanation mechanism which allows the users to ascertain what rules are activated and when. There needs to be some visualization of the runtime trace of rule execution.

- Rules are eligible for firing if appropriate events are raised. We could have multiple rules being triggered by an event. These rules form a conflict set. Hence these rules can be executed in any order. The user can focus on this set and change priorities if required.

- As mentioned previously there can be nested execution of rules. Sometimes the nested execution may lead to potential cycles i.e. a set of rules repeatedly triggering each other (which may have been determined by static analysis). It may be helpful to show or highlight these potential cycles.

- It is also very helpful to show how the execution of rules effects the state of the database. This could be shown in terms of the objects modified by the rules. This makes sense in an object-oriented context since we have object identifiers to keep track of the objects.

Hence both static and runtime analysis are important in an active database management system. In this thesis we concentrate on the runtime analysis of rules.
It can also be seen that we cannot adopt the model of the traditional debuggers (of conventional languages) readily in the context of active rules. The conventional debuggers show the sequence in which the tracked units (instructions in a programming language) executed. The instructions are executed in a fixed sequence given by the programmer. If there is an exception or an error then the program fails and it is shown by the debugger. Hence here the situation is context independent since the user is already aware of the context. Now by contrast the rules in an active database system are executed without any user intervention. As mentioned previously there is no means by which the user can know in advance which rules will be fired. Hence the debugger in context of active rules has to be context dependent [Dia93]. The context for rules is determined by the events which caused them to fire.

CHAPTER 3
EXISTING SYSTEM

3.1 Object Oriented Databases:

It has been observed that the capabilities of the record oriented data models are limited in capturing complex structural relationships and behavourial properties in advanced application domains. The Object Oriented models provide a variety of modeling constructs, which simplify the task of modeling complex data. An Object Oriented Database Management System (OODBMS) has the following salient features which distinguish it from the conventional systems.

1. They support unique identification of objects by system assigned object identifiers.

2. Support abstract data types and allow complex objects to be defined in terms of hierarchies.

3. Support inheritance of structural and behavourial properties among object classes in these hierarchies.

Using the feature of abstract classes and the inheritance mechanism, we can easily extend an existing system. For example, if we have an abstraction called "figure," we extend this abstraction and implement different types of figures such as a square, circle, etc. We use the extensibility features inherent to the Object Oriented paradigm to implement the extensions proposed in this thesis. A prototype OODBMS called **Open OODB** was used as the basis for design and implementation of our work.

3.2 OpenOODB

The Open OODB project [Wel92, Tex93] initiated by Texas Instruments, was an effort to build a high performance, multi-user object oriented database management system (OODBMS) in which the the database functionality can be tailored for the diverse needs of applications.

The system provides an incrementally improvable framework that can also serve as a common testbed for research by database, framework, environment and system developers who want to experiment with different system architectures or components. The Open OODB system architecture is divided into i) a *meta-architecture* consisting of a collection of kernel modules and definitions providing the infrastructure for creating environments and boundaries, specifying and implementing event extensions and regularizing interfaces among modules, and ii) an extensible collection of *policy manager* modules which provide functionality to the system.

3.2.1 Functional Features

- **Persistence** is the ability of objects to exist beyond the lifetime of the program which created them. Open OODB seamlessly adds functionality such as persistence to the developers existing environment. Open OODB extends the existing programming languages (C++ and Common Lisp) to provide for persistence rather trying to invent a new "database language." Open OODB does not require changes to either type (class) definitions or the way in which the objects are manipulated. Rather, applications "declare" normal programming language objects (C++ and Common Lisp) to possess certain additional properties such as persistence and it is the system's responsibility to move the object from the computational memory to the persistent memory at the program termination. The Persistent Policy Manager in Open OODB provides applications with an

interface through which they can create access and manipulate persistent objects. EXODUS is used as the Persistent store for objects. The interaction with EXODUS in transferring and saving objects is built into the Persistence Policy Manager and hence it is transparent to the user.

- **Sentry mechanism.** The Open OODB computational model allows developers to define behavioral extensions of events, which is an application of an operation to a particular set of objects. In this model all objects accessible to a program exist in a "universe of objects." This universe is partitioned into "environment" by "environment attributes." Environmental attributes include information about the address space where the object resides (e.g. persistent or transient, local or remote), replicas of the object, lock status, and transactions owning lock etc. These environments identify where extensions may be required. For example, if we need an extension to allow objects to reside in an address space, we can define an environmental attribute named "address space" that defines the location of the object could reside. To perform these extensions we must be able to interrupt or trap operations. Thus, the trapping mechanism combined with the protocol for permitting the entity performing the trapping to invoke an arbitrary extension is known as a "sentry". The primary function of sentries is to detect events which controls and performs the actual extension if it is determined that an event should be extended. The sentry manager is used for specifying events to be extended and is responsible for deploying sentries to detect extended events.

3.2.2 Operational Features

The class lattice which manifests the functionality of the system is shown in Figure 3.1.

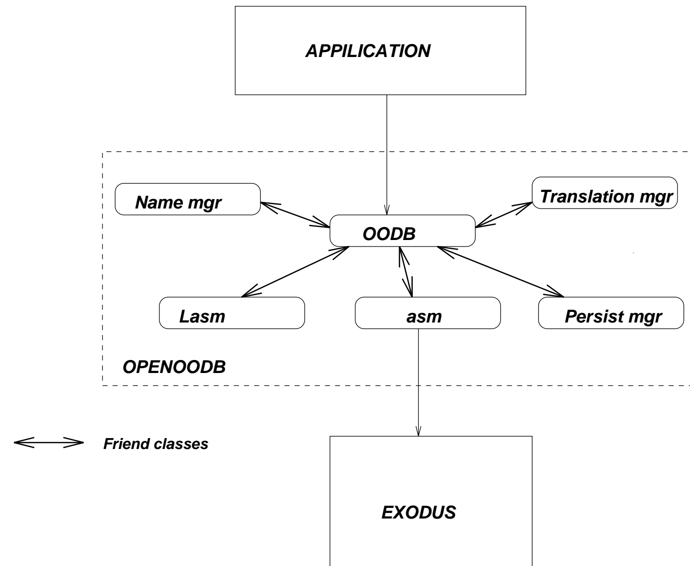

Figure 3.1. Functional class lattice of OpenOODB

- **OODB main class.** This is the main interface class between the application and the system. In every application before doing anything, an instance of this class need to be created. The constructor of this class, among other things, does the following: establishes connection with the exodus storage manager, loads all the system tables into the main memory.

- **Preprocessor.** The OpenOODB extends every application class with certain additional member functions to take care of object translation, sentry mechanism and persistence. By default every application class is derived from the wrapper class. This is again achieved by the preprocessor. If the user does not wish to extend a certain class, could do so by using the -n option of the preprocessor. By using this option the application class is left as such and it is not derived from the wrapper class.

- **Cache manager.** OpenOODB does not maintain a true Cache of its own. The tables implemented by the class *lasm* which is the local address space manager and the C heap serve as the cache. When objects are fetched from the persistent

store they are placed in the C heap. Each persistent object is given a global identification. The tables maintained by the local address space manager map the local address to the GID (Global id) when the persistent object is fetched.

- **Name manager.** The name manager has a flat namespace. This implies that we cannot have two objects with the same name in two different databases. To distinguish them however we can qualify them by the name of the database they are stored in.

- **Persist manager.** This is the policy manager which takes care of persistence. An object is made persistent by invoking the *persist* function. The persist manager first assigns an Global identifier (GID) and uses the services of the name manager to associate the name of the object with its GID. Secondly the persist manager determines the transitive closure of the object. Every object reachable from the original object is in its transitive closure. Only the root object (original object) is assigned the GID.

- **Translation manager.** The class implements the Object translation from an external to an internal computational format. In particular, the translation converts the C++ objects (along with its transitive closure) into a particular format and vice versa. Also addresses (of objects) are swizzled by the Object Translation module when a persistent object is fetched. The fetched objects are allocated on the heap. If the persistent object contains a pointer to another persistent object, a surrogate for that object is created and pointer is swizzled to the address of the surrogate. Later, when a member function is called which accesses the state of the referenced object, that object is fetched.

- **Address space manager.** This ASM implements the interface to exodus. The address space manager is implemented by the class *ASM_Client*. This

class is responsible for establishing connection with exodus, and also fetching and storing objects in exodus. This interface infact utilizes the client interface functions (of exodus). It consists of functions to create, modify and delete objects. When an instance of *ASM_Client* is created and is initialized with a name and certain parameters, the instance of ASM_Client is set up as an surrogate for the ASM.

3.2.3 Differences between Zeitgiest and Open OODB

- **Sentries.** As an enhancement to Zeitgiest, OpenOODB provides for sentry mechanism. With the help of this mechanism as mentioned above the sentry could trap events and pass control to a policy manager if it determines that an event should be updated.

- **Object translation** (translation of C++ objects). In OpenOODB an object translation mechanism is provided which automatically traverses object transitive closure and determines the most-derived object, allowing the pointer address initially to be pointing to a sub-object of the actual-object. Each class definition is extended with copy member functions that copies the object, and its sub-objects (base clases) and copies the non-pointer and pointer members of the object. The translation converts the C++ object (along with its transitive closure) into a particular format and vice versa. This feature would be of immense help when dealing with different address spaces.

