SCHEDULING RULES IN AN ACTIVE DBMS USING NESTED TRANSACTIONS

By

SHASHI NEELAKANTAN

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

UNIVERSITY OF FLORIDA

1998

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Sharma Chakravarthy for giving me an opportunity to work on this challenging topic and for providing continuous guidance, advice, and support throughout the course of this research work. I thank Dr. Eric Hanson and Dr. Herman Lam for serving on my supervisory committee and for their careful perusal of this thesis.

I would also like to thank Sharon Grant for her tireless efforts in maintaining the database center and making it a great place to work.

I sincerely thank Hyoungjin Kim and Shiby Thomas for their invaluable help throughout the course of this theses and the patience they showed whenever they helped me solve problems. Also contributing towards this work is the helpfulness of all my friends here in Gainesville, who made my stay here a very pleasurable one.

I would like to thank the Office of Naval Research and the Navy Command, Control and Ocean Surveillance Center RDT&E Division for supporting this work.

Last but definitely not the least, I would like to thank my father, mother and brother. Without their unconditional love and support all this would have been impossible.

TABLE OF CONTENTS

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

SCHEDULING RULES IN AN
ACTIVE DBMS USING NESTED TRANSACTIONS

By

Shashi Neelakantan

December 1998

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

Database management systems (DBMS) are moving from the traditional systems to more advanced systems that represent real-world problems. These systems are required to react to situations without user or application intervention. This is the very goal an active DBMS strives to achieve and Sentinel is being developed at the University of Florida to support all the functionalities of an active DBMS. The active capability is brought about by the powerful rule based system, which includes event detection coupled with rule processing. We already have an event detection mechanism that detects both local and global events. This thesis deals with the scheduling of rules that have been triggered by the detection of an event and the satisfaction of the condition clause of the rule. Scheduling is required to satisfy the priority associated with each rule as well as to schedule rule execution based on their relative priorities. Scheduling is also required to take care of the coupling mode (Immediate, deferred and detached) semantics of the triggered rule.

Scheduling is sufficient to handle priority-based rule execution. However, when several rules can be executed in parallel and when rules are deferred, we need an extended transaction model to generate the rule execution semantics. The parallel/concurrent execution brings along concurrency control issues that needs to be addressed. In this thesis, the nested transaction model is used for that. In addition the transaction semantics is extended to handle deferred execution of rules. Nested transactions ensure that concurrent rule execution is serialized using the nested transaction semantics.

CHAPTER 1
INTRODUCTION

There has been a lot work going on in the field of Active DBMS as it supports many real world applications that cannot be handled well using conventional DBMS.

Conventional DBMS's perform updates and executes queries using a demand based mechanism, either when application programs are executed or when interactive users perform some operation. This was one of the features an Active DBMS improves upon, by monitoring and reacting to pre-defined situations automatically without user/application intervention. This feature lends the name Active to a DBMS and those DBMS's without this capability are called passive.

The mechanism to bring about this active capability is the use of ECA rules. The E stands for events, which are the indicators that needs to be reacted to. These events can be one occurrence of a particular method called primitive events or a group of such methods which happen in a certain sequence and are called composite events. When such an event takes place a boolean condition, which is a query based on the state of the database, is executed. If the condition evaluates to true, an action is executed which might or might not affect the state of the database.

An Active DBMS offers three new features over a conventional DBMS:

- Rule Interface: This allows applications to define ECA rules.
- Event Detector: This is the entity which monitors applications as well as the database to detect the occurrence of primitive events. These primitive events can then be grouped based on some predefined criteria to detect composite events.

- Action Execution: The rules are scheduled and executed based on the execution semantics of the ECA rule

Sentinel is an active OODBMS, it uses Snoop [1] as its event/rule specification language and provides various parameter contexts or event-consumption modes for detecting composite events to meet the requirements of a wide range of real-world applications.

As mentioned earlier a set of rules have to be specified to provide active capability. These rules include the event, condition and action specification as well as other relevant information, such as rule priority and coupling mode.

This theses concentrates on the scheduling of rules once an event has been detected and the rules subscribing to those events are identified. The scheduler allows the condition and action execution based on some scheduling policies.

Rule execution as part of a transaction is different from rule execution in expert systems. There are different problems encountered here which includes rule priorities and coupling mode.

When a rule is specified the user has to specify a priority associated with it. Thus in case there is more than one rule triggered by an event, an ordering has to be imposed on rule execution using priorities. In an application, when a rule is executed, the action portion of the rule in turn could trigger an event (and possibly a rule). Thus there are cases where there are several levels of nested execution of rules. The scheduler has to decide the order of rule execution based on relative priorities of rules.

Another important factor to be considered is the coupling mode. We support two coupling modes: immediate, where the condition-action is executed as soon as the event occurs, and deferred, where the parent transaction rule continues execution even after the rules are triggered. The rules are then activated at the end of the triggering transaction.

It has already been stated that there can be more than one rule associated with an event and an action can generate an event in turn. This situation is best implemented if you can allow more than one rule to execute at the same time. The semantics of rule execution can be accomplished using an extended nested transaction model [2]. This thesis focuses on the implementation of rule execution using the nested transaction model implemented for open OODB.

In our implementation, each rule is going to be executed as a thread and multiple threads can be active at the same time. There has to be some concurrency control mechanism that makes sure that the same data is not accessed by more than one thread and changed. This leads to inconsistencies that needs to be avoided. In our system, each thread of execution is a sub-transaction. The use of nested transactions brings about many advantages as described in chapter 4.

The use of rule priorities and coupling modes for the execution of rules has been addressed in this thesis. The scheduler that has been implemented provides rule execution semantics and achieves concurrency using nested transactions.

CHAPTER 2
RELATIVE WORK ON RULE PROCESSING

Different systems use different approaches for rule processing. One system that uses rules as its basic concept of processing is a production system, of which OPS5 is a typical example.

2.1 Architecture of a Production System

The components of a production system model are the same as that of a procedural model, only differing in the details of these components. In a procedural model, a *program* is an ordered sequence of instructions. The *program manager* carries out these instructions in serial order except when explicitly specified as part of a program instruction. In contrast, a production-system program consists of an unordered collection of basic units called rules [3].

The production system architecture typically includes components that are as shown in 2.1:

- *data memory*: acts as a global database containing symbols that represents facts and assertions about the problem. In this type of system data are instances of objects. These objects are either physical objects or facts related to the domain of application or conceptual objects (such as goals) related to the problem solving strategy [3].

- *rule memory*: the whole program is made up of a set of rules. Each rule is made up of two parts. The *Condition* which describes the data configuration for which the rules is applied. Thus if the data configuration matches the condition part, the condition turns out to be true else the condition is False. The second part of the rule is an *Action* part, which is reached if the Condition is evaluated to true. The action is usually some set of instructions that change the data configuration.

- *inference engine*: This is the rule manager. The rules are fired depending upon the data memory configuration at that particular instance. The inference engine chooses the most suitable rule that applies based on a particular matching criteria. This process is also called conflict resolution.

2.2 Inference Engine

As shown in 2.1 the inference engine is the most powerful part of a production system. It can be described as a finite-state machine with a cycle consisting of three action states: *match-rules, select-rules and execute-rules* [3].

In the first part, match-rule, the engine finds all the rules that satisfy the contents of the data memory at that instant using a comparison algorithm built into the inference engine. All rules that match become part of the conflict set and are all potential rules for execution. The same rule can appear multiple times in the conflict set if it matches different data items. The next stage is when the engine applies a selection strategy (also termed resolution strategy and varies among the different production-system models) and determines the rule to be executed. The last stage is the rule execution stage where the rule selected, is executed. The execution of a rule changes the data memory and thus a different set of rules will be selected and executed. We can see that computation in a production-system is data driven instead of instruction driven which is the most common approach.

As we can see from the working of the inference engine, the conflict resolution stage is of utmost importance. OPS5 handles conflict resolution in two ways. One method is to use the idea of *recency*. Here when a number of rules match different elements of the working memory, the choice is narrowed down to those rules that match the most recently updated memory element. Each memory element has a time tag associated with it, and when modified, the time tag changes. So basically the rules that match memory elements with the most recent time tags are preferred over older ones.

Another part of conflict resolution is the idea of *specificity.* Each condition has one or more elements that needs to be matched with the elements in the memory. If there are more than one rule in the conflict set, each rule condition is checked and the one that has most elements matching is chosen. This step usually comes after the *recency* check, so there are only a limited number of rules that are in the conflict set.

Another way of reducing the conflict set is rule ordering. Here the rules in the rule set are ordered as desired by the user and the first rule that matches an element in the data memory is selected.

Thus production systems like OPS5 have a different approach to programming. The rule based approach has given us a lot of insight into how rules are incorporated into a system and how they are processed.

2.3 Active Databases

An active database needs to have the capability to dynamically react to changes in the database, this makes them similar to an OPS5 like system. The basic difference lies in the fact that the rules in OPS5 are triggered by comparing the data memory with rules in the rule-base while in an active database the rule is triggered when a specified event occurs. The execution of the method is an indicator of occurrence of an event which initiates the firing of a rule.

An active DBMS generally follows the ECA pattern of rule processing. ECA consists of events, conditions and actions. The condition and action part are similar to that in OPS5, what is different is how the rule is fired.

Unlike in OPS5, there is a notion of a transaction in a database system. Rules get triggered within transactions and the rule processing itself should conform to the transaction semantics. It should also allow coupling modes and priorities for the sake of flexibility and expressiveness.

Rules are not an essential part of the system unlike OPS5. Rules are triggered only when the event detector detects an event that is part of a rule specification. The rule specification is done by the user at the start of an application. The user has to specify the event, condition, action and other attributes such as priority and coupling mode.

2.4 Rule Execution in Sentinel

Rule execution in Sentinel is done as part of a series of steps, the first being pre-processing. After the application is written and the rules defined, the code is run through a pre-processor that inserts code for event detection based on the method name which was given as part of the rule specification. This event specification is then wrapped and the rule processor code is called after the event is detected.

During the execution of the application, when it reaches an event method, the corresponding event occurs. The condition to be checked, as in OPS5, could be some data value stored in the database. If this condition evaluates to true then, as in OPS5, the action part is executed.

Similar to OPS5, there may be more than one rule that qualifies for execution, which is the case when more than one rule specification has the same event method that triggers it. Unlike OPS5, there is no conflict resolution as priorities are provided at rule specification time. The user can specify the order in which the rules execute. This is done by assigning priorities to each rule and the rule with the highest priority is executed first followed by lower priority rules, in that order.

Also each rule is executed in a specific coupling mode (more in chapter 3) i.e. the rule can execute immediately after it was triggered or at the end of the transaction that triggered it.

Conventional top level transactions proceed serially while rules can execute concurrently. Using a nested transaction model allows sub-transactions to execute concurrently but unlike conventional nested transactions, rules are not processed as if they are all equal. Each rule has a priority and the order of rule execution depends on their priorities.

Each rule is executed as a separate thread and is part of a sub-transaction. The need for making it a sub-transaction is to preserve concurrent rule execution semantics (explained in detail in chapter 4).