- **Storage manager** (ingres vs Exodus). OpenOODB uses Exodus as the underlying storage manager while Zeitgiest used Ingres as the underlying storage

- **Differences in preprocessor.** When an application executes, some of the objects it creates are only needed temporarily for the duration of the program, then the class definitions of these objects need not be processed by the DDL

translator. Such classes are termed as transient classes. On the other hand, some objects need to be persisted to the database after the duration of the program. Such objects are termed as persistent objects. The class definitions of these objects have to be processed by the DDL translator. In Zeitgiest, for an object to be persistent, two actions must occur

1. Its associated class definition must be annotated (for persistence) and processed by the DDL translator (this allows instances of the class to become persistent)
2. The object must be created in an application program and must be designated for persistence, must be embedded within an object which has been designated for persistent (notion of Independent and Dependent Persistent objects), must be reachable from an object that has been designated for persistence (it is the transitive closure of a persistent object via referenced instances.

In OpenOODB a slightly different approach is adopted. The application classes need not have to be annotated in the application program. The decision to persist an object to the persistent store is made at run time. This is achieved by invoking the persist function which marks the object for persistence. The user class definition in an application program is extended by the preprocessor to take care of persistence , sentry mechanism and object translation. When an object is persisted, all the objects in its transitive closure are also persisted. This is taken care of by the translation functions introduced by the preprocessor. In a way we can say that there is a late binding of persistence in OpenOODB.

CHAPTER 4
RULE EXECUTION IN SENTINEL

We briefly discuss the overall architecture of sentinel and its components and highlight the extensions to Open OODB. The sentinel architecture shown in figure 4.1 extends the passive Open OODB system [Wel92, Tex93]. The Open OODB toolkit uses Exodus as the storage manager and supports persistence of C++ objects. Concurrency control and recovery are provided by the exodus storage manager. A full C++ pre-processor is used for extending the user class definitions and as well as application code. To make Open OODB active the following extensions were made.

- The ECA rule was incorporated either as part of the class definition or as part of the application code. This allows the specification of class level and instance level definition of rules and events. The C++ pre-processor of Open OODB was enhanced to preprocess the event and rule definitions into appropriate code for event detection and rule execution.

- Detection of primitive events [Kri94] was incorporated by adding Notify into the wrapper method of the Open OODB. The wrapper method permits us to invoke a notification when an event occurs and conveys it to the composite event detector.

- To detect composite events a composite event detector [Kri94] was introduced. Each Open OODB application has a composite event detector. The event detector is implemented as a class and we have a single instance of this class per application. This is shown in the figure 4.1 as the local event detector. Also there is a clean separation between event detection and application execution.

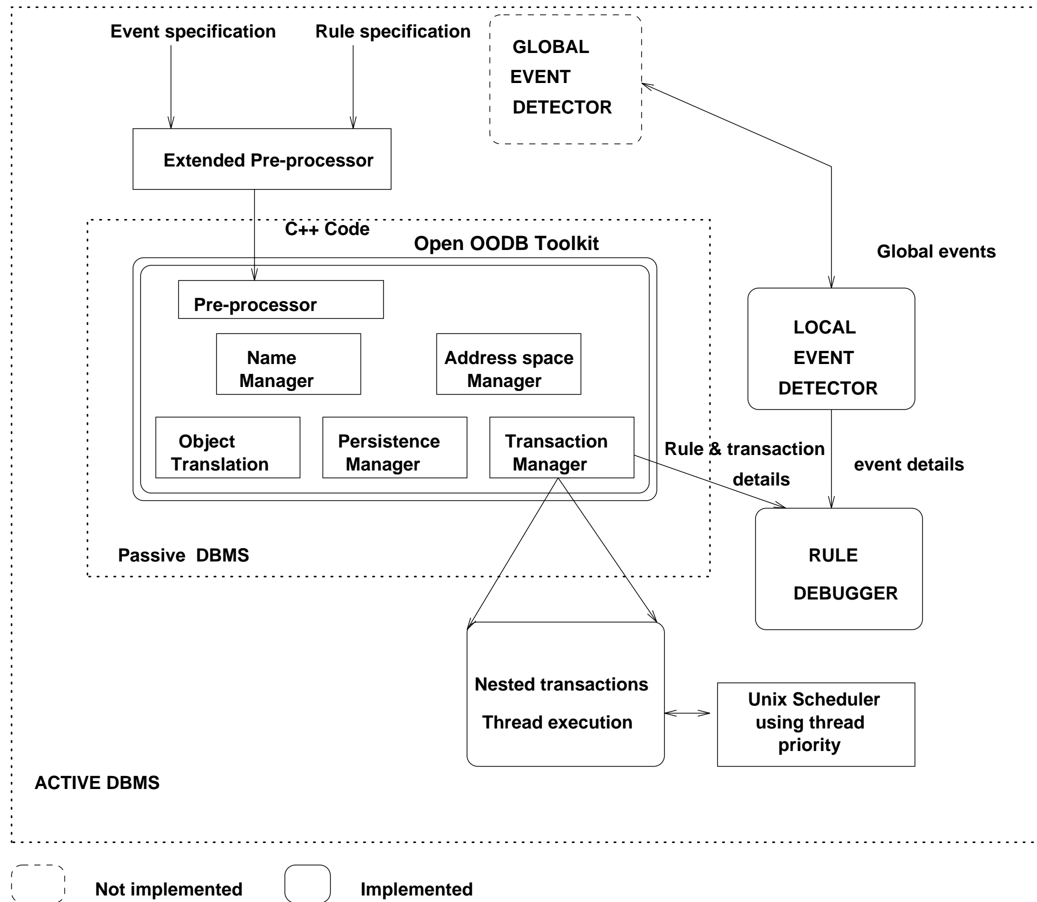

Figure 4.1. Sentinel Architecture

- As shown in the sentinel architecture we have also designed and implemented a Rule visualization/debugger module to provide feedback about rule execution. We discuss the detailed design and implementation of the rule debugger in the next chapter.

- To support rule execution, we have extended the skeletal transaction manager of the Open OODB to a fullfleged transaction manager supporting the nested transaction model. The design was according to that proposed in [Bad93]. In this chapter we discuss the implementation of the transaction manager.

Before we discuss the implementation we highlight the limitation of the existing system.

4.1 Limitations of the Existing System

To handle the execution of each rule as a separate thread of execution, the sentinel architecture uses light weight processes instead of processes. The light weight processes (threads) are used for obvious benefits: i) threads communicate via shared memory rather than file system, thus allowing applications to share the same address space, ii) the overhead involved in creating threads and the inter-task communication is low, and iii) it is easy to control the scheduling and communication of threads by assigning priorities. The condition and action part of the rule are packaged as a unit to be executed when the thread is scheduled. When the events are detected, in addition to notification, rule threads are created. The local event detector schedules the threads. Although rules can be executed as threads, to allow for commit dependencies between rules and the transaction (toplevel) which triggered the rule, we have modified the system to execute rules as subtransactions. To support nested execution of rules as subtransactions we have adopted the Nested transaction model

[Mos81]. In nested transactions, flat transactions are enhanced by a hierarchical control structure. Each nested transaction consists of either primitive actions or some nested transactions (called subtransactions of the containing transaction). This allows a dynamic decomposition of a transaction into a hierarchy of transactions, hence preserving all properties of a transaction as a unit. Subtransactions follow all the ACID properties of a flat transaction except durability, due to the fact that the commit of a subtransaction is conditional upon its parent's commit. As shown in [Hsu88], the nested transaction concept fits well with the semantics of rule execution in immediate mode and some extensions are necessary to model other coupling modes. Concurrent execution of several rules can be provided using a concurrency control strategy for an implementation of the nested transaction model. We have implemented a lock-based algorithm for supporting nested transaction model in the Open OODB. Badani[Bad93] discusses nested transaction implementation for the Zeitgeist system (a precursor to Open OODB) in detail. For understanding the rational for the approach taken in Sentinel for implementing nested transactions, it is useful to understand the interaction of the exodus storage manager with Open OODB. The exodus storage manager has client server architecture and Open OODB acts as the client of exodus. Essentially the storage manager consists of two modules: the *client* and the *server* modules. The client module serves as an interface to the application program (Open OODB in our case). The Open OODB calls routines in the *client module* of the storage manager to access and manipulate data in the persistent storage. Also, the client module cooperates with the server process to access and manipulate objects and the files that contain them. The Exodus server is a separate and possibly a remote process which provides a variety of services to the clients.
It is important to distinguish between the application (Open OODB) and the client module of exodus. The Client module is a library which is linked with Open OODB.

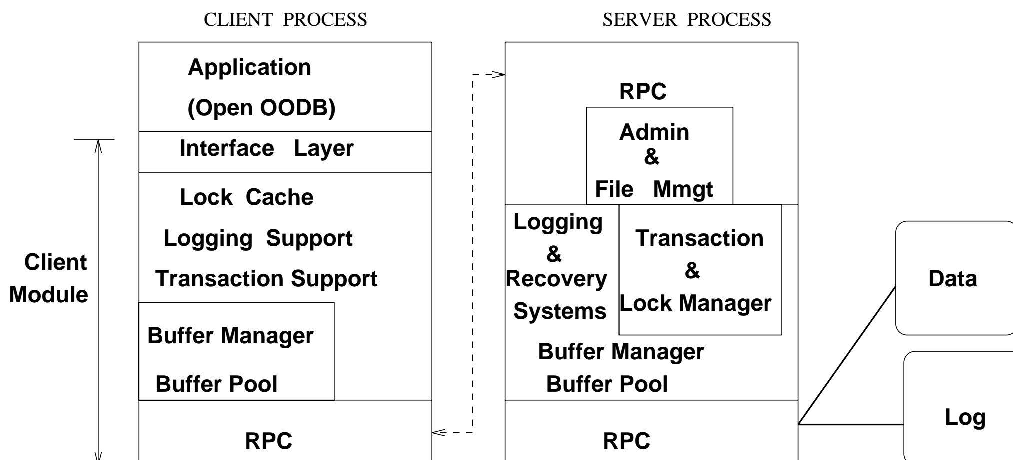

Figure 4.2. Exodus Architecture

The interface for this library provides routines for creating, destroying, reading, writing, inserting into, deleting from, and versioning objects. It is important to note that the Open OODB calls do not access the server directly, rather they in turn invoke the client module routines which accesses the server directly. Architecture of Exodus: The client-server architecture of Exodus is shown in the figure 4.2. As can be seen the client has its own buffer pool, which it uses to cache pages containing objects and indexes. Object and index operations are performed by the client module in its buffer pool. This is a departure from the standard client-server architecture where the client simply sends a request to the server to change the object instead of performing the operation itself. For transaction management the client maintains its own lock cache. The client communicates with the server's transaction and lock managers to maintain the lock cache and transaction information. The Exodus server is a multi-threaded process providing I/O, file, transaction concurrency control and recovery to the clients. All request processing in the server is performed within the context of a thread. Server threads are units execution similar to the co-routines provided by Modula-2. Each thread has its own stack for maintaining the execution states. Thread switching is implemented using the setjmp() and longjmp() functions in the

standard C library. When a request arrives, the server assigns a thread from the inactive pool to handle the request. The thread runs until it waits for a resource, voluntarily gives up the CPU, or completes the request. Once a thread has completed a request, it responds to the client and returns to the inactive pool. For logging and checkpointing, threads are also executed on the behalf of the server.

To guide the discussion of interaction of Open OODB with Exodus, we present a general application model. An application begins by performing some initialization and then starts the transaction. The initializations are done within the constructor of the OpenOODB class. The application begins by calling Initialize(). This initializes the data structures on the exodus client and establishes connection with the server. Then the data structures pertinent to Open OODB (local cache, name manager etc) are initialized. Two volumes are mounted, one for system tables (name table etc) and for the application. Finally the system tables are read into the cache from exodus.
Once the initialization is complete the application begins another transaction. The call to begin_transaction() of Open OODB in turn invokes the begin_transaction of the client interface. The client can be in one of the three states: not running within a transaction, running within a transaction, running an aborted transaction. After initialization the client is in the first state. The client then requests the transaction ID (TID) from the server. All future data and lock requests sent to the server in the scope of this transaction will contain this TID.
All Open OODB calls with regard to transactions and accessing of objects translate to one or more calls of the client interface. Now when the application requests an object that is not found in the Open OODB's cache, a client interface function is invoked to request it from the clients buffer pool. As mentioned previously the Open OODB does not maintain a true cache. The system tables implemented in the local address space manager together with the C heap serve as the cache. Open OODB

does not manage its own memory. The Open OODB cache and the client buffer pool are two different levels of buffering. If the object is not present even in the clients buffer pool, the client requests the page(s) containing the object form the server. The current interface to exodus gets the request in the default shared mode.
After performing all the object operations, an application can either commit or abort the transaction. When Open OODB commits the transaction, the following sequence of events is observed. Firstly, all objects retrieved from the persistent store and put in the Open OODB cache are written back to the exodus client buffer pool. The newly created objects (persistent) would have already been written to the client buffer when the persist function was invoked. Secondly, the commit() of the client interface is invoked. The client first ships all remaining dirty pages and log records to server and then requests the not running within a transaction. To abort a transaction the client simply discards all pages in its buffer pool and request an abort from the server. When the application is done running the transactions, a different transaction is started on the system volume to write back the system tables to the server. Finally all volumes which were mounted are dismounted and the client module is shutdown.

The limitations of the OpenOODB architecture for implementing nested execution of rules are as fallows:

1. The underlying storage manager (Exodus) does not support nested transaction calls. Although Exodus supports multiple clients (each as a thread), each client supports only a flat transaction model,

2. All the lock information about the objects is maintained by the Exodus server and the client. No lock information is present in the address space of Open

OODB . Furthermore, lock modes (except setting it globally) cannot be currently specified to Exodus through the interface in the version used (Exodus2.2.ti).

3. Currently, all Objects that are fetched by the Open OODB from Exodus are always written back to the server when the transaction commits. The reason for this is that presently the Open OODB has no way of knowing if the object fetched from the persistent store is modified or not. Hence the system has to necessarily write back the objects fetched from the persistent store. The lock is escalated into an exclusive lock at commit time. Hence in case of data conflict among applications, only one transaction will commit and the rest are aborted.

4. The transaction calls (beginTransaction, commit, abort) of Open OODB in the current architecture are tightly coupled with the same calls in the Exodus client interface. That is, there is a one to one correspondence between the transaction calls of Open OODB and Exodus. This implies that Open OODB has only a skeletal transaction manager which maps calls to Exodus utilizing the underlying transaction model provided by Exodus.

4.2 Our Approach

With the above limitations, we could either i) extend Exodus to incorporate a nested transaction model or ii) extend the skeletal transaction manager of Open OODB to incorporate nested transactions without including recovery. Although the first option is more desirable in the long term (as it would involve recovery), we opted for the second choice as it involves developing the transaction layer in the Open OODB in a shorter time to support concurrent execution of rules. The implementation entailed extending the skeletal transaction manager of Open OODB to a fullfledged transaction manager. The subtransaction calls are managed by the

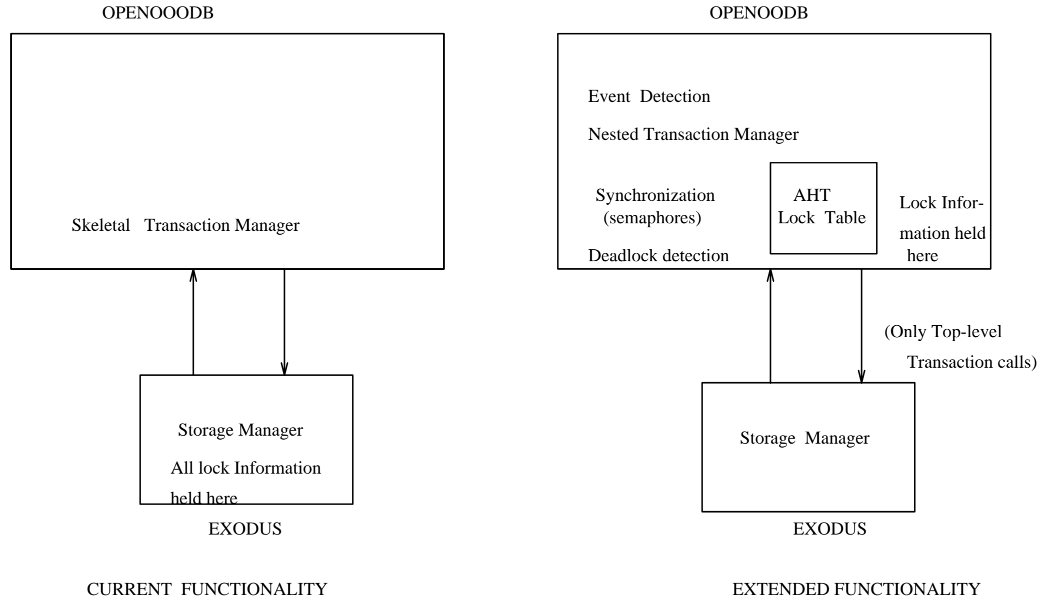

Figure 4.3. Extensions

transaction manager of Open OODB. An initial connection to Exodus is established when the the top-level transaction is invoked. If there is an object fault in the nested call (subtransaction), the object is fetched through this connection. Although the lock information of this (toplevel) transaction is maintained by the Exodus server, it is also maintained in the Open OODB as well to enforce concurrency control among substransactions. A separate lock manager is implemented in the Open OODB to handle concurrency control among nested transactions. The lock manager contains the lock information for all transactions including the top level transaction. The modification to the Open OODB is illustrated in the figure 4.3. The approach we have adopted in implementing the lock manager is a two level implementation. At one level there is the lock table maintained by the Exodus server. This maintains the lock information of all top level transactions. At the second level we maintain the lock table for the nested transactions in Open OODB.

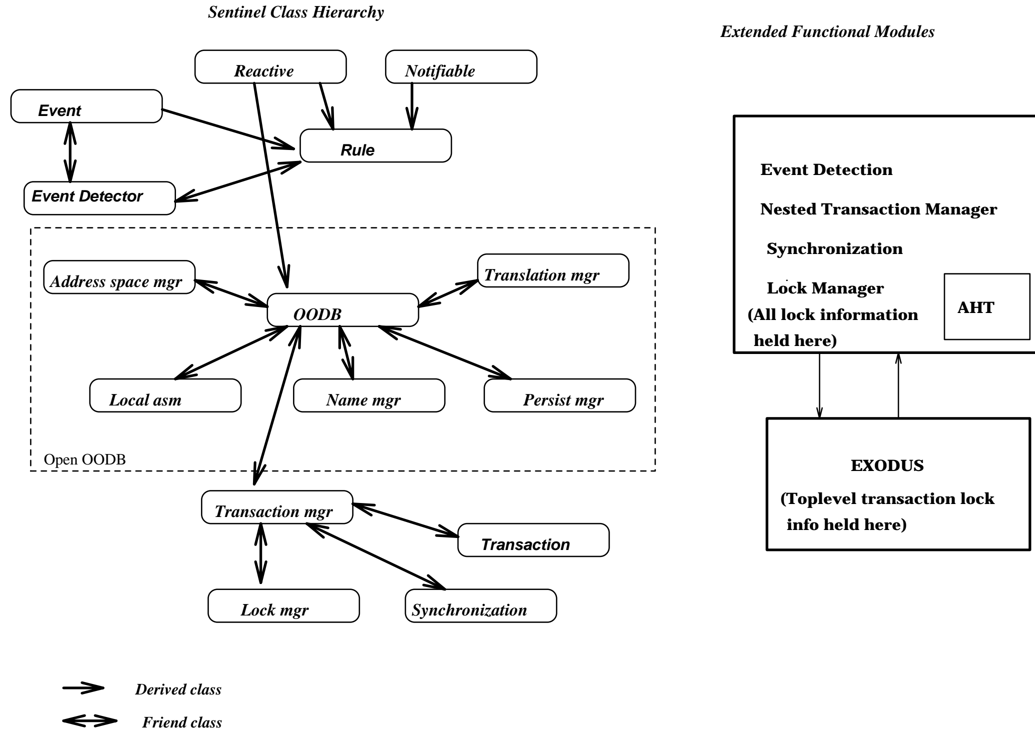

Figure 4.4. Class lattice and Extensions

Hence our approach does not alter the semantics of concurrent top-level transactions and at the same time supports nested subtransactions within each application/client. The condition-action portions of a rule are packaged into a procedure (by the pre-processor) that can be scheduled as a thread by the rule manager. This is discussed in a later section. Below, we briefly highlight the implementation of the lock manager and synchronization of subtransactions. The extensions made in the OpenOODB are illustrated in the figure 4.4.