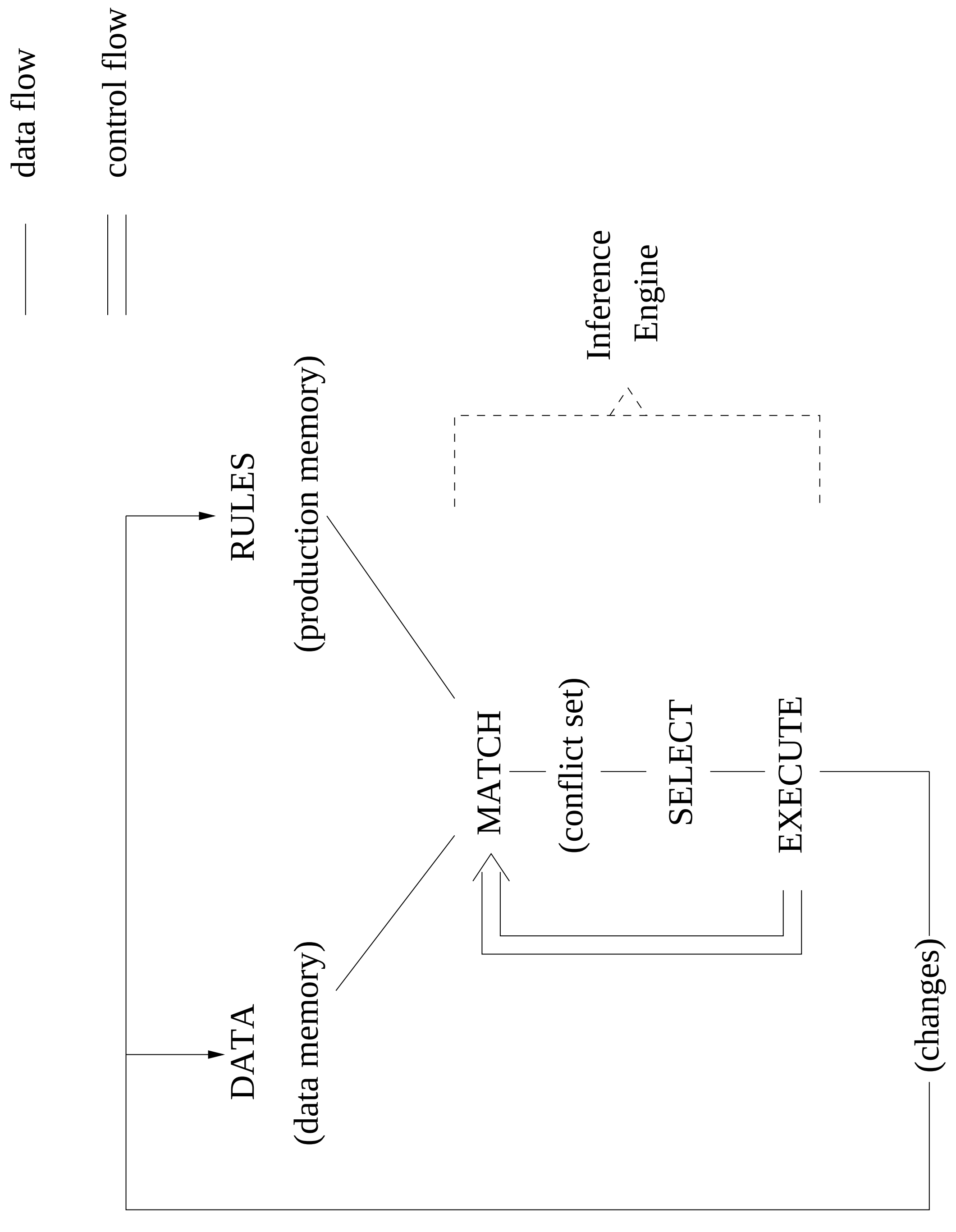

Figure 2.1. Architecture of a Production System Model

CHAPTER 3
ECA RULES

The design of any Active DBMS involves specifying rules as these are fundamental to provide reactive capability. Rules have many components that need to be specified individually. The primary components are: event, condition and action. Below, we examine their roles and specifications.

3.1 Events

Events are instantaneous, atomic (happens completely or not at all) occurrences [4]. Events represent state changes that are induced by database operations. Each message sent to an object is a potential event. Events are classified into

- *Primitive events*: are the simplest form of events detected by the system. Primitive events are the building blocks from which composite events are formed and detected.

- *Composite events*: are formed by applying a set of operators to primitive and composite events constructed so far.

Primitive events are further classified into database, temporal and external events.

- *Database events* correspond to database operations, such as retrieve, insert, update and delete (in the relational model) and methods (in the object-oriented model).

- *Temporal events* are either absolute, specified with an absolute value of time or relative, where we have a reference point and an offset. The reference point may be an event, (including an absolute event) and the offset is a time string indicating the duration after the event specified occurs.

- *External events* are those events that are detected along with their parameters by application programs (i.e., outside the DBMS) and are only managed by the DBMS. Once registered with the system, they can be used as primitive events.

Composite events provide a powerful mechanism for expressing events. Many applications are not well served by primitive events alone. For example, an application may require that event E be expressed as the conjunction of events E1 and E2. A composite event is derived by applying event operators to primitive events. The operators are disjunction, conjunction, sequence, non-occurrence, aperiodic and periodic.

Sentinel uses Snoop as the event specification language. It can specify both local and global events. Two event detection mechanisms, namely, a local event detector and a global event detector, are implemented to monitor the behavior of local events as well as global events across applications.

3.2 Rules

The primary structure defining an ECA rule is, the condition which is evaluated when the rule is triggered, and the action which is executed if the condition is satisfied.

The condition evaluation and action execution can immediately follow the triggering event as in-line expansion of the triggering transaction. However this is not always desirable, as in the case where the rule has to enforce an integrity constraint. A transaction may have a series of operations, each possibly changing the state of a database . Suppose some data is modified during such an operation and an event is triggered. If this event fires an integrity rule immediately, the integrity check is done when the database is in an inconsistent state. Ideally in this case the rule need to be executed at the end of the transaction inspite of the event occurring in the middle of the transaction [2].

3.3 Coupling Modes

Coupling modes are introduced to specify the relative time lapse between an event detection and the testing of the condition associated with the event. The same can be done between the condition evaluation and the action execution. In our system the condition and action form a single transaction so the coupling mode applies only to the event detection with respect to the condition and action execution as one unit. There are three coupling modes:

- *Immediate*: When an event is detected, the transaction is suspended immediately and the condition associated with the event detected is executed. If the condition evaluates to true, the action part of the rule is executed else the triggering transaction continues. This feature has been implemented using threads. The triggering transaction executes as a thread and when an event is detected, the "Notify" creates threads for all the rules that this event subscribes to. The triggering transaction then waits for the completion of all the created threads.

- *Deferred*: When an event is detected, the rules that subscribe to this event are noted and threads are created for them but the threads are not scheduled or executed. The triggering transaction proceeds normally and before its commit, all of the deferred rules are executed. The triggering transaction has to wait for the deferred rules to complete before it commits. When a deferred rule is triggered from a triggering transaction (cycle-0), the rule is said to execute in cycle-1. All deferred rules created within the top-level transaction are part of cycle-1. All rules in cycle-1 execute at the end of the triggering transaction. Further, if any deferred rule is triggered within the action part of a cycle-1 rule, then that rule becomes part of cycle-2. In general all deferred rules created in cycle-n become part of cycle-n+1. At the end of the execution of all cycle-n rules, cycle-n+1 rules start executing and this process continues until there are no more new cycles created at any stage.

- *Decoupled*: Here the rule execution is done in a separate transaction from the triggering transaction. The two transactions can be either totally independent or be 'causally dependent' where the commit of the spawned transaction depends on the commit of the triggering transaction. This theses does not address the implementation of this coupling mode.

3.4 Rule Processing

The scheduler has to take the coupling mode into account when triggering rules. The coupling mode of the rule is specified during rule specification. When a rule is triggered by an event, the scheduler has to order the execution of rules based on their coupling mode.

Another important aspect of rule scheduling, is the use of rule priority. Each rule is given a priority with which it should execute. The priority is given at the time of rule specification.

The scheduler has to manage a possible set of complex rule hierarchies. This case arises when an event subscribes to more than one rule. Thus when the event occurs there are multiple rules that are spawned (each a thread of execution, in our design). Each of these rules have a coupling mode and individual priority value. The rules have to be executed based on their relative priorities and coupling modes and the action part of the rule in turn might trigger an event spawning more rules.

Assume there are three rules A with priority 8, B with priority 8 and C with priority 5, that are triggered when an event E occurs. The scheduler allows rules A and B to execute concurrently. Now if A's action in turn triggers rules A1 with priority 4 and A2 also with priority 4 and at the same time B's action triggers rule

B1 with priority 6. Then (if all the rules are in immediate coupling mode) we have A1, A2 and B1 all executing at the same time inspite of their varying priorities. This is because all their triggering rules were executing with the same priority. Thus rules are executed not just based on their individual priorities but also relative to their parents.

Rule management also 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 the deferred mode as the database state can be inconsistent 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.

3.5 Rule Execution Model

Here we describe a rule execution model as proposed in Widom and Finkelstein [5]. Here the rules are activated automatically as a result of database state transitions caused by externally generated operation blocks [5]. Operation blocks include a stream of data manipulations onto the database system grouped as one. This operation block always finishes execution and is indivisible. There can be a level of abstraction which supports concurrent processing, thus we can have multiple users who are transparent to one another. During execution of an operation block, data items may be updated, deleted, or inserted.

When a stream of operation blocks are submitted for execution, the execution begins in a state *S0* and continues as shown in figure 3.1

$$\begin{tikzcd} S_0 \arrow[r, "E_1", "T_1"'] & S_1 \arrow[r, "E_2", "T_2"'] & S_2 \arrow[r, "E_3", "T_3"'] & {} \end{tikzcd}$$

Figure 3.1. Rule execution model

$$\begin{tikzcd} S \arrow[r, "E", "T"'] & S' \end{tikzcd}$$

Figure 3.2. Single rule execution semantics

Here T1, T2 ... are transition labels; E1, E2 ... are effects of the transitions. The transition effect can be shown as a triple [U,D,I] denoting the three possible operations on the database: update, delete and insert. Each state in the execution sequence as shown above corresponds to a state in which a transaction begins execution. Thus there is a one-to-one correspondence between transition and transactions for externally generated operations.

3.5.1 A Single Rule

Let's take a single rule R defined as follows:

R: **when** *trans-pred*

where *predicate*

then *op-block*

This can be shown as in figure 3.2.

Now consider a transition T with effect E, we say that rule R is triggered by transition T if R's transition predicate, which acts like a trigger, is true. Triggering is only the first step. For the action part of the rule to execute, the condition must also hold. The condition of the rule may be with reference to the current state of the database or the logical transition tables.

If rule R's condition holds, the action part of rule R is executed. This action is given top priority and is executed first. This is the case even if there are other

$$\begin{tikzcd} S \arrow[r, "E", "T"'] & S' \arrow[r, "E_R", "T_R"'] & S'' \end{tikzcd}$$

Figure 3.3. Extended rule execution

externally-generated operation block, ready to execute. Execution of rule R's action in this case causes a new transition Tr as shown in figure 3.3

Though transition Tr is caused by a rule rather by an externally-generated operation block, Tr is just like any other transition. It is an operation block producing an effect and a new state. Thus the transition generated by a rule's action execution can trigger other rules or even the same rule again.

3.5.2 Multiple Rules

The semantics for multiple rule execution is similar to that for a single rule, the interaction of externally generated operation blocks and rules is as follows:

- Execute the operation block thus creating a transition
- Execute rules (creating transitions) until there are no more to execute
- Go to top.

Let T1 be a transition in step-1 with effect E1. Let there be rules R1, R2, R3, ..., Rn be generated as a result. These form part of set P, "pending rules". Out of the pending set a rule is selected based on some selection criteria. If rule Ri is chosen then it is removed from the set P and is executed. If Ri's condition holds then the action part is executed else another rule is taken out of P (again based on the same selection criteria) and the condition part is tested, if satisfied, the action part is executed.

Now the set P is updated, by removing Ri from it and adding any rules that were produced as a result of Ri's execution. The next step is to select a rule Rj from the set

P and continue the same operation. One question that needs to be answered is what is the state of the database. Is it the initial state before Ri started executing or the state after Ri's execution? This could make a difference because the condition part of Rj may satisfy the one state and not the other. It is better to have the database state to be that after the execution of Ri since that would be the most recent state and there is no point using an outdated state.

3.6 Rule Specification in Sentinel

Rules are instances of a system defined Rule class. The Rule class is derived from the system defined Notifiable class, thereby enabling rule objects to be notified of the primitive events generated by reactive objects.

Rules can be classified into class level and instance level rules depending on their applicability. Class level rules are applicable to all instances of a class whereas instance level rules are applicable to particular instances. Since class level rules model the behavior of a particular class, they are declared within the class definition itself. On the other hand, instance level rules are declared in the application code. Rules, regardless of where they are declared, are translated to notifiable rule objects.

There are mainly two differences between class level and instance level rules. First, class level rules are applicable to all instances of a class, throughout program execution (when enabled). Instance level rules, however, are applied to a varying subset of instances. Secondly and more importantly, a class level rule can only be applied to one type of object (e.g. to only person objects). Instance level rules are more powerful since they can be potentially applied to different types of objects. Instance level rules can thus monitor situations spanning different classes. This is accomplished by the rule subscribing to the different types of objects to be monitored.

3.7 Sentinel Architecture

The Sentinel architecture proposed in this section extends the *passive* Open OODB system [6]. 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 transforming the user class definitions as well as the application code. Extensions incorporated for making the Open OODB active, are:

- Specification of ECA rules either as a part of the class definition or as part of an application; this is pre-processed (by using an enhanced C++ pre-processor) into appropriate code for event detection and rule execution,
- Detection of primitive events by using the sentry mechanism of the Open OODB. Sentry mechanism provides a wrapper method that permits us to invoke notification of an event to the composite event detector,
- A composite event detector for detecting composite events in various contexts [7]. There is a composite event detector for each Open OODB application or client (each application of Open OODB is a client to the Exodus server),

Figure 3.4 shows how the class lattice of the Open OODB has been extended. The classes outside the dotted box have been introduced to make Open OODB active.

In order to satisfy the above requirements in an object-oriented framework, we use the architecture shown in Figure 3.5. The architecture supports the following features: i) detection of primitive events, ii) detection of composite events, iii) parameter computation of composite events, and iv) clean separation of composite event detection with application execution.

The primitive event detection is based on the design proposed by Anwar et al.[8]. Both primitive and composite events can be signaled as soon as they are detected.

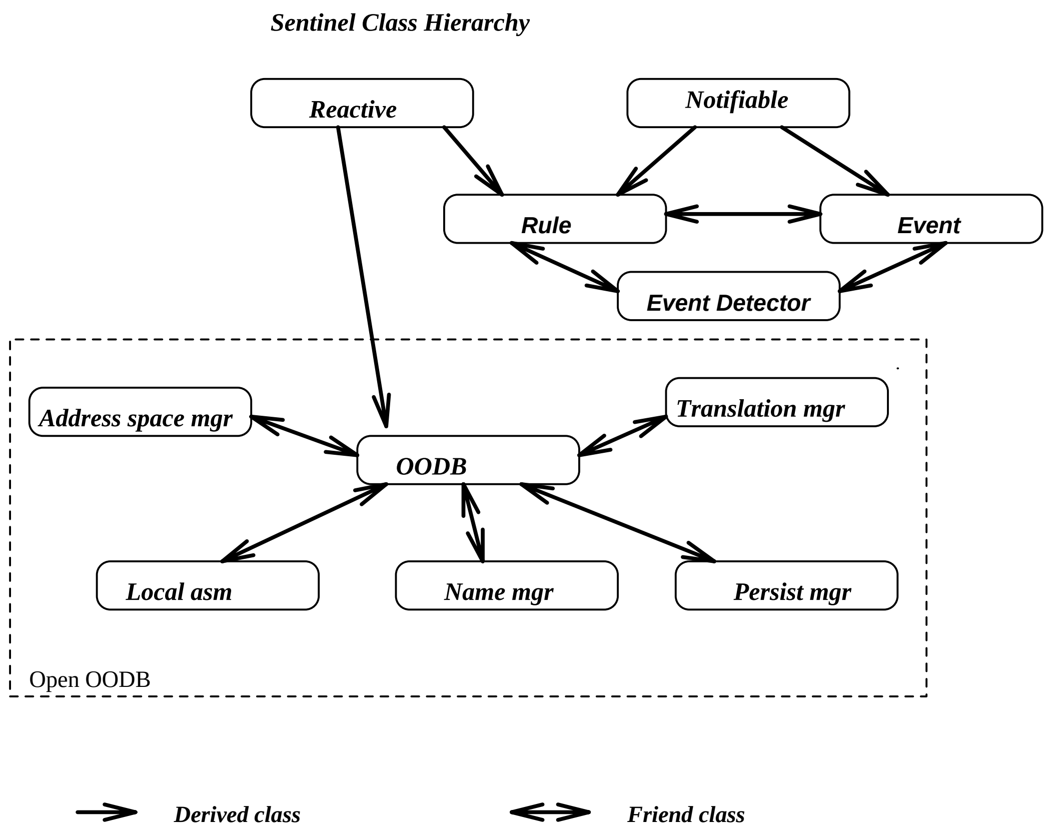

Figure 3.4. Class Lattice of Sentinel

However, the detection of a composite event may span a time interval as it involves the detection and grouping of its constituent events in accordance with the parameter context specified. We have modified the Open OODB to support the detection of primitive events. A clean separation of the detection of primitive events (as an integral part of the database) from that of composite events allows one to i) implement a composite event detector as a separate module and ii) introduce additional event operators without having to modify the detection of primitive events.

Each application has a local event detector to which all primitive events are signaled. In addition each application will have a thread that handles the execution of rules whose events span applications (a global event-handler thread).

When a primitive event occurs it is sent to the local event detector and the application waits for the signaling of rules that are detected in the immediate mode.

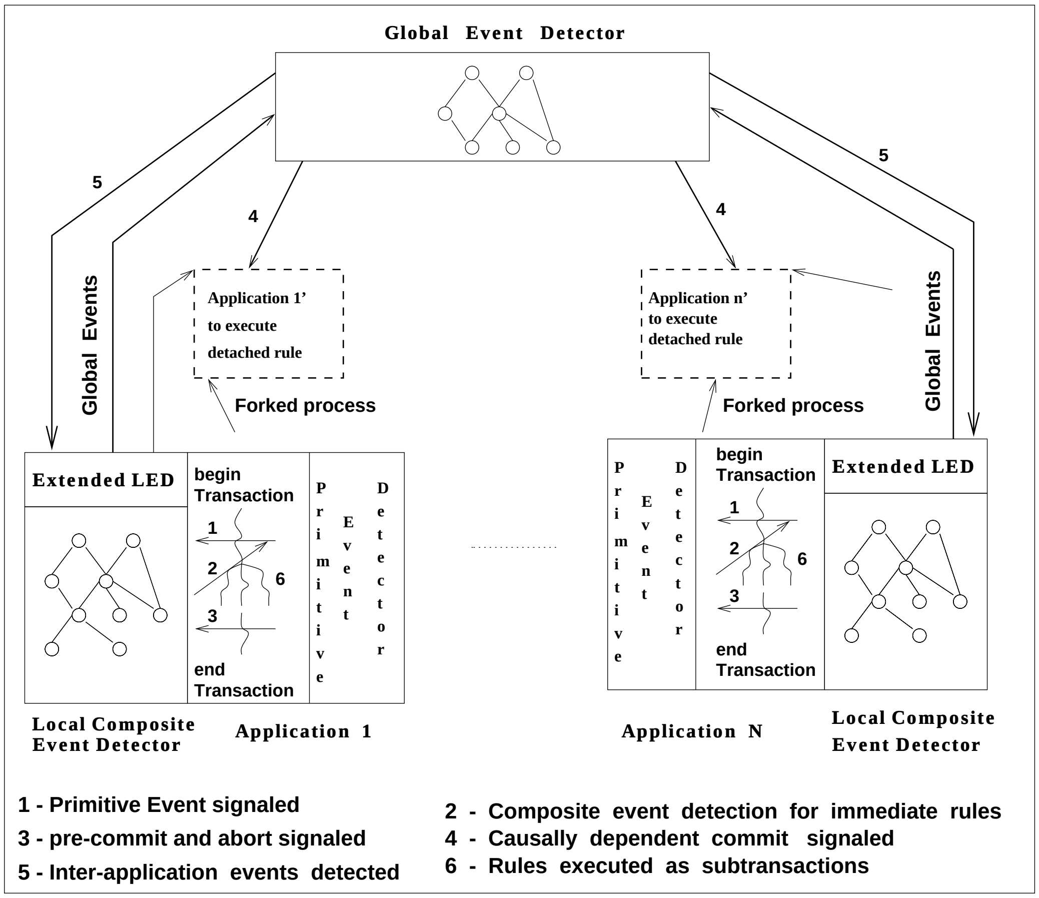
Global Event Detector
5
4
4
5
Global Events
Global Events
Application 1' to execute detached rule
Application n' to execute detached rule
Forked process
Forked process
Extended LED
begin Transaction
1
2
6
3
end Transaction
Primitive Event Detector
Extended LED
Local Composite Event Detector
Application 1
Application N
Local Composite Event Detector
1 - Primitive Event signaled
3 - pre-commit and abort signaled
5 - Inter-application events detected
2 - Composite event detection for immediate rules
4 - Causally dependent commit signaled
6 - Rules executed as subtransactions

Figure 3.5. Sentinel Architecture

The global event detector communicates with the local event detectors for receiving events detected locally and with the application's global event handler for signaling the detection of global events for executing tasks based on global events. Again there is a clean separation between the events detected by the local event detector and the global event detector. Finally, as the local event detector and the application share the same address space and our event detection uses an event graph similar to operator trees, it is possible to combine rule evaluation with event detection (when the coupling mode permits and rules are non-procedural) and optimize the entire tree as a whole.

A notifiable rule object subscribes to a set of reactive objects. All the primitive events generated by those reactive objects are propagated to the rule object via the notification mechanism. The notifiable rule object records only those primitive events of interest and discards the rest.

The Notifiable Class

The primary objective for defining the notifiable class is allowing objects to receive and record primitive events generated by reactive objects. The rule class is a subclass of the notifiable class, thus rule objects receive and record primitive events generated by reactive objects.

The Reactive Class

In order for a class to provide reactive capabilities it requires a facility for specifying which of its methods generate primitive events, a mechanism for propagating generated primitive events along with their parameters to notifiable objects, and a method for notifiable objects to request the acquisition of information regarding generated primitive events.

The requesting mechanism is termed as the subscription mechanism and the propagation of generated primitive events is termed as the notification mechanism.

Due to the fact that potentially many classes may require reactive capabilities, i.e. the subscription and notification mechanism, a class was defined whose sole objective is the provision of these reactive capabilities. This class is named the reactive class.

CHAPTER 4
NESTED TRANSACTIONS

A transaction is the basic unit of atomic, consistent and reliable computation in a database system. Although rule execution can be treated as part of the transaction, this approach: i) Sequentializes rule execution (hence there can be no concurrent rule execution) and ii) increases the duration of the top-level transaction, this may hold system resources for a long time. The problem with a long transaction is that it holds system resources for that duration and if aborted, the whole transaction has to be redone. All these factors indicate that we need a mechanism that avoids the above pitfalls. This brings nested transactions into the picture as proposed by Eliot and Moss [9] and shown to be useful in the context of rule processing by Chakravarthy et al. [2].