4.3 Implementation Details

Objects retrieved form the persistent store (exodus server) are alway written to the server irrespective of the fact whether they were modified or not. This leads to exclusive lock escalation and in case of data conflicts only one transaction can commit. In order the overcome this limitation we temporarily introduced two functions namely **fetch_for_read()** and **fetch_for_write()**. When we fetch an object from the

exodus server using **fetch_for_read()** we do not write back the object to the persistent store during the commit of the transaction. This allows a read only transaction to concurrently execute and commit with other transactions.

Guided by the principle of extensibility and modularity we designed the the transaction subsystem consisting of the following components.

- **The Transaction manager container Class (Trans_Mgr)**: This Class maps the user function calls to the appropriate transaction manager calls of the kernel. Two function calls were introduced to distinguish between the top level transactions and the subtransactions namely *begin_transaction* and *begin_subtransaction*. The later takes the transaction pointer of the parent as the input parameter and both return transaction pointers as a result of the invocation of the call. This container class provides an interface by which we can seamlessly integrate different policies of the transaction system. Only one instance of this class is created per application. The constructor of this class does the following: (i) create and initialize the semaphores which would be used for synchronization. (iii) create an instance of the global *Counter* class. This keeps track of the transaction id of the previous transaction based on which we calculate the current transaction id. It also takes care of allocation and freeing of semaphores. (iii) the lock table is initialized. The following is the interface provided by this class

  - **begin_transaction():** This calls begins the transaction in the exodus storage manager. Every operation from now on is within the scope of this transaction as far as exodus is concerned. In addition this function creates an instance of transaction class. While creating the transaction instance

its transaction id is calculated appropriately. It is calculated with respect to the previous toplevel transactions already created.

- **begin_subtransaction():** This function call is introduced to distinguish between the toplevel transactions and the subtransactions. It takes the pointer of the parent transaction object as the input parameter. An invocation of this call creates an instance of the transaction class. The transaction id of the subtransaction is calculated with respect to the parent transaction.
- **Set_lock():** Invocation of this call allows for setting the locks in appropriate modes on particular object for a particular transaction. Presently only two modes are supported namely read only and exclusive locks.
- **set_transaction()** This call sets the current transaction to the transaction supplied as the argument (transaction ID) to the function. This function is needed to switch the context of the transaction when subtransaction are executing in a concurrent manner.

• **Transaction Class.** This is the component of the kernel which implements the nested transaction model. Every transaction in an application is an instance of this class. Among other methods the critical member functions which implement the nested transaction model are Set_lock, Upgrade_lock and Release_lock.

- **Set_lock and Upgrade_lock.** The Set_lock and Upgrade_lock methods were implemented according to the algorithms given by Badani [Bad 93]. In our implementation semaphores are used to used to synchronize and ensure correctness of setting and upgrading locks. In the nested transaction model, when a lock is released on a object, not all transactions requiring

a shared lock can acquire it. This is due to the spheres of control phenomenon i.e. to acquire the lock the requesting transaction has to be in the new sphere of control formed by the release of the lock. Hence the locks are granted based on the position of requesting transaction with regard to the releasing transaction. Hence in addition to having a unique semaphore for transactions waiting for exclusive lock, the transactions waiting on shared locks also should be waiting on a unique semaphore.

  - **Release_lock.** The Release_lock algorithm is implemented according to that given by Badani [Bad 93]. As mentioned above, due to the sphere of control phenomenon, before releasing a transaction form the blocked state we have to check for the eligibility of that transaction to acquire the lock

- **Lock Manager.** We have implemented lock tables in the local address space (which is shared by all subtransactions) for each instance of the Open OODB. Hence, the overhead associated with creating and maintaining shared memory is avoided.

  We assume the basic locking rules proposed in [Hae83]. Enforcement of the basic locking rules requires maintaining retain mode and lockmodes as a minimum. However, to efficiently implement *distributed nested spheres of control* and *distributed disjoint spheres of control* [Hae83, Bad93], it is necessary to search the entire tree to make sure locks can be granted in an appropriate mode. To avoid an exhaustive search of the transaction tree, we have additional information in each node of the transaction tree. This reduces the search to the nodes along the path to the root (in the worst case when the closest common ancestor is the root node). Below, we describe the holdmodes introduced for that purpose.

  Lockmodes & Holdmodes: Lockmodes specify the mode (Shared, Exclusive or Read only) in which a lock is is held on an object. To avoid excessive search,

we introduce the following **holdmodes** which are in addition to the lockmodes: Hold(x)/holdsub(x)/retain(x)/retainsub(x)/grantmode. These fields define the status of a lock and its availability to the transactions in the sphere of control and to the transactions outside. In our algorithm we have used these fields along with the lockmodes to determine the availability of lock in a particular mode. Below we briefly describe these fields.

- **Hold(x):** Indicates if the transaction in question is holding a lock (shared, Exclusive or Read only) on a particular object 'x'.
- **Holdsub(x):** It is a numerical value indicating the number of subtransactions holding the lock on object 'x'. If the lock held is shared or read only, then it will be the number of (sub)transactions holding the lock on the object. If the lock being held is exclusive, then this value will be one.
- **Retain(x):** When a subtransaction commits locks are retained by its parent thereby forming sphere of control. This field indicates the sphere of control.
- **Retainsub(x):** It is a numerical value indicating the number of subordinate transactions retaining the lock on the object. By checking this field of the parent/ancestor, it is possible to obtain the information about the existence of one or more spheres of control.
- **Grantmode:** Gives the lockmode in which the request can be granted.

The use of these holdmodes avoids exhaustive search of the hierarchy of the nested execution. In fact the algorithm guarantees that the search can be limited to the path from the node requesting the lock to the root of the transaction tree containing that node. We have also introduced the notion of **virtual node** that serves as the root of all top-level transactions in the system. This further

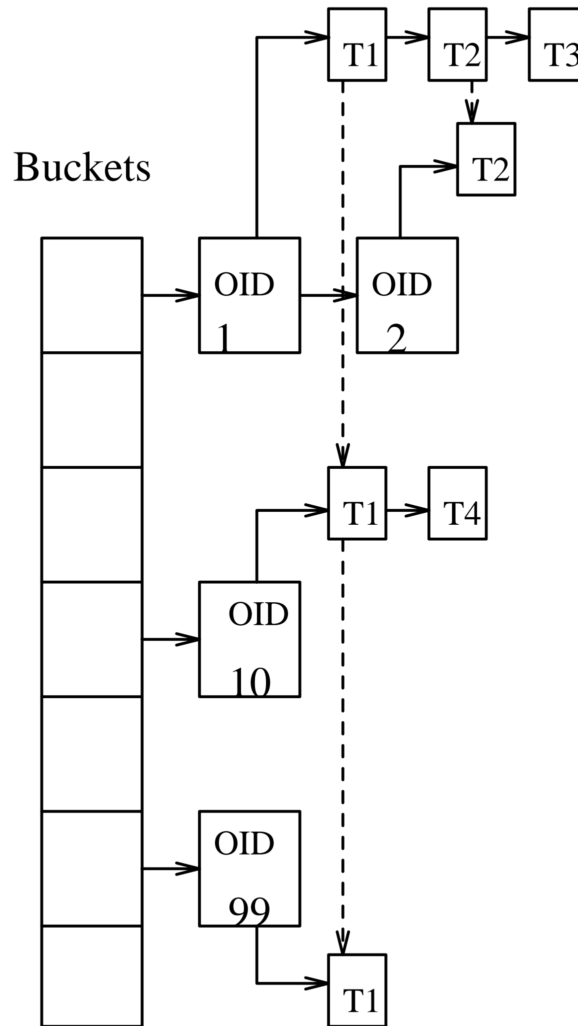

Figure 4.5. Anchored Hash Table

reduces the search when a lock request is made for an object currently not used by any transaction in that tree.

Lock table: Unlike in flat transactions where each node in the lock table (which is a hash table on the object-id as the hash attributes) corresponds to a unique object, in the case of nested transactions, an object may be used by several subtransactions resulting in multiple nodes for a given object.

In our implementation, if a transaction acquires a lock on an object, then all its superiors also (along with the transaction itself) will be represented by nodes

in the lock table. Nodes corresponding to transactions (along a path form itself to the root) not holding a lock on an object give us information regarding holdsub, retainsub, retain and grantmode on their part of the subtree. Hence in order to determine the grantability of a lock mode for an object for a transaction (node) we need to traverse along a path from itself (node) to the top level transaction(root). A naive implementation of this would be inefficient since the number of nodes in a bucket of the hash table increases by a number depending upon the depth of the tree. Also, the number of times the lock table is accessed is increased by the same amount (each acquire and release would require us to update the information to all superiors). To circumvent this problem we use an 'Anchored Hash Table' (AHT) for implementing the lock table. An AHT will have an anchor node for each object that hashes to that bucket. If more than one object hashes to the same bucket in the lock table, then we have a linked list of anchor nodes which in turn has a linked structure for the object. This anchor node serves as the virtual node for that object. The hash table is illustrated in Figure 4.5. In the figure 4.5, the broken line represents the linked list of nodes belonging to the same transaction. With the AHT, the number of nodes accessed in the worst case is

depth of tree * number of nodes with non-null hold field + m

as compared to a naive implementation whose worst case complexity is
$\sum_{i=1}^{m}$ **(depth of tree * number of nodes with non-null hold field)**

Here, m is the number of objects hashing to the same bucket. In the AHT implementation we have an anchor node for every object accessed. Therefore to grant a lock on an object we have to traverse from the anchor node to all other

nodes with the a non-null hold field. In worst case we may have to traverse the entire depth of the tree to get relevant information regarding the grantability of the lock on an object. As mentioned before more than one object may hash to the same bucket and hence we may have traverse the linked list anchor nodes and hence the factor m. In naive implementation however we have to traverse through all the nodes of the tree in the worst case.

- **Synchronization.** We have used semaphores for acquiring and releasing of locks atomically. When a lock requested is held in a conflicting mode, the (sub)transaction is required to wait on that lock i.e., it waits on a semaphore. In a flat transaction model, for a shared lock, a transaction can wait on a single semaphore with all other transactions which are waiting for the same shared lock. In case of an exclusive lock we need a unique semaphore for each transaction. In a nested transaction model, even the transactions waiting for a shared lock cannot be blocked on the same semaphore. This is because, when an existing (shared) lock on an object is released, not all transactions waiting on the same shared lock can acquire it (due to nested spheres phenomenon). Hence even for a shared lock mode each transaction has to wait on a unique semaphore. Therefore the total number of semaphores required in a transaction increases and is equal to the total number of transactions (toplevel and subtransactions).

Rule Scheduling and priority:

We have used threads to model the execution of rules. We have preferred threads over processes for obvious benefits: i) threads share the same resources unless specified otherwise, hence they facilitate applications sharing the same address space. ii) it is easy to control the scheduling by assigning priorities. iii) Overhead in creating threads is low.

We maintain a pool of free threads. Whenever a rule is triggered in the immediate

coupling mode, it gets a free thread id from the thread pool. Then the function which checks the condition and performs the action is transformed into a thread with the appropriate priority. The psuedocode for implementing the task is as follows

```
Schedule_thread()
{
  if (there is a rule to be scheduled)
    {
      priority = assign_priority();
      if( free thread is available from pool)
       {
        thread_id = get_thread();
        update_thread_pool();
        create_thread(thread_id,priority, cond_action,rule_cond,rule_action);
        }
}
```

The thread function *cond_action* executes both the condition and the action within a subtransaction. The function is packaged as follows

```
cond_action(rule_cond,rule_action)
{
  void (*cond)() = &rule_cond;
  void (*action)() = &rule_action;

  sub = begin_subtransaction(currenttransaction);
  if ((*cond)() == True)
(*action)();
   end_subtransaction(sub);
}
```

The priorities are assigned to threads based on their level (because nested rules) and the priority of the rule that triggered them. The way priorities are assigned supports depth first execution of rules. For example, if the rule which triggers has a priority of 10 and the nested rule has been assigned priority 4 (by the user), then the priority of the nested rule is calculated as 14 (10 plus 4). The sibling of this transaction will execute in a concurrent or a serial manner depending on its priority. If it is assigned the same priority (e.g 4) as its sibling then there would be a concurrent

execution else there is a serial execution.

The deferred rule is executed as a transformed rule (with an A* event) [Cha91] in the immediate coupling mode, and is signaled only when the pre_commit primitive event [Kri94] is signaled by the transaction manager and hence it is treated in the same way as an immediate rule.

The implementation of detached coupling mode stipulates that a new top-level transaction be started for executing a rule. There are two alternatives for starting a new top-level transaction: i) by starting a separate process and ii) by forking process from the triggering transaction. The first alternative has severe ramifications in the object-oriented model where the rule's condition and action could be arbitrary functions requiring the declaration of all classes. Unlike the relational model, creating an independent transaction for a rule in an object-oriented case, may be limited by the host environment (e.g., objective C, C++, Common Lisp, SmallTalk). However, for this alternative, the commit dependencies (causally detached mode) can be modeled by events spanning applications. Each transaction can signal a pre-commit and (possibly) an abort event which can be used by the global event detector to enforce abort dependencies between two top-level transactions. However this is subject to the address space issues. For the second alternative the detached rule could be forked as a process that evaluates the condition and executes the action function of the rule. In this approach, the commit dependencies can be easily established between the parent and child processes. Signals inform processes of asynchronous events. The forked process would have the copy of the resources of the parent process and hence the address space issues do not arise in this case.

CHAPTER 5
DESIGN AND IMPLEMENTATION OF THE RULE DEBUGGER

5.1 Introduction

In a conventional program debugging environment, the factors considered are variables, subroutine calls, exceptions (stack overflows), pointer referencing/dereferencing etc. The user knows the context in which to debug the program. For example, the instructions of a program are executed in a in a fixed sequence given by the programmer. If there is an error such as an overflow, non-existent pointer dereferencing etc, then the program fails and with the help of the debugger the user can locate where the error has occurred. The debugger aids the user in this process by furnishing low-level details such as the line number of the program where the error occurred, variables accessed etc.

Now when we consider debugging in the context of an Active database system, we need to take into account the following factors.

- **Database component** The database operations are performed on database objects. The database operations are carried out in well defined atomic units called transactions. In order to ensure atomicity and the correctness of the operations, locking schemes are used according to the transaction model adopted. These locking mechanisms enforce concurrency control mechanisms when several transactions compete for access to database objects. Hence to get a complete picture of what is going on in a database system we need to take into account the transactions, database objects and locks held (especially when there are concurrent operations).

- **Rule and Event Interactions** The active feature of a database system adds another dimension to the process of debugging. We have to consider the interactions among rules, between rules and events and between rules and database objects. When event(s) are raised appropriate rule(s) fire. A rule may raise an event which may cause some other rule to fire and so on. Hence we may have a nested execution of rules. An event may trigger several rules simultaneously. The rule and database objects interactions is in terms of locks acquired/released on objects in the process of rule execution.

The objective of a rule debugger for an active object-oriented database management system is the following.

- The rule debugger should concentrate on the high level details such as rule event interactions, interaction between rules and interaction of rules with database objects rather than the low-level details as in a programming environment.
- The debugger should show the execution of rules in some manner. This is particularly useful when there is nested execution of rules.
- In addition to the rule trace, the context (i.e. the events raised) should be shown.
- The debugger should allow the user to monitor only certain rules/events of his interest. This is useful in applications having a large number of rules

5.2 Design choices

Before we get into the details of the architecture and implementation, let us discuss some of the design alternatives we have. The alternatives have an impact on the overall architecture .

- **Problem oriented vs program oriented:** One approach to debugging a program is to provide a high-level description of the expected behavior of the program to the debugger. The expected behavior is terms of what the program is intended to do rather than the low level details of the program such as program variables, subroutines etc. The expected behavior is formulated in such a way that it aids the debugger in bringing out or detecting certain problems in the application domains e.g. deadlock, starvation etc in distributed programming. This would help the programmer to compare the actual behavior and the expected behavior at the execution time. This approach is adopted in debugging of parallel programs [Hse90]. The expected behavior is specified abstractly in terms of control flow, data flow and synchronization events. In particular program behaviors are described in a formal notation called Data Path Expressions (DPE). The DPE consists of events and actions. Events and relations among them specify the behavior of program execution, while the actions are performed by the debugger on program or debugger variables when the particular behavior is recognized during execution. This is useful to detect race conditions, deadlocks and starvation in a parallel program. This in contrast to the conventional debugging approach (of conventional programs) as exemplified by dbx [Lin81] where the program behavior is modeled at a low-level, in terms of the source code entities such as subroutine names, line numbers etc. In this approach the programmer is responsible for comparing the actual and expected behavior of the program. As categorized in [Hse90] the former approach is termed as *problem-oriented* and the latter low-level approach is termed as *program-oriented*. When we try to debug/visualize rule execution and event detection in a active database management system, the *problem-oriented* approach is desirable . This is due to the fact that rule execution is dynamic i.e.

rules are fired in response to a situation without any user intervention. Due to the event based nature of rules the user does not have any apriori knowledge of rules that will be fired in a given execution of the application. Hence the "context" of program/application is not available with user as in case of a conventional debugging model. The "context" has to be provided by the system. Also the expected behavior of rule execution is specified in the event specification and rule specification language [Kri94, Cha91]. The specification details rule(s) are fired in response to event(s).

- **Integrated vs Coupled:** When the Rule debugger is an integral part of the underlying database management system, it is termed as the integrated approach. The integrated approach favors rule debugging per application as opposed to global rule debugging (across applications). Also in the integrated approach since we are operating within the same address space it easy to show the modification of data (attributes) at least in certain object-oriented environments such as Objective C and Smalltalk.

  In contrast in a Coupled approach the Rule debugger would be envisaged as a separate system which would be linked to the underlying database system by means of some communication mechanism (sockets , pipes, RPC etc). The Coupled approach favors global monitoring of rules, that is, we could have event detection and execution of rules across applications. An application may be interested and respond to events generated in some other application. For example, a stock broker application may respond to the event of decrease in IBM stock in a stock exchange application. The main advantage of this approach however is that since it can built as a separate module independent of the active database system, it will not be constrained by the features of the latter. The main disadvantage with this approach however particularly in an

object oriented context is that we are operating in different address spaces. This makes tracking of data modification difficult since it would involve issues like accessing of remote objects.