4.1 Motivation

The motivation for using Nested transactions can be summarized as follows:

- We can achieve decomposition and finer grained control of concurrency and recovery, the very reason why traditional transactions are unsuitable.
- Intra-transaction parallelism allows execution of a long transaction into concurrently running smaller parts thus bringing about an increase in overall efficiency and decreasing response time.
- Intra-transaction recovery allows sub-transactions to fail independently of each other and independently of the parent transaction. Thus uncommitted sub-transactions can be aborted and rolled back without any side effects to other

sub-transactions. This brings down the recovery expense mainly in terms of time.

- System modularity allows the modules of a transaction program to be designed and implemented independently and they facilitate a simple and safe composition of the transaction program. It also achieves encapsulation (information hiding), failure limitation and security

- Distribution of implementation allows the use of distributed algorithms that achieves a flexible control structure for concurrent execution. Distribution can be in terms of data or in terms of processing both bringing a positive effect on efficiency. It allows cost-effective use of hardware (processors, I/O devices) and improves responsiveness. Distribution of data i.e. replication of data makes data more available.

4.2 Role of Nested Transactions in Rule Processing

In the ECA paradigm, an event occurs which is followed by the testing of a condition. Based on the outcome of the condition evaluation, the action associated with that condition is executed. This whole process can be done in the top-level transaction but that would encounter some of the problems explained above about long running transactions. It is thus preferable to break up that long transaction into sub-transactions. In our model we have the condition and action execution in a separate thread. The whole execution is part of a sub-transaction. Concurrent execution of rules will be difficult unless there is a mechanism that provides correctness semantics for their concurrent execution.

Hence we resort to nested transactions is to handle the situation where one event triggers more than one rule. This implementation schedules rules such that if one event triggers more than one rule and they have the same priority value, then the scheduler allows concurrent execution of these rules. This may bring about conflicts with respect to data access. To deal with data conflicts, we let each rule thread to execute as a sub-transaction and the nested transaction synchronization scheme is used to handle the conflicts. Moreover there can be multiple levels of nesting where the action part of a rule could in turn trigger more rule executions and here again the nested transaction model helps in preserving data consistency.

4.3 The Nested Transaction Model

The most important issues that needs to be addressed in our design is the concurrency control issues with respect to nested transactions. For this, we need to understand the overall picture of the nested transaction model before we proceed. This description follows the definition of [9].

A transaction in this schema can have any number of sub-transactions and each sub-transaction in turn can have its own sub-transactions. This gives the transaction model, a tree like structure. The root transaction is called the top-level transaction and all others are called sub-transactions. Transactions having sub-transactions are called parents and the sub-transactions are called children. Also there is a concept of superiors and inferiors or ancestors and descendants. The set of all descendants of a transaction forms the sphere of that transaction.

As shown in the figure 4.1, A is the top-level transaction, B and I are its children and all other transaction are its descendants. C is the parent of D, F and G. D, E, F and G are inferiors of C. Also D, C, B and A are superiors of E. The sphere shows the sphere of control of C with all its descendants.

This nested hierarchy can also be seen as a collection of nested spheres of control with the top-level transaction as part of the outermost sphere and acting as an interface to the outside world.

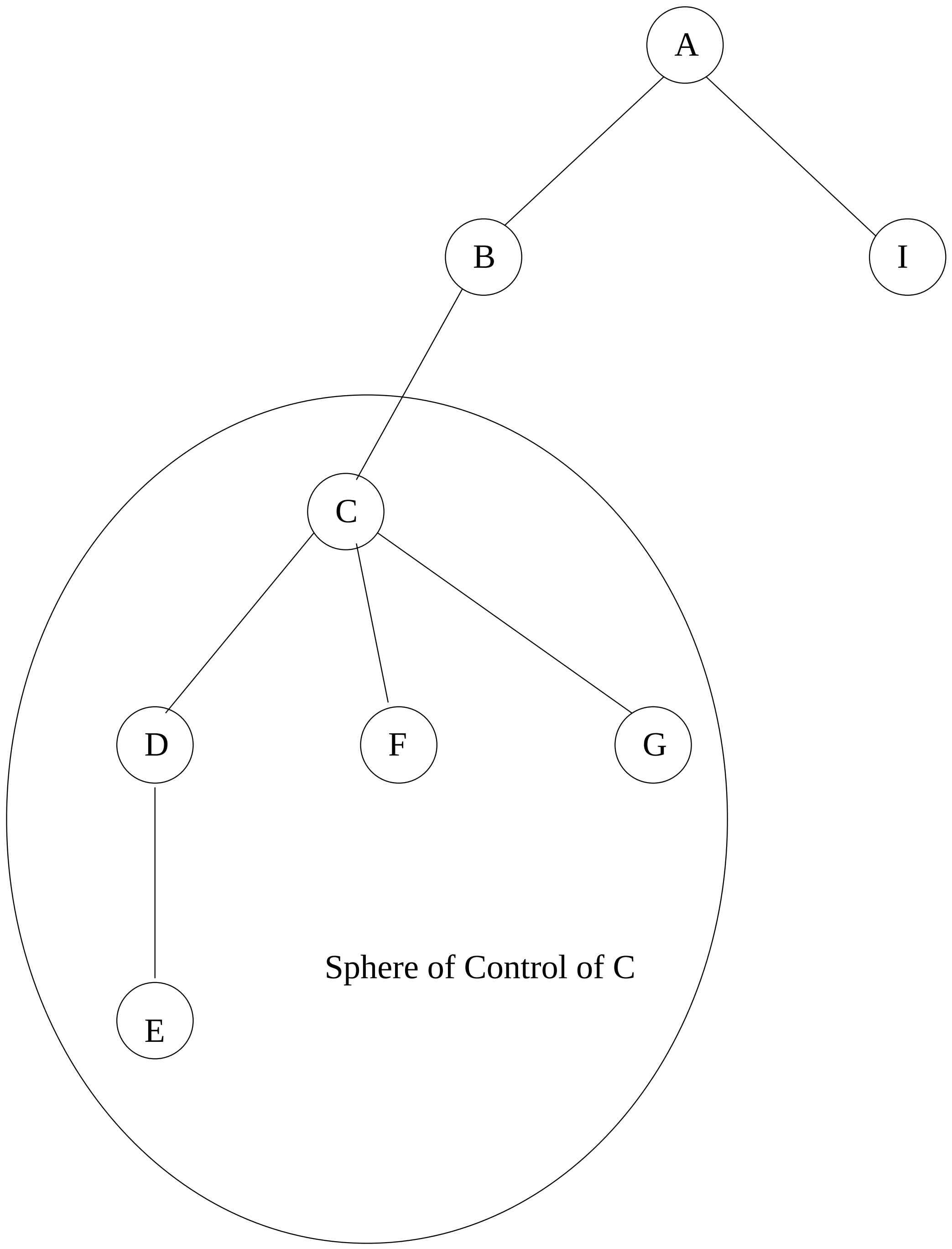

Figure 4.1. Nested Transaction Model

The ACID properties hold for the top-level transaction but not necessarily for the sub-transactions. Sub-transactions terminate by either aborting or committing. If it aborts it does not pose a problem to the rest of the hierarchy, that is, other sibling sub-transactions need not be aborted. Instead that sub-transaction and all its descendants try to recover. But the commit of all sub-transactions are dependent on the commit of all superiors, all the way up to the top-level transaction. Thus the sub-transaction is atomic and isolated. It need not be consistent, especially in the case where the parent transaction needs results of several child transactions to perform some consistency preserving actions.

As mentioned in the advantages of using nested-transactions, intra-transaction parallelism is among the most important. There are four levels of intra-transaction parallelism that we can achieve.

- Neither parent/child nor sibling parallelism: This is actually a misnomer because this state has no intra-transaction parallelism at all. Only one transaction can be active in a sphere at a time. Thus there is no worry about concurrency here. Only systems that have synchronization between processes provide this kind of parallelism. This means that two top-level transactions can be running at the same time on two separate processes s in the same system.

- Sibling parallelism: Here the parallelism is supported among siblings. Thus the parent is suspended and all its siblings can simultaneously execute. Thus transactions can share objects with the parent without need for any concurrency control mechanism, but there has to be some concurrency control scheme for objects shared among siblings.

- Only parent/child concurrency: In this kind of parallelism each parent can concurrently execute with one of its children. Thus only transactions along a single path in the hierarchy execute concurrently. This simplifies matters as far as concurrency control, as only transactions along the same path need be synchronized.

- Parent/child as well as sibling concurrency: This kind of parallelism allows any combination of transactions and sub-transactions to execute in parallel. This is definitely supports maximum parallelism but at the same time the most difficult to implement. The amount of overhead to achieve concurrency control is enormous and usually not worth the effort.

4.4 Concurrency Control in Nested Transactions

The synchronization details that we want to achieve are the following

- A notion of serializability should exist among top-level transactions i.e. one transaction should be able to perform updates without interference from any other transaction trying to perform an update or retrieve. This means that a strict two-phase locking protocol needs to be followed for synchronizing among transactions.

- Within a nested transaction we should be able to achieve as much parallelism as possible but making sure that there is concurrency control to maintain acceptable data consistency demands.

The locking rules in our nested transaction implementation follow closely with the one proposed in [9]. Any transaction can acquire a lock on an object in a specific mode (Read/Shared or Write/eXclusive) and it holds the lock in the same mode until its termination (commit or abort) or until it explicitly upgrades the lock.

Also a transaction can retain a lock besides holding a lock. Parent transactions retain a lock held by a sub-transaction when it commits by just inheriting the lock.

A retained lock is like a place holder, the transaction that retains a lock actually has no access to the object it locks, all it does is to ensure correctness with respect to acquiring locks.

A retained X lock for instance ensures that transactions outside the sphere of control of the retainer cannot acquire the lock but its descendants can, based on some locking rules discussed below.

4.5 Basic Locking Rules

- i) Transaction T can acquire a lock in X-mode if
  - no other transaction holds the lock in any mode
  - all transactions retaining a lock in X or S mode are ancestors of T
- ii) Transaction T can acquire a lock in S-mode if
  - no other transaction holds the lock in X-mode
  - all transactions that retain a lock in X-mode are ancestors of T.
- iii) When a sub-transaction commits, the lock it holds or retains are inherited by its parent. The parent retains a lock in the same mode that T held it. If the parent P already retains a lock when the sub-transaction commits then the new retain mode of the parent P is given by

  new mode of P = MAX (old mode of P, mode of T)

  where $S < X$ i.e. a shared (Read) lock is considered smaller than an eXclusive(Write) lock.
- iv) When a transaction aborts, it releases all locks it held or retained. If any of its superiors held or retained a lock, they continue to do so.

Thus the rules stated above only allow upward inheritance of locks at commit time. A parent can therefore inherit the locks held by its children only when they commit, an abort does not affect the inheritance of locks.

4.6 How Nested Transaction Scheme Fits into Our System

As mentioned before this implementation allows parallel rule execution based on priorities and coupling modes of the rules involved. Thus in an application there can be a number of active threads, each a rule executing on data in the database. This is the classic scenario where nested transactions come into the picture. Each rule thread should be executed as a nested transaction to take care of the concurrency issues that have been discussed.

The nested transaction model has been developed by Bhadani[10]. This uses an anchored hash table that keeps information about every user level object that a transaction acquires. This is then referred to whenever a transaction needs to acquire an object and based on the locking rules described above, a decision is made as to whether the requesting transaction can be given the lock or not.