- **Online Vs Postanalysis:** We can visualize events and rules as and when they occur or we can do so over a stored event/rule log i.e. in a batch mode. It is probably desirable to have both the options in a /debugging environment. To support online we need to have some mechanism by which there is instantaneous notification of event occurrences and firing of rules as and when they occur to the Rule debugger. In a Coupled approach this could be achieved by means of socket mechanism.

5.3 Overall Architecture

The architecture for the Rule debugger goes is dictated by the architecture for event detection in Sentinel. It is helpful to highlight the architecture for event detection before we discuss the architecture for the Rule debugger. Each application has a local event detector to which all the primitive events are signaled. In addition to this each application has a thread that handles the execution of rules whose events span across applications. This is the global event handler thread. When a primitive event occurs it is sent to the local event detector. The application waits for the signalling of a composite event that is signalled in the immediate mode. The global event detector communicates with the local event detector for receiving events locally and with the application's global event handler for signalling the detection of global events. The overall architecture is as shown in the figure 5.1. As illustrated in the figure 5.1 the Rule debugger communicates with the local event detector and the rule manager of each application. Since the local event detector is aware of the event object and the rule object ids, also it detects the occurrence of an event and notifies the appropriate rule to be fired, the runtime information about the event occurrence

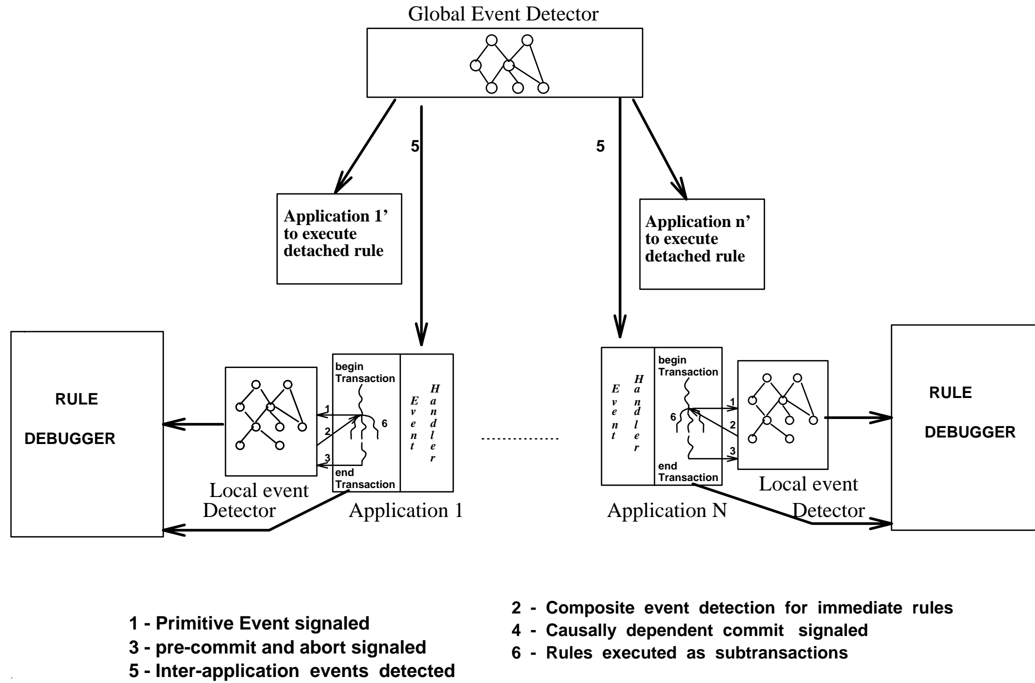

Figure 5.1. Overall Architecture

and rule execution is obtained here. From the rule manager we get the transaction ids of the subtransactions within which the rules are executed.

The Rule debugger can also monitor global events in the following way. Since the global event detector communicates with the local event detector, the local event detector is notified of the occurrence of global events.

In this thesis we have implemented the Rule debugger to monitor rules and events within an application. We have adopted a *problem oriented* and *coupled approach* to debugging. Since we have adopted a coupled approach, there needs to be a means of communication between the debugger and the active database system. The way communication is achieved is via log files. The log files contain information regarding rule/event definitions, rule/event object id's, the actual occurrence and firings of events and rules respectively and other transaction related information. From the rule/event definition we can infer which rule subscribes to which event and hence

rule-event interactions. The transaction related information gives the rule-database interactions. The information in the log files is furnished by the pre-processor, event detector and the rule manager. The Rule debugger then reads these files to obtain the trace of event occurrence and rule execution. It is important to note that there is no actual runtime visualization done by the Rule debugger. We can say that runtime visualization is achieved when the Rule debugger shows the occurrence of events and rule firings as and when they happen. If this were to be achieved then the communication between the Rule debugger and the database system has to be done at real-time using sockets, named pipes etc. However currently, the Rule debugger gathers information from the log files and simulates runtime visualization. It makes only a single pass on the log files to obtain the relevant information.

The visualization of global monitoring of rules is not addressed in this thesis. The following are some of the issues which need to resolved

- The specification of the global events.

- There may be some difficulties in obtaining the object ids especially of complex events. In the current implementation of the event detector, it can supply both the event name (user supplied) and the object id of the primitive event at runtime. However in the case of complex events the situation is different. Essentially a complex event is an event expression of primitive events. Hence detection of complex events is done using an event graph. The leaf nodes are primitive events and from thereon the graph is built in a bottom-up fashion. The intermediate nodes may constitute a complex event. The name of the complex event nodes is associated with event operator (AND, SEQ, OR) and not the user supplied name of the complex event. Hence the only way to identify a complex event is by its object id which is generated when the event object is created. Within a application this information is easily accessed since we

operating in a single address space. In a global event monitoring situation the global complex event object is created in a different application (different address space with respect to the application being visualized).

5.4 Implementation

In this thesis we have implemented a debugger for active rules in a object-oriented context. The rule manager supports the event-condition-action paradigm as mentioned before. Apart from tracing the execution of rules the Rule debugger also keeps track of the events. As mentioned in [Dia93] the tracing of events gives important hints to the user: the event-rule cycle allows the user to not only to know which rules are fired but also which event(s) caused the rule(s) to fire. The occurrence of the events sets the context for the rule execution. Sentinel supports active behavior in terms event-condition-action rules. The following features of the rule manager of Sentinel have influenced the development of the Rule debugger.

- Events and Rules in Sentinel are treated as first class objects. Hence rule and event definitions are identified by their object identifiers, described by attributes and are manipulated through methods.

- Both events and rules can either be defined either at class level or at instance level. The class level rules fire for all objects of that particular class when a certain situation (event-condition) is satisfied. The instance level rule fires only for a specific object.

- Events can be either primitive or composite. Currently, Sentinel supports primitive events as behavior invocations: either as global functions or as member methods. The system supports composite events by incorporating all the operators defined in Chakravarthy and Mishra [Cha91].

- As mentioned in the previous chapters the condition and action are packaged into a single thread of execution. This thread is executed as a subtransaction of the parent triggering transaction. Hence we could typically have a cascaded firing of rules. Hence there is a one to one mapping between the subtransactions and the rules.

- The nested transaction model supported by Sentinel allows for sibling concurrency.

- The event interface is described through the following attributes: the *method* which is to be monitored, *when* the event is to be raised *before* or *after* the method execution, and the *class* the objects of which are sending messages to be detected.

The functional architecture of the Rule debugger is as shown in the figure 5.2. The

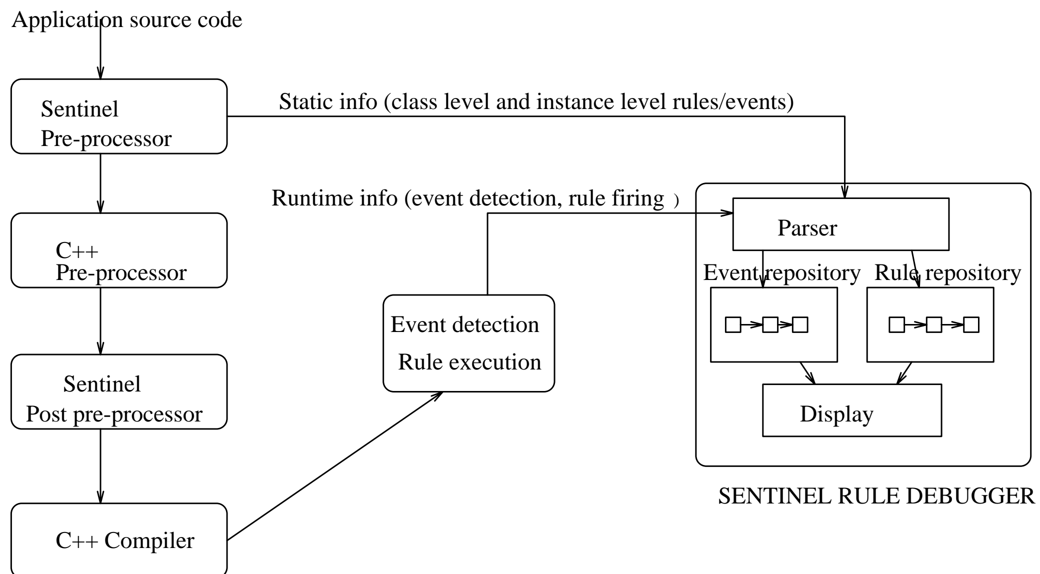

Figure 5.2. Functional Modules

input to the debugger is threefold:

- First is the class level and the instance level rule and event definitions. This information is supplied by the user in the application program using the event[Cha91] and rule definition [Anw92, Kri94] language. Since we have incorporated the event and rule definition language on the top of C++, we preprocess the definitions. During this process the preprocessor supplies this static information in the form of a file to the Rule debugger. The debugger requires the following information with respect to an event

  1. The user supplied name of the event.
  2. If it is a class level event then the classname with which the event is associated.
  3. The type of the event whether it is primitive or complex.
  4. The signature of the method on which the event is defined along with the event modifier whether it is raised before or after the invocation of the method.
  5. If the event is complex then the event expression.

  Also the following information is provided to the debugger with respect to a rule

  1. The user supplied rule name and the classname to which it is associated if it is classlevel rule.
  2. The signature of the methods implementing the condition and action of the rule.
  3. The context for which the rule id fired, coupling mode, rule trigger mode and the rule priority.

  Consider the following example:

```
class STOCK : public REACTIVE {
      private:
      ......
      public:
      ......
      event end(e1) int sell_stock(int qty);
      event begin(e2) && end(e3) void set_price(float price);
      int get_price();
      event e4 = e1 ^ e2; /* AND operator */
      /* class level rule */
      rule R1[e4, cond1, action1, CUMULATIVE, DEFERRED];
};

int STOCK::sell_stock(int qty) { ................ }
void STOCK::set_price(float price) { ................ }
int STOCK::get_price() { ................ }
/* Main program */
STOCK IBM, DEC, Microsoft;
main()
{
      .................
      /* Creating instance level primitive event */
      event set_IBM_price("set_IBM_price",IBM,
                              "begin","void set_price(float price)");
      /* SEQUENCE operator */
      event seq_event = STOCK_e4 << set_IBM_price;

      /* Creating a rule which contains both class level
      and instance level events */
      rule R2[seq_event, cond2, action2,,,20, PREVIOUS];
      ...................

      OpenOOBD->beginTransaction();
            IBM.set_price(115.00);
            DEC.set_price(100.00);
            Microsoft.sell_stock(200);
            DEC.get_price();
            IBM.set_price(75.95);
      OpenOODB->commitTransaction();
}
```

For the above application, file class.STOCK would contain the following

Event Stock Primitive Stock_e1 [int sell_stock(int qty)]

Event Stock Primitive Stock_e2 [void set_price(float price)] begin

Event Stock Primitive Stock_e3 [void set_price(float price)] end

Event Stock Complex e4 Stock_e1 AND Stock_e4

Rule R1 Stock cond1 action1 CUMULATIVE DEFRRED NOW Stock_e4

Event Null Primitive set_IBM_price [void set_price(float price)] begin

Event Null Complex seq_event STOCK_e4 SEQ set_IBM_price

Rule R2 Null cond2 action2 RECENT IMMEDIATE PREVIOUS 20 seq_event

- Secondly, the event detection information is furnished. This is the runtime information on the occurrence of events and the creation of event objects. This information is obtained at two points in the execution of the program. First, when the event objects are created, the event name and the event object oid is given to the rule debugger. This information is used later by the rule debugger to associate the event object oids with their user supplied names. As mentioned previously the object id information is particularly useful in the case of complex events. Finally when the event is actually raised it is notified to the rule debugger by the event detector.

- Thirdly, the rule firing information is furnished to the rule debugger in the same way as the event detection information. The rule object identifiers and the user supplied rule names are associated when the rule objects are created. In addition to the rule firing information, the transactions in which rules where fired, the locks acquired/released on database objects is also furnished. This helps the rule debugger to associate the rules in the transactions in which they were fired and also the objects which were accessed in the process of rule execution.

It may be noted that the event detection and rule firing is made available due to the fact events and rules are considered first class objects. All the information mentioned above is given to the rule debugger in the form of two files. One of the files contains the static information on class and instance level events and rules. The second file contains the runtime information on event and rule object oids, the occurrence and firing of rules and the transaction and object access information. Below we present a sample application program.

```
   STOCK IBM, DEC, Microsoft;
   LOCAL_EVENT_DETECTOR *Event_detector

   main()
   {
  ................................
        .........................

      EVENT *STOCK_e1 = new PRIMITIVE("STOCK_e1","STOCK","end","int sell_stoc
                                      int qty)");  ------   1
      EVENT *STOCK_e2 = new PRIMITIVE("STOCK_e2","STOCK","begin","void set_
price(float price)"); ----- 2
      EVENT *STOCK_e3 = new PRIMITIVE("STOCK_e3","STOCK","end","void set
price(float price)"); ---- 3
      EVENT *STOCK_e4 = new AND(STOCK_e1, STOCK_e2); ---- 4
      RULE *R1 = new RULE("R1",  STOCK_e4,cond1,action1,CUMULATIVE); ---- 5
      PRIMITIVE *set_IBM_price = new PRIMITIVE("set_IBM_price", IBM,"end","
void set_price(float price)"); ------ 6
      EVENT *seq_event = new SEQ(STOCK_e4,set_IBM_price); ------ 7
       RULE *R2 = new RULE("R2", seq_event, cond2, action2,RECENT) --- 8

     OpenOODB->beginTransaction();
        IBM.set_price(115.00);  ------ 9
        DEC.set_price(100.00); ------ 10
        Microsoft.sell_stock(200) ---- 11
        DEC.get_price(); ------ 12
        IBM.set_price(75.95); -------- 13
      OpenOODB-Commit();
 }
```

The above is the preprocessed program. The event and rule definitions are converted to appropriate C++ statements. At line 1 when the event object STOCK_e1 is

created, its object id is known. This information is associated with the user supplied name of the which was obtained from the static information. Similarly the object ids of the rule objects R1 and R2 are associated with the user supplied name. Now when the events and rules are actually fired there object ids are notified to the rule debugger. Since rule executes within a subtransaction, the transaction information is also provided to the rule debugger. With the help of the transaction information the Rule debugger displays the nested execution of rules. The runtime information for the complex event STOCK_e4 provided to the Rule debugger would be the following.

```
EVENT STOCK_e4 oid_e4
RULE R1 oid_r1
oid_e4 raised
Started Subtransaction tid
.......
oid_r1 fired
......
EndSubtransaction tid
```

The data structure which captures the nested execution of rules is an n-ary tree. The root node represents the toplevel transaction of the application. When this transaction triggers a rule and since rules are executed as subtransactions, the child node of the toplevel transaction represents the rule fired. This node in turn could trigger another rule and it is represented as the child node (subtransaction) and so on. In the current implementation each node (apart from the root node) represents a subtransaction. If a rule is fired within a subtransaction it is also indicated appropriately. Also the information about objects accessed, locks held by a transaction is associated with each node. This information is dynamic and is updated at each step of the nested execution. The algorithm for displaying the nested execution of rules/subtransaction is given below

```
Display()
{
get_token();
while(token->type != EOI)
 {
   switch(token->type)
    {
      case BEGIN_TX:
        get_token();
        tid = atoi(token->val);
        cursor = create_node(tid,level);
        head = cursor;
        head->parent = NULL;
        head->next = NULL;
        draw_node(head);
        get_token();
        break;

      case SUB_TX:
        get_token();
        tid = atoi(token->val);
        temp = create_node(tid,level);
        parent = find_parent(tid/100,head);
        parent->status = SUSPEND;
        alloc_child(parent,temp);
        temp->parent = parent;
        cursor = temp;
        set_coor(cursor,head);
        draw_node(head);
        get_token();
        break;

      case SUB_COMMIT:
        get_token();
        tid = atoi(token->val);
        update_cursor(tid,head);
        draw_node(head);
        get_token();
        break;

      case T_COMMIT:
        get_token();
        tid = atoi(token->val);
        update_cursor(tid,head);
        draw_node(head);
        get_token();
```

```
        break;

      case ACQ:
        get_token();
        lockmode = atoi(token->val);
        get_token();
        append_object(cursor,token->val,lockmode,0);
        get_token();
        break;

      case REL:
        get_token();
        lockmode = atoi(token->val);
        get_token();
        append_object(cursor,token->val,lockmode,0);
        get_token();
        break;

      case RULE_TK:
        get_token();
        strcpy(cursor->rule, token->val);
        draw_node(head);
        get_token();
        break;

      case EVENT_TK:
        get_token();
        break;

      default:
        break;
      }
  }
}
```

When a transaction is started, a node is created in the tree and its status is set to “ACTIVE”. When the transaction commits its status is set to “COMMIT”. In addition whenever a subtransaction is started the status of its parent is set to “SUSPEND”. To distinguish between these states different colors are used while drawing the nodes. Each node in the display is represented as a rectangle. For each step of the nested execution the tree is redrawn. The following recursive alogorithm is used

to draw the tree.

```
draw_node(head)
{
    switch(head->status)
     {
        case ACTIVE:
         set_color(``green'');
         draw_rectangle();
         if (head contains a rule) display rule details;
        case COMMIT:
         set_color(``red'');
         draw_rectangle();
        case SUSPEND:
         set_color(``yellow'');
         draw_rectangle();
     }
     if (head->next != NULL) draw_node(head->next);
     if (head->sibling != NULL) draw_node(head->sibling);
}
```

Presently the nodes represent only rules. The events which cause these rules to fire are made available to the user in the form of statement on one side of drawing screen. Now one way of detecting a potential cycle is that if in a single branch of the tree, we have a repeated occurrence of an event causing the same rule to fire, then this may lead to a potential cycle.