This code was written for the Sun OS environment which uses the zeitgeist system. This code required modifications to port it to the Solaris environment and the OpenOODB system that we currently use. The porting mainly involved rewriting the thread routines with respect to the Solaris thread package that has quite a few changes over the Sun thread package.

4.7 Our Nested transaction model

Our nested transaction model supports only sibling parallelism. This offers the most parallelism second only to parent/child as well as sibling concurrency. The most convenient mechanism to achieve concurrency control, is locking and our implementation is also based on locking. We have two modes of synchronization:

- Read: where multiple transactions can Share the object and none of them try to modify the object.

- Write: only one transaction can have eXclusive control over the object and is not shared with any other transaction.

4.7.1 Original Model

The original implementation was tied to the Zeitgeist system. This has been ported to the OpenOODB system. The transaction calls are the same in both systems. They are begin, commit and abort transaction.

4.7.2 Transaction Calls

The following are the operations that take place when transaction calls are made to OpenOODB:

- Begin transaction: This starts an OpenOODB transaction and any "fetches" of objects in the database within a transaction has to notify the lock manager and the transaction is given the object based on the locking rules discussed in section-4.4. Update of an object is performed in process memory. If the transaction terminates normally, usually with an explicit *commit transaction*, then the modified objects are written back to the persistent store. If there is an *abort transaction* none of the modified objects are written back to the persistent store. Each access to an object needs to consult with the lock manager to decide whether it can get access to that object based on the locking rules.

- Commit transaction: The objects held in process memory that have been modified is written back to the persistent store. Locks held by the transaction on that object are released and a check is done for any other transaction that is waiting to hold a lock on that object. The lock is given again based on the locking rules.

- Abort transaction: None of the objects held by that transaction are written back to the persistent store and all locks held by the transaction are released.

4.7.3 Lock Manager

The lock manager is a very important component of the nested transaction model. It is the lock manager that maintains concurrency control. The main data structure in the lock manager is an anchored hash table. The data structure used by the lock manager has to serve two different purposes, i) given an object which can be uniquely identified, it should be possible to find out all the transactions that hold a lock on that object, ii) for a transaction that can be uniquely identified, it should be possible to find out all the objects on which it holds a lock. To get these two kinds of information we can have two different data structures one for each purpose. Instead using the anchored hash table, we can achieve both needs using a single data structure.

The hashing is done on the object (the object-id and the storage group number) and each object has an anchor node. If more than one object hashes to the same bucket then a linked list at that bucket is maintained. Also for each access of that object by a transaction, a separate node is created which is anchored on the object node. Thus a separate linked list from each anchor (object) is formed for all transactions that hold a lock on that object.

A list of all transactions is maintained which links nodes in the anchored hash table with the same transaction-id are connected as a list. This is helpful in tracking the transaction and all the objects held by it.

The anchored hash table is shown in Figure 4.2.

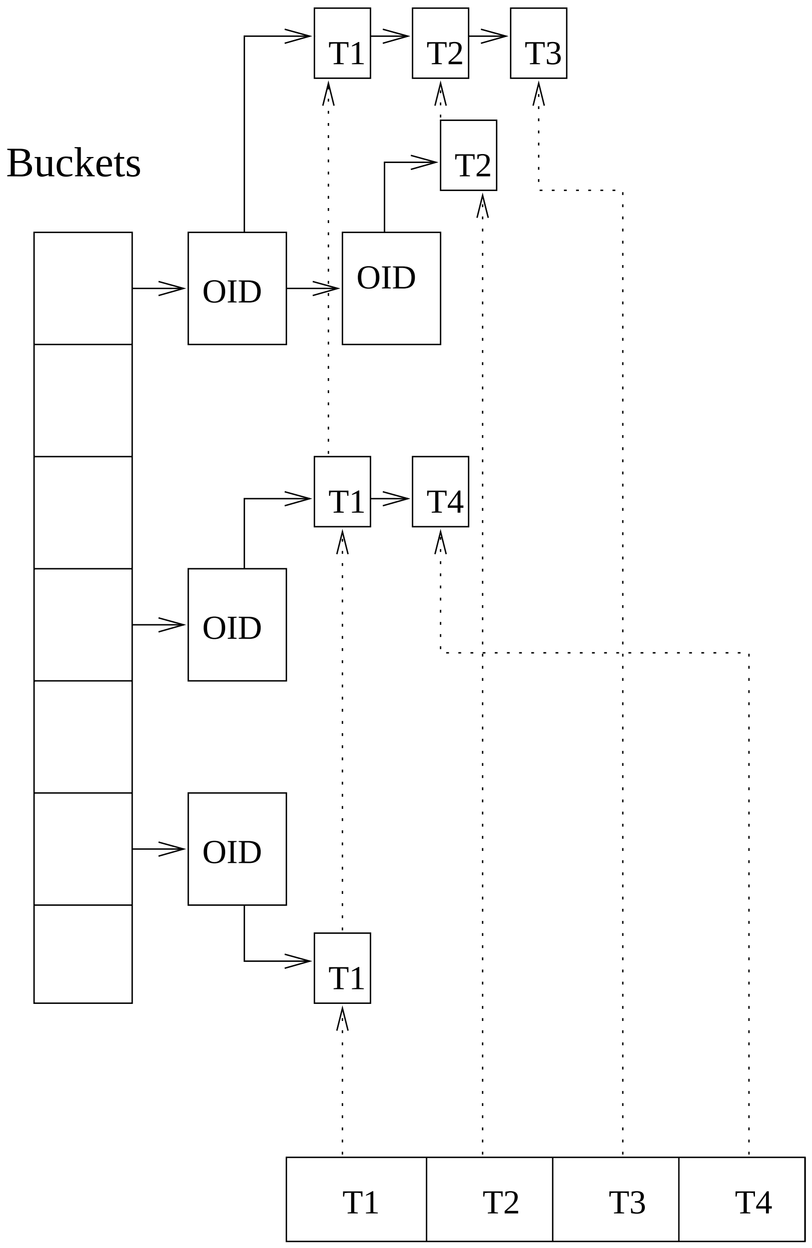

Figure 4.2. Anchored hash table

4.7.4 Extensions of the Model

The nested transaction model as mentioned before has a tree like structure with the parent transaction as the root and the sub-transactions as its children. To determine whether a transaction (or sub-transaction) can be given a lock to an object the lock manager has to check all ancestors of the transaction and come to a decision based on the holdmodes.

To make the task of looking for parent nodes in the hash table easier we introduced a naming scheme for transactions and sub-transactions. All top level transactions are named in numerical order. A sub-transaction is named using the following formula:

child tid = (parent tid << 4) + child counter

child counter = child counter + 1

"<<" indicates binary shift, to the left. This makes it easy to find immediate children of a sub-transaction, simply by using the ">>" operator.

The semaphores and threads used in the previous model were that of Sun OS. The whole system has been ported to Solaris, the thread and semaphores had to be converted to Solaris that have different capabilities, more of which is described in chapter-5.

The enhanced thread capabilities of the operating system has also been used to improve the granting of locks. When a sub-transaction requests a lock from the lock manager, it either receives the lock based on the locking rules discussed before or does not. In the latter case we change the mode of that sub-transaction to the 'wait' mode and put it at the end of the transaction list. We thus achieve a first-come-first-serve ordering for the blocking sub-transactions. The sub-transaction is now in a dormant state and execution can proceed only on acquiring the lock, this is achieved by blocking on a unique semaphore. When the transaction (or sub-transaction) that was holding the lock commits or aborts, it frees the lock. The lock manager then scans the list of sub-transactions waiting for a lock on that object and un-blocks the first semaphore on which that sub-transaction is blocking. If there are more sub-transactions waiting for a lock on that object they are added further down the transaction list and as mentioned above are un-blocked on a first-come-first-serve basis.

There were a number of fixes that were made to the system to support the nested transaction model. The lock manager was re-visited and parts of the code re-written to improve functionality. Also fixed the transaction and sub-transaction constructors and the commit for transactions. OpenOODB's fetch methods had to be changed, to add a call to the lock-manager which inserts the object into the hash table along with other transaction information.

All dependencies to the earlier Zeitgeist system have been removed, including transaction calls, and have been changed to OpenOODB calls. The transaction calls have already been discussed above.

CHAPTER 5
RULE SCHEDULER : DESIGN AND IMPLEMENTATION

5.1 Why Do We Need Rule Scheduling ?

As mentioned in chapter 3, when an event occurs in the database system, the rules that subscribe to that event are triggered. A simplistic approach at this stage would be to let all the rules that have been triggered by this event to run one after another (i.e., serially using a conflict resolution mechanism). This approach does not usr parallelism or maximize throughput. It would be beneficial to let the rules execute in parallel, thereby reducing the overall execution time for that transaction.

Each rule has a priority associated with it. If a set of rules is executed by an event the rules have to be triggered based on their relative priorities. Thus the rule scheduler has to fire rules based on the priority of that rule with respect to its sibling rules. The scheduler has to allow rules with the highest priority to execute first and after their completion, the scheduler has to allow the next rule (or set of rules) with lower priority to execute.

The role of coupling modes has already been described in chapter 3. Our system supports both Immediate and deferred coupling modes. The coupling mode is specified at the time of rule creation. Rules have to be scheduled based on their coupling modes. If a rule has been triggered from within a transaction and the rule has an immediate coupling mode, then the transaction from which the rule was spawned has to wait and the rule should be given processor time for execution. Only at the end of the rule execution can the triggering transaction continue.

If the rule has a deferred coupling mode, then the triggering transaction can continue and the triggered rule has to wait for the completion of the main transaction for it to be scheduled. The important issue here is that the deferred rule has to run at the end of the triggering transaction but the triggering transaction cannot commit until all the deferred rule are executed. This needs careful scheduling of rules.

5.2 Design Issues

Thus the major factors to be taken into consideration while designing the scheduler is the priority and coupling mode issues. What needs to be taken into account is the possibility of an event triggering many rules, at the same time, with different priorities and coupling modes at the same time. The scheduler would first have to segregate the rules based on their coupling modes because immediate and deferred rules have to be treated differently. Thus, even if a deferred rule has a higher priority than an immediate rule, the immediate rule has to be spawned first.

After the segregation, the scheduler has to suspend the main transaction. The main/triggering transaction and the triggered rules (sub-transactions) are all different threads of operation. The scheduler can thus use thread related functions (discussed later) to suspend and continue rule processing. The triggering of a set of immediate rules is achieved by thread operations. The triggering of rules have to be done based on priority. Thus the scheduler has to keep track of all the rules that have been triggered and spawn the rules one after the other based on their priority.

The deferred rule semantics are quite different from those of the immediate rules. The scheduler has to keep track of all the deferred rules spawned by that transaction as the actual execution of the rule can take place only after the completion of the spawning transaction, just before it commits.

At the end of the transaction, all the deferred rules that were created from within the transaction can start executing in order of their priorities. This again is done by the scheduler using thread functionalities. The execution of these deferred rules in turn could trigger more rules. If the rule triggered has an immediate coupling mode, then the deferred rule that triggered it has to wait for its completion. In case of a deferred rule, we follow the *cycles of execution* policy, where all deferred rules spawned by the top-level transactions and their sub-transactions have to finish execution before any of the second cycle of deferred rules can start. The deferred rules spawned by this cycle are processed in the next cycle. Thus at any time there can be only two cycles of deferred rules.

From the above we can see that, one of the most important functions of the scheduler is to keep track of the stage of execution of each rule. Based on this knowledge alone, the scheduler decides when to allow each rule to execute. A set of operating modes associated with each rule is maintained to keep track of the state a rule is in. Four operating modes are possible in our scenario: *ready* - when the rule is triggered and is ready to execute, *wait* - when the rule spawns another rule in the immediate coupling mode and is waiting for that rule to finish execution, *executing*- when the rule is currently executing, and *finished* - when the rule has finished processing. The *finished* operating mode is used by the scheduler to make sure that all the rules in the immediate coupling mode have finished executing. The scheduler can then allow the triggering transaction to continue. The *finished* mode also helps the scheduler to know if all the rules in a deferred cycle have finished executing. Similarly a *wait* indicates that the scheduler has more immediate rules generated by that rule to execute.

By the time a transaction completes, the scheduler has no more rules to schedule. It is woken up when another transaction is executed (maybe within the same application) which triggers another set of rules.

5.3 Threads

The scheduler has to provide a variety of functions that help in achieving our model of parallel and prioritized rule execution and to incorporate the handling of different coupling modes. The tool that we use to achieve these goals are threads.

Having decided to choose threads to execute rules concurrently, the question was which multi-threading system do we use. The choice was between POSIX threads or pthreads and Solaris threads. After a detailed study of the two, it was clear that at a higher level there were no real fundamental differences between the two. There were no incompatabilities, i.e. what could be expressed in one system could also be expressed in the other though it may involve a different approach. What stood out though were key API related issues that were different from one another. The key differences between the two API's can be summarized as follows:

- Features in Solaris threads API but not in pthreads API
  - Suspending and continuing threads: This feature allows any thread in an address space to suspend or continue any other thread using a thread-id which is unique to every thread
  - Reader/writer locks: This feature helps when applications need to synchronize data access using the reader/writer approach where threads can hold the lock in either mode. This is best suited if there are more readers than writers in the system.
  - Setting concurrency: The user can specify the number of threads in the system at any given time.
- Features in pthreads API but not in Solaris threads API
  - Attrribute objects: It is possible to specify an attribute object that can be shared among a group of threads or synchronization variables. This makes

programs more portable and allows a group of pre-defined attributes that are used by every thread in the system.

- Cancellation semantics: Using an API call, a related set of threads can be cancelled or stopped and the state of the system can be restored to the original state. This is particularly useful if more than one thread is trying to complete one task. At the end of the task, a cancel can be called stopping execution of all threads.
- Scheduling policies: pthreads supports FIFO as well as round-robin scheduling policies

The most important feature that was needed by the scheduler was the ability to start and stop threads (rules) based on their priorities and coupling modes. This feature could be easily implemeted using the Solaris thread API but doing it using pthreads would be more complicated. The main advantages of pthreads, the attribute objects was really not needed by the scheduler as the thread attributes (like thread priority) were not largely used. The same reason can be applied for the cancellation semantics too. The scheduling policies provided by pthreads are of no use, as we are doing all the scheduling.

Solaris threads support the concept of multi-threading. Multi-threading has a variety of benefits including performance gains from multiprocessing hardware (parallelism), increased application throughput, enhanced process-to-process communication, mainly because it removes the difficulties of IPC and can use shared data structures instead [11].

On the other hand, threads can also produce many problems that are difficult to handle. The most important of which is the concurrency issue. As mentioned earlier, the communication between threads in the same address space can be achieved with the help of shared data structures but this can lead to inconsistent values as two threads might be simultaneously accessing the same data and corrupt it in the process.

Threads provide a couple of ways of circumventing this problem. These are called synchronization objects and include mutex locks, condition variables, reader/writer locks, and semaphores.

- Mutex locks: The most basic synchronization mechanism, mutual exclusion locks, ensures that only one thread can either execute a crtical section of code or access shared data at one time.

- Condition variables: A thread can block until a condition is statisfied. The condition testing must be done under the protection of a mutex lock. When the condition is false the thread blocks. Another thread can change the condition and can signal the associated condition variable causing all threads wating on the condition variable to wake up.

- Reader/writer locks: This approach is best suited where data can either be locked for update or just for reading. These locks are slower than mutexes but can improve performance if they protect data that are not frequently written but can be read concurrently by many threads. There is no real acquisition order when there are many threads waiting for a read/write lock. But to avoid starvation, Solaris tends to favor writers over readers.

- Semaphores: Uses the classic P and V operations to allow synchronization. This can be used just like the condition variables but is not as well structured.

This implementation uses mutex locks as well as condition variables to support synchronization among the threads. Mutex locks have been used because at any given time there may be multiple threads executing in the system, each trying to access the shared data structure. The data structure is accessed either to add or delete information. Our goal is to make sure that only one thread can access the data structure at one time, in other words making the access mutually exclusive. The scheduler also accesses the main data structure without updating it. It may seem as though the reader/writer locks are more suitable but the 'read' operation does not really need to see the latest view of the data structure. This is so because the scheduler iterates over the data structure in an infinite loop and if it misses an update, it would see it on the next iteration. Using reader/writer locks would hurt performance if the data-structure has to be locked every time the scheduler loops through it. Condition variables are used to block transaction's from committing until all the rules that it spawned finish execution.

5.4 Thread Functions Used in This Implementation

thr-create: creates a new thread, parameters include: the start routine and flags, which are

- Detached - The thread-id and other resources can be used as soon as the thread terminates.
- Suspended - Suspends the thread as soon as it gets created, has to be explicitly continued
- Bound - Permanently binds the new thread to an LWP (light weight process)

thr-self: returns the thread id of the calling thread.

thr-suspend: blocks the execution of the thread specified by its id.

thr-continue: unblocks a blocked thread

thr-kill: sends a signal to a thread

thr-join: wait for the termination of a thread(s).

thr-yield: yield execution to a thread of same or higher priority in the system

thr-exit: terminates execution of the thread *mutex-lock*: locks a mutex variable initially created

mutex-trylock: lock with a non-blocking mutex

mutex-unlock: unlock the locked mutex

5.5 Scheduler Design

The design of the scheduler is based on the capabilities provided by the Solaris thread package, as described above. Each rule triggered during the execution of an application has to be a separate thread. The execution of the thread is controlled by the scheduler depending on its priority (its own priority and its priority relative to its parent), and coupling mode.

Every event found in the application is wrapped with a 'Notify' by the preprocessor. 'Notify' identifies all the rules that correspond to the event and creates a thread for each rule. The thread is created using the thr_create function:

thr_create(NULL,0, &do_action, param, THR_SUSPENDED, &tid[Count])

The first two arguments correspond to the stack for the thread. The stack is not used by the threads in this implementation. The function that is executed as the body of every rule-thread is called 'do-action'. 'param' is the argument list sent to 'do-action', which includes the 'RULE' object and the paramter list obtained from 'Notify'. 'THR_SUSPENDED' indicates that the thread is created in a suspended state, the scheduler wakes it up at the approppriate time. Finally tid is an array of thread-id's and tid[Count] generates an id based on the value of Count, which is a global variable that is incremented for every thread spawned in that address space.

The data structures used within Notify, including the event graph, are global and can be accessed by all the threads in the system. To make sure these data structures are consistent, we lock them using mutex locks whenever a thread updates or traverses them.

The 'Notify' then calls insert_rule as follows,

insert_rule(tid[Count],this->get_priority(),thr_self(),mode);

This call inserts a new node in the data-structure (discussed in the following section) corresponding to the rule being triggered. The first argument is the thread-id of the new thread generated. The second argument is the priority which was specified during RULE construction. 'thr_self' gets the thread-id of the spawning process and mode specifies whether the rule has an immediate or deferred coupling mode. The function 'do-action', checks the validity of the condition for that rule and if true it performs the action function specified by the rule. This whole thread of execution is wrapped within a sub-transaction. The reason being that there is a possibility of a number of rule threads executing at the same time. And running each thread as a sub-transaction ensures the correctness of concurrently executing rules. The nested transaction semantics is enforced by the nested transaction manager which is independent of thread execution.

5.6 Main Data Structure Used

There are a number of parameters that have to be known when forming a thread to execute a rule, namely, its parent rule id, priority and coupling mode. To store all this information we use a data structure called the *process-rule-list*. Each node in the list corresponds to one rule that has been triggered. When creating an instance of the *process-rule-list* node for a new rule, it has to be supplied with the parent thread id, coupling mode and priority.

The id's of the rules are the thread id's that are obtained when creating a thread for that rule. The system assigns a unique id (of type thread-t) to every thread it creates. Rules triggered in the top-level transaction have the parent id as 1. Rules triggered from within a rule execution get the parent id by doing a *thr-self()* from the executing thread.

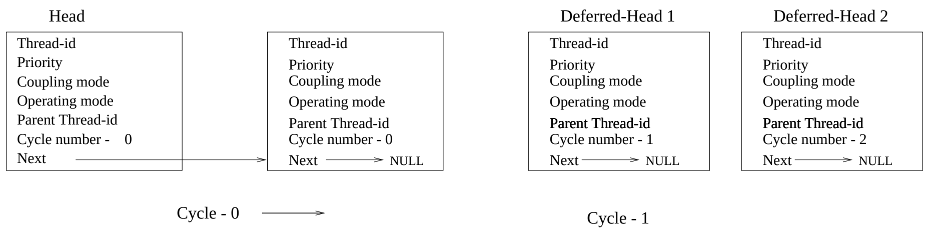

Figure 5.1. process-rule-list

The priority and coupling mode can be obtained from the rule object as they are provided during creation. The default priority is the lowest possible priority (which is 1) and the default coupling mode is immediate.

Also associated with each rule is an operating mode which captures the state of rule execution. There are four possible states : Ready, Wait, Executing and Finished.

Deferred rules have an extra data member, the cycle number. All top level deferred rules and those triggered by rules executing with Immediate coupling modes are assigned cycle-1. All rules triggered from a deferred rule are assigned the next cycle number.

5.7 Overall Picture

In an application the user specifies the rules associated with their respective event methods. These rules have a priority and a coupling mode specified by the user. The preprocessor inserts a Notify call inside the wrapper to detect these events at run time. The notify creates a thread whose body of execution is the condition and action part of the rule. An instance of the process-rule-list is created (which is basically a node in the list) and inserted into the list. The thread-id is obtained when creating the thread and the parents thread-id is obtained using the thr-self() thread API call.

At the end of every notify, the scheduler is woken up. The scheduler now traverses the process-rule-list performing actions which will be described later. At the end of the notify, the function 'wait-for-immchild' is called. 'wait-for-immchild' is a function that waits until all triggered rules in the 'Immediate' coupling mode finish execution.

The commit of a transaction is an event at which time a pre-defined rule is triggered. This rule has an immediate coupling mode and has a priority of '-1'. This priority is chosen to differentiate this rule from others generated by the application. The action of this rule first puts the scheduler to sleep and then calls the function 'process-defrules'. The scheduler is put to sleep because the function 'process-defrules' does a consistency check on the *process-rule-list* making sure that there are no other immediate rules except the one with priority '-1'. The function also sets the 'deferred-flag' which indicates to the scheduler that deferred rules can now be processed. After setting the flag, 'process-def-rules' wakes up the scheduler which triggers the appropriate rules based on the scheduling algorithm. The commit waits for the completion of all deferred rules before the application can proceed. A condition variable is used on which the action of the commit rule waits. This condition variable is triggered by the scheduler once all the rules in that transaction complete processing.

As briefly mentioned in section 5.2 and 5.3, concurrent access is needed for the process-rule-list. Many threads have to access or/and modify the process-rule-list. This could lead to inconsistencies. Mutex locks have been used to avoid incorrect update of this list. When any part of the process-rule-list is under modification, the mutex-variable is locked and is released at the end of the operation. This ensures that no other thread can modify the process-rule-list as that thread in turn would try to lock the mutex-variable, which is not possible and that thread would be put on block until the first thread unlocks the mutex-variable.

As shown in figure 5.2, the application proceeds normally until it reaches a method call that is detected as an Event. This part is taken care of by the snoop preprocessor, which parses through the application code searching for method calls that match event methods already declared in the specification file.

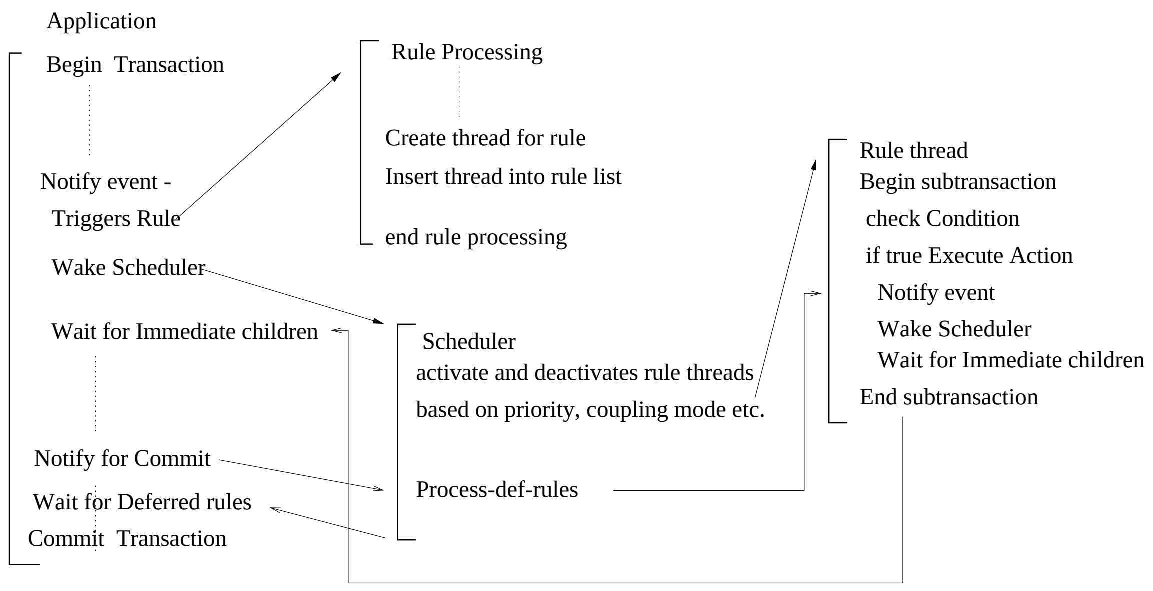

Figure 5.2. Implementation overview

Once an event is notified, a set of rules maybe triggered. This is done in the Rule Processing stage as shown in the figure 5.2. For each rule that qualifies, a separate thread is created and inserted into the rule list. Recall that, the thread is created in the suspended state.

The notify now calls the function wait-for-immchild. This function wakes the scheduler and the rules are executed based on their priorities and coupling modes. This function then looks at the process-rule-list and based on the current thread-id (got by using thr-self()) finds if there are any rules triggered by this thread in the Immediate coupling mode. If there are, then the function waits for the completion of those threads (using thr-join).

When the scheduler activates a rule, it wakes up the thread associated with the thread-id in the process-rule-list. The thread routine is enclosed within a sub-transaction for reasons already mentioned. The thread checks the condition of the rule and executes the action associated with the rule if the condition evaluates to true.

To process deferred rules, we define a rule which is triggered by the event - commitTransaction. The action part of this rule calls a function process-deffrules, as shown in figure 5.2. This function activates all the rules in cycle-1 of the process-rule-list. It then waits for the completion of deferred rules, if any, that was spawned by this transaction. When a rule thread completes its execution the corresponding rule node in the rule list is deleted and at the end of the top-level transaction, the rule list is empty.

5.8 Inserting Rule Nodes

When a rule is triggered in the application, the rule-node that is created as mentioned above has to be inserted into the process-rule-list. The placement of rules in the list depends on the coupling mode, its parent and its priority. The rules are inserted in decreasing order of priority. Also the process-rule-list is sub-divided into two sections, one for rules with Immediate coupling mode and the other for rules with deferred coupling mode. Thus in a scenario consisting of rules in both coupling modes, the head rule-node is an Immediate-top-level rule with the highest priority.

If the rule has been triggered from within the action of another rule then the parent rule is searched for and the new rule is placed after the parent in the process-rule-list. Again if there are more than one child rule for a parent, then the placement is based on the relative priorities of the sibling rules. The operating mode of all new rules inserted into the process-rule-list is 'Ready'. In case of rules with immediate coupling mode when a rule is inserted the parents operating mode is changed to 'Wait'. The parent rule has to wait for the termination of all child rules before it can proceed since its coupling mode is Immediate. This is not the case for rules that trigger rules in the deferred coupling mode

The basic pattern of rule insertion based on priority is retained for deferred rules. The main difference is that the placement is based on cycles. All deferred rules triggered by the top-level transaction and those triggered by rules in the immediate mode are part of 'cycle-1'. A rule with a deferred coupling mode triggered during the execution of any cycle-n ($n > 0$) rule is placed at the end of all rules in cycle-n. The rule becomes part of the next cycle of rules, cycle-(n+1). The placement of rules in a cycle is based on its priority if the coupling mode is deferred. If a rule in any of the deferred cycles triggers a rule with an immediate coupling mode then that rule is inserted next to its parent although the parent's coupling mode is deferred. Here again the the triggering rule has to wait for the completion of all the rules that it triggers which have an immediate coupling mode.

Figure 5.3 shows a possible scenario of the process-rule-list after inserting rule nodes for each rule triggered. The top level rules that are triggered from the application are R1, R2 and R3. Notice that the parents of these top level rules are given the value 1. This is the default id of the 'main()' thread and indicates that the rule has been spawned from the main thread and are top level rules. R1 and R2 are rules that are in the immediate coupling mode while R3 has a deferred coupling mode. Thus the main transaction waits for the completion of R1 and R2's execution from which they were triggered. The transaction then continues and at commit time, allows R3 to execute.

Notice that R1 and R2 are placed in the list based on their relative priorities, here R1 has a higher priority than R2. R3 is placed at the end of all Immediate rules and becomes part of cycle-1. This is done inspite of R3 having a higher priority than R2, because R3 has a deferred coupling mode while R2 has an immediate coupling mode.

R11, R12 and R13 have been spawned from the action execution of R1. R11 and R12 have been placed after R1 in the rule list while R13 has been placed in cycle-1 of the deferred section as it has a deferred coupling mode. R11 and R12 have been placed based on their relative priorities. Similarly R21 and R23 are rules triggered

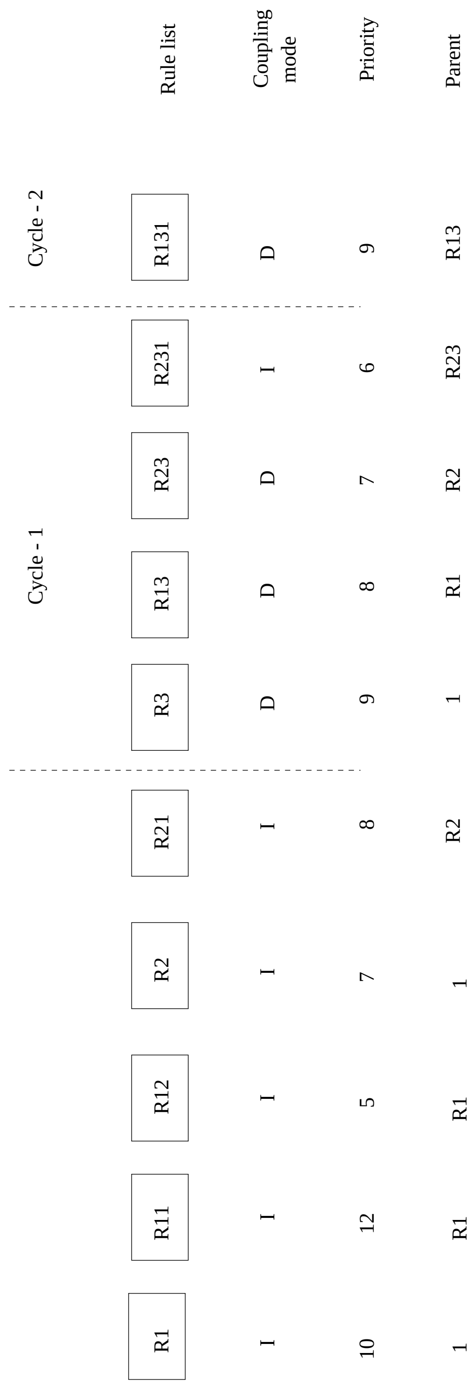

Figure 5.3. Rule List scenario

from R2 and are placed at specific points in the table based on the placement of the parent and the coupling mode.

In the present scenario let us assume that the deferred rules have started execution. Rule R3 starts executing following which rule R13 executes, this is based on their priorities. As we can see during the execution of rule R13 it spawned another rule R131. R131 is then put at the end of all rules in cycle-1 and becomes part of cycle-2. The execution of R131 has to be started only after execution of all rules in cycle-1.

Another point that is of importance is that rule R231 which is spawned from R23 has an immediate coupling mode. It is still added to cycle-1 because it got spawned by a rule executing in a deferred coupling mode. Thus it is placed after R23 in the rule list inspite of its coupling mode. So R23 has to wait for the completion of R231's execution before continuing.

5.9 Scheduler

The scheduler runs in an infinite loop acting on the process-rule-list. Though in an infinite loop, the scheduler (which is a thread) can be suspended and awakened by any other thread in the system.

The scheduler starts at the head of the process-rule-list. If the Head points to a rule with priority '-1' and there are no deferred rules in the list, it means that all rules that were triggered within that transaction have completed execution, the scheduler then sends a signal to the triggering transaction informing it of the status. If this is not true, the scheduler checks for the 'deferred-flag'. The 'deferred-flag' indicates to the scheduler if the transaction has started processing deferred rules. If this is true, the scheduler looks at the operating modes of the rules in the first cycle. An operating mode of ready indicates that the rule is ready to execute, the scheduler awakens that thread allowing rule execution. An operating mode of 'WAIT' indicates that the rule is waiting for the completion of rules that are in the immediate coupling mode. This process is repeated for all rules in the first cycle. If the 'deferred-flag' is not set to true, the scheduler looks for immediate rules in the process-rule-list.

The scheduler operates on rules in the process-rule-list based on its operating mode. Starting from the header, if the operating mode is 'Ready' the scheduler fires the rule execution and all the following top-level rules that have the same priorities. Whenever a rule is fired the operating mode of that rule in the process-rule-list is changed to 'Executing'.

If the operating mode is 'Wait' it means that the thread is on wait, which further means that there are some child rules that need processing. This is done recursively in a function called child-recurse().

The operating mode 'Finish' symbolizes the completion of the rule processing of that particular rule. What needs to be checked in this case is whether all Immediate rules have completed. If they have then all deferred rules in the first cycle can be executed based on their priorities. If all Immediate rules have not been processed then based on their priority, they are allowed to execute.

The scheduler acts similarly on all top level rules making sure that rules in the rule queue are processed.

child-recurse()

This function is called by the scheduler when it finds the operating mode of one of the rules to be 'wait'. This function takes care of nested rules. Once again processing is based on the operating modes of the child rules. The highest priority child rules which have the operating mode as 'Ready' are fired and the operating mode is changed to 'Executing'. There could be one more level of nesting where the child rules are themselves in the 'Wait' state because more rules are triggered during their execution. This would mean that we have to make a recursive call to child-recurse().

As discussed earlier, after every notification the thread waits for all the rules it spawned in the immediate coupling mode to finish execution. The method is called 'wait-for-immchild'. This function searches the process-rule-list for the triggering rule. This can be done using the thr-self() function call which returns the thread-id of the executing thread, using which the list can be searched for that thread-id. Once found the scheduler traverses from that point on doing a thr-join() on every following thread-id whose parent is the current thread-id. If the current thread is the main transaction, the thread-id is going to be 1. In this case the whole list has to be searched for rules with parent-id 1, having an immediate coupling mode.

The function 'process-deff-rules' is called by the action of the commitTransaction rule. This function first checks to see if there are any deferred rules that have been generated by the transaction. If there are no rules the function returns, otherwise it traverses the list of deferred rules in the first cycle and starts their execution in order of their priorities. The 'deferred-flag' is set to true. This tells the scheduler that the deferred rules can now be processed.

CHAPTER 6
EXAMPLE APPLICATION SHOWING RULE SCHEDULING

This is an application which uses Sentinel's rule definition language and OQL (Object query language). It uses OpenOODB's transaction calls which in turn calls Sentinel's transaction manager, which supports the nested transaction semantics that have been described earlier. The motivation behind this example is to show the features of the rule scheduler and how it operates on multiple, nested rules generated by the application. It also shows how the rule scheduler handles priorities and coupling modes of rules.

6.1 Event and Rule Definitions

The application is based on a Hospital application written for Sentinel.

```
class MedicalRecord {
....
....
// Event definitions
event end(event_setRegNum) void set_reg_num(int reg_num);
event end(event_getRegNum) int get_regist_num();
event end(event_get_date) char* get_date();

// Rule definitions
rule R01[event_setRegNum,true_condition,action_getRegNum,RECENT,6,IMMEDIATE];
rule R02[event_setRegNum,true_condition,action_update,RECENT,3,IMMEDIATE];
rule R03[event_setRegNum,true_condition,action_getDate,RECENT,8,DEFERRED];

rule R04[event_getRegNum,true_condition,action_update,RECENT,8,IMMEDIATE];
rule R05[event_getRegNum,true_condition,action_update,RECENT,8,DEFERRED];

rule R06[event_get_date,true_condition,action_update,RECENT,5,DEFERRED];
rule R07[event_get_date,true_condition,action_update,RECENT,9,IMMEDIATE];
};
```

The class MedicalRecord has three methods that raise an event if invoked. 'set_reg_num' raises event 'event_setRegNum', 'get_regist_num' raises event 'event_getRegNum' and 'get_date' raises event 'event_get_date'. 'end' indicates that the rule has to be triggered after the event method is called. The other option would be 'begin', where the rules are triggered before actually calling the event method. We have defined six rules subscribing to these three events. The rule definition includes the following information : R01 is the rule's name which is triggered by event event_setRegNum, the condition to be checked is defined by function 'true_condition' and if true the action to be performed is defined by function 'action_getRegistNum'. The next argument defines the context which in this case is 'RECENT', which is followed by the rule priority, 6 in this case and the last argument defines the coupling mode, which is immediate here. Thus rules R01, R02 and R03 subscribe to event event_setRegNum, R04 and R05 subscribe to event event_getRegNum, and R06 and R07 subscribe to event event_get_date.

6.2 Condition and Action Functions

The conditions and actions for all the rules defined in the previous sections have to be defined.

```
int true_condition(L_OF_L_LIST *n1_list)
{
  return 1;
}
```

From the rule definitions we can see that all rules execute the same condition, 'true_condition'. To make things simple we just return true when the condition is executed, this ensures that all the rules will be triggered.

```
void action_getRegistNum(L_OF_L_LIST *n1_list)
{
  ....
  l1 = n1_list->getFirst();
  l2 = l1->get_head();
  u = (MedicalRecord *) l2->get_reactive_obj();

  u->get_regist_num();
  ....
}

void action_update(L_OF_L_LIST *n1_list)
{
  ....
  ....
}

void action_getDate(L_OF_L_LIST *n1_list)
{
  ....
  LIST_OF_LIST  *l1;
  LIST_NODE     *l2;

  l1 = n1_list->getFirst();
  l2 = l1->get_head();

  u = (MedicalRecord *) l2->get_reactive_obj();

  u->get_date();
  ....
}
```

Action function 'action_getRegistNum' gets the reactive object which it extracts from the argument. It then calls method regist_num() which as defined earlier is a method from class MedicalRecord. This raises the event 'event_getRegNum, to which rules R04 and R05 subscribe. Action function 'action_getDate' calls on method date which generates event 'event_get_date' to which rules R06 and R07 subscribe. Action function 'action_update' does not raise any event.

6.3 Main Application

The main application consists of just one transaction where a MedicalRecord object is fetched from the database and a member method is called.

```
main()
{
  .....
  .....
  OpenOODB->beginTransaction();

  medrec = (MedicalRecord*) OpenOODB->fetch("medical-record");

  medrec.set_reg_num(11111);

  OpenOODB->commitTransaction();
}
```

This program is run through the preprocessor which detects set_reg_num as an event. We also detect beginTransaction and commitTransaction as events. The Notifies are set for each event and the application is compiled. On execution the MedicalRecord object is fetched and the method set_reg_num is called. After execution of set_reg_num the Notify for that event is executed. Within the Notify, three threads are created, one for each rule. Three nodes of the process-rule-list are also created and are inserted into the process-rule-list.

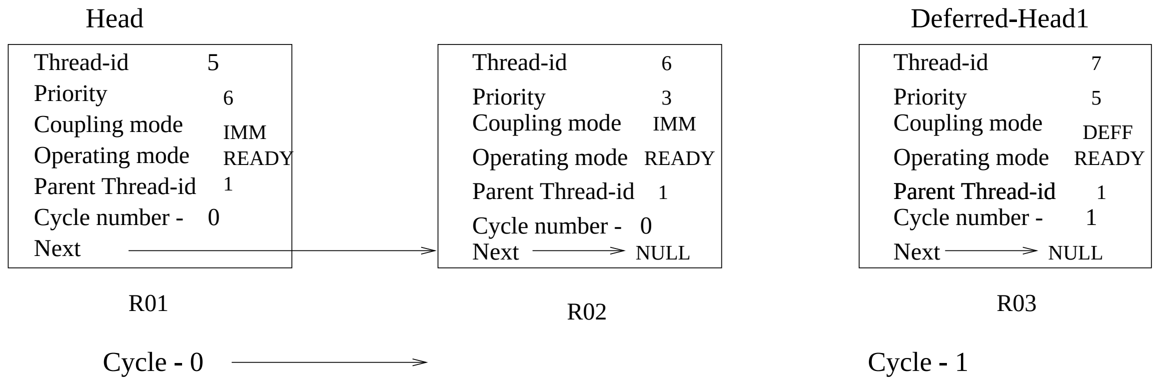

Figure 6.1. Process-rule-list

From the figure 6.1 we see that rule R01 has thread-id 5, priority 3, and parent 1 which is a top level transaction. Rule R02 with thread-id 6 is placed after rule R01 because of its lower priority. Rule R03 with thread-id 7 is put into another list since it has a deferred coupling mode. This rule R03 becomes part of cycle-1 of deferred rules. The operating mode for each rule is 'READY'.

The next call within Notify is 'wait-for-immchild'. This suspends the top-level transaction and wakes up the scheduler. The scheduler traverses the process-rule-list and awakens thread with thread-id 5. This is the only rule that can execute, as rule with thread-id 6 has a lower priority and rule with thread-id 7 is in the deferred mode. The execution of rule R01 starts and the operating mode is changed to 'EXE' or executing. The body of the thread first executes the condition given by function 'true_condition' which returns true. The rule continues and executes the action which is the function 'action_getRegistNum'. As seen above, the function 'action_getRegistNum' calls the method get_regist_num. Remember, get_regist_num detects event 'event_getRegNum', which spawns two rules R04 and R05.

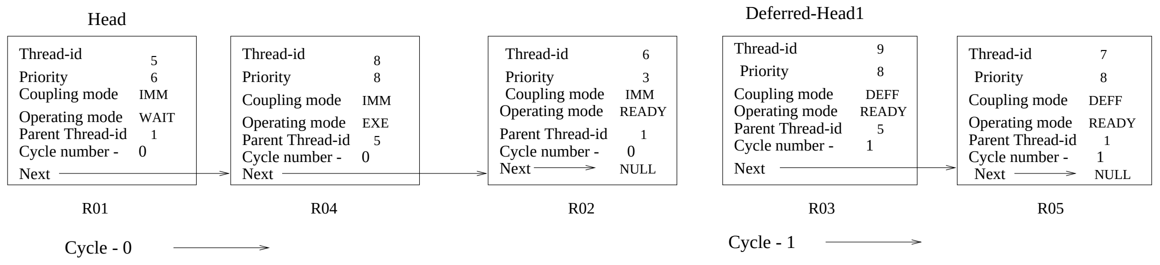

Figure 6.2. Process-rule-list

The Notify for this event creates two more threads, with thread-id 8 for rule R04 and thread-id 9 for R05. Thread 8 is placed after thread-5 which represents rule R01, that triggered it. Thread 9 is placed before thread 7 in the deferred rule list, it does not really matter if its placed before or after thread 7 because both have to same priority and will be executed simultaneously. This can be seen in figure 6.2. As soon as rule R04 is inserted into the list, its parent's operating mode, rule with thread-id 5, is changed to 'WAIT'. 'wait-for-immchild' is again called within this Notify. The scheduler gets woken and finding rule R01 in the 'WAIT' state, it calls 'child-recurse'. 'child-recurse' gets to rule R04 and allows it to execute changing its mode to 'EXE'. The body of R04 executes 'true_condition' which again returns true and goes on to execute action 'action_update'. Once 'action_update' finishes executing, it deletes its node from the process-rule-list. Now thread 5 representing rule R01 continues, as thread 8 was the only rule that it had spawned with an immediate coupling mode. The current state of the process-rule-list is shown in figure 6.3.

Rule R01 finishes processing and its node in the process-rule-list is removed. The scheduler now finds only rule R02 in the immediate rule list. It awakens its thread which executes 'true_condition' followed by 'action_update'. The top-level transaction

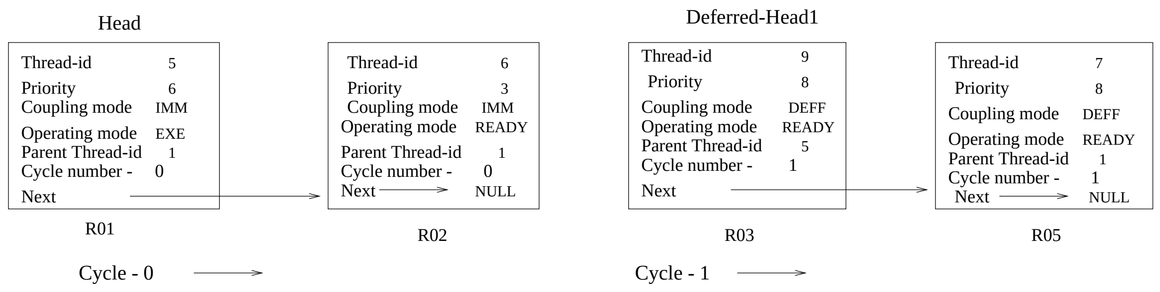

Figure 6.3. Process-rule-list

can now continue as both the rules R01 and R02 with immediate coupling mode have finished execution.

The application finally reaches the commitTransaction. As mentioned