Thus far we have seen that all rules and events produced are shown by the Rule debugger. This can become quite inconvenient for the user if he/she is dealing with large number of rules. Potentially we can identify two ways of pruning the tree: the user can either choose the rules or events he wants to monitor. This is possible in our case since events and rules are objects and more over there is a one to one correspondence between rules and transactions. When the tree is pruned in this way

the child node may not be an immediate subtransaction of the parent node rather a descendant of the parent transaction.

5.5 Functionality

The following features are implemented in the Rule debugger.

- **Trace:** This is one of the menu choices in the Rule debugger. When this button is activated by the mouse click the runtime trace of the transactions is shown. The trace is shown on step by step basis or in a continuous mode. In the step mode after each node is drawn, the display routine checks for a buttonpress event (mouse event) and then the next node is drawn. Although each node represents the subtransaction since there is a one to one mapping between the subtransactions and the rules the nodes also represent the rules. The color of the nodes of the tree changes at each step of the execution. Green color represents that the transaction is active, yellow represents that the transaction has suspended, and red represents that the transaction has committed. A sample trace is illustrated in figure 5.3.

- **Rules** This menu option when selected list all the rules defined for the particular application. From the list of rules when one of the rules is selected the description of the rule pops up.

- **Events** This has a similar functionality as that of Rules. When selected it lists the events to be monitored in the application. When one of the events is selected from the list the complete description of the event is shown. The event description consists of the event name, type of the event: primitive or complex, the method which causes the event to be raised, if it is class rule then the class name for which the event is defined, if the event is complex then the event expression is also shown.

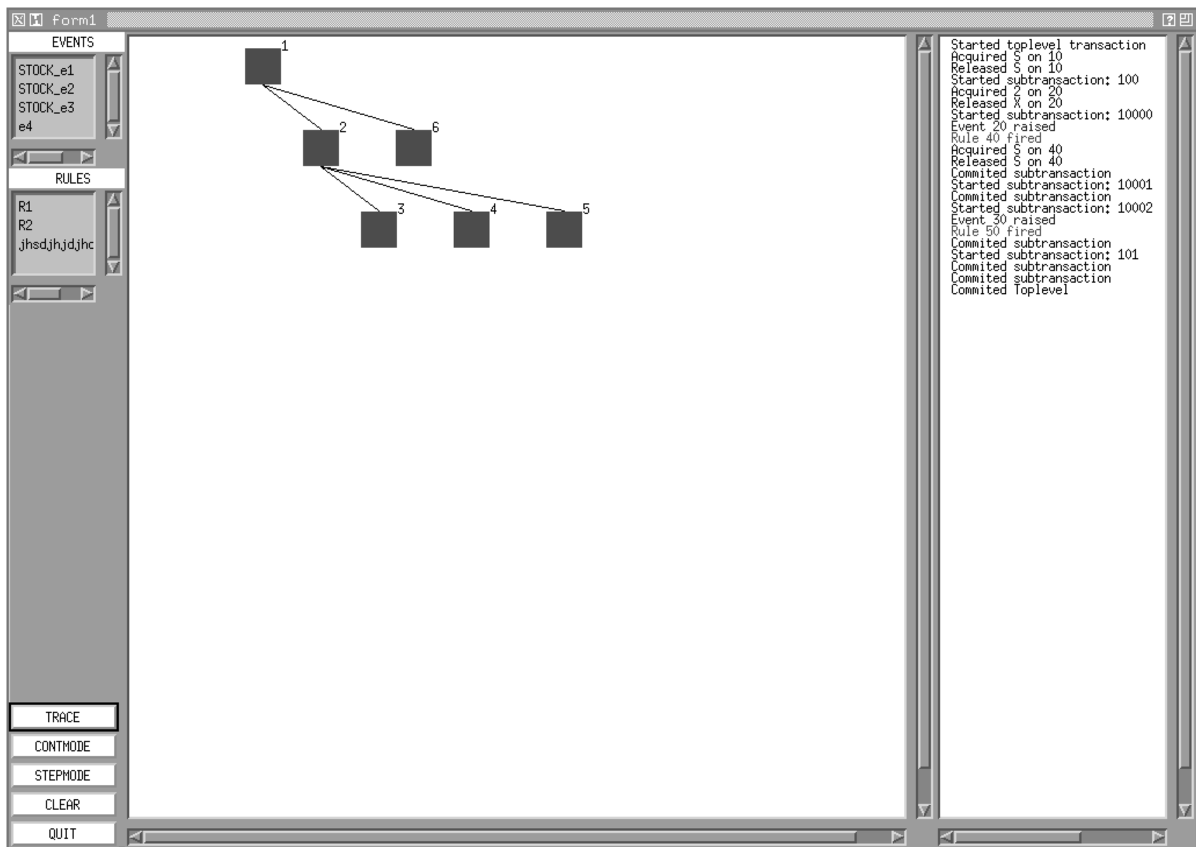

Figure 5.3. Rule execution trace

- **Clear** This button clears the run-time trace of the rule execution on the drawing area.

- **Rule Select** This option allows the user to select the rules to monitor.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

In this thesis we have implemented a rule execution model for an active object-oriented database management system. The extended nested transaction model was used to govern the execution of rules. We have extended the transaction manager of OpenOODB to incorporate the nested execution of rules. We have used threads to model the execution of rules. We have preferred threads over processes for obvious benefits. In our implementation condition and the action part of the rule are packaged in a single thread of execution. This ensures a correct serializable execution of rules without extending the serializability criteria. We have also achieved sibling concurrency in our implementation. Our implementation was governed by strict object oriented design principles and hence is extensible.

The second major contribution of thesis is the design and implementation of the Rule debugger for debugging rules. We have presented the overall architecture keeping view the requirements of an active database system. In this thesis, however, we have designed and implemented the local Rule debugger. The debugger supports runtime visualization of rules and post-analysis of rules. We have adopted an problem oriented approach to debugging of rules. In addition to merely showing the trace of rules the debugger also furnishes the context(events) in which the rules are fired albiet in a primitive manner.

Although the implementation of the underlying nested transaction model supports sibling concurrency, the design and implementation of a scheduler to schedule concurrent execution of rules at each level of the transaction hierarchy remains a

challenge. We have to necessarily adopt a conflict resolving approach since the conflicts between rules are known only at runtime.

Event detection in OpenOODB is done using an Event graph. The leaf nodes of the event graph are primitive events and from thereon they are generated in a bottom up fashion. The same graph however is used for different context (Cumulative, Recent, Chronical). In the current implementation of the Rule debugger the event graphs are not shown. An immediate extension to the toolkit would be to support the visualization of the event graphs which would make the context more explicit.

Currently the Rule debugger supports only runtime analysis of rule execution. A long term plan would be incorporate static analysis tools in the Rule debugger.

REFERENCES

[Anw92] E. Anwar. Supporting complex events and rules in an oodbms: A seamless approach. Master's thesis, Computer & Information Sciences Department, University of Florida, Gainesville, 1992.

[Anw93] E. Anwar, L. Maugis, and S. Chakravarthy. A new perspective on rule support for object-oriented databases. In *Proceedings, International Conference on Management of Data*, pages 99–108, Washington, D.C., May 1993.

[Bad93] R. Badani. Nested transactions for concurrent execution of rules: Design and implementation. Master's thesis, Computer & Information Sciences Department, University of Florida, Gainesville, 1993.

[Cha89a] S. Chakravarthy. Hipac: A research project in active,time-constrained database management. Technical report, Xerox Advanced Information Technology, 1989.

[Cha89b] S. Chakravarthy. Rule management and Evaluation: An Active DBMS Perspective. *Special issue of ACM Sigmod Record on rule processing in databases*, 18(3):20–28, Sep. 1989.

[Cha91] S. Chakravarthy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical report UF-CIS TR-91-23, University of Florida, E470-CSE, Gainesville, Sep. 1991.

[Dia93] O. Diaz, A. Jaime, and N. W. Paton. Dear: A debugger for active rules in an object-oriented context. In *Proc. of the 1st International Conference on Rules in Database Systems*, San Diego, September 1993.

[Hae83] T. Haerder and K. Rothermel. Concurrency control issues in nested transactions. IBM Research Report RJ5803, NY, Aug. 1983.

[Hse90] W. Hseush and G.E. Kaiser. Modelling concurrency in parallel debugging. In *ACM S IGPLAN Notices*, number 3, pages 11–20, March. 1990.

[Hsu88] M. Hsu, R. Ladin, and D. McCarthy. An execution model for active data base management systems. In *Proceedings 3rd International Conference on Data and Knowledge Bases*, Washington, D.C., Jun. 1988.

[Kri94] V. Krishnaprasad. Event detection for supporting active capability in an OODBMS: semantics, architecture, and implementation. Master's thesis, Computer & Information Sciences Department, University of Florida, Gainesville, 1994.

[Lin81] M. Linton. A debugger for the Berkeley pascal system. Master's thesis, University of California at Berkeley, June 1981.

[Mos81] J. Moss. Nested Transactions: An Approach To Reliable Distributed Computing. MIT Laboratory for Computer Science, Cambridge, MIT/LCS/TR-260, 1981.

[Tex93] Texas Instruments. Open OODB toolkit, Release 0.2 (Alpha) Document, Austin, September 1993.

[Wel92] D. Wells, J. A. Blakeley, and C. W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–81, October 1992.

[Wid90] J. Widom and S. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. of ACM-SIGMOD*, pages 259–270, May 1990.

BIOGRAPHICAL SKETCH

Ziauddin Tamizuddin was born on August 15, 1967, at Bangalore, India. He received his undergraduate degree in electrical engineering from the University of Madras, India, in April 1989. He also received his graduate degree in business administration from the Regional Engineering College, Trichy, India, in May 1991. He will receive his Master of Science degree in computer and information sciences from the University of Florida, Gainesville, in August 1994. His research interests include active, object oriented databases and graphical user interfaces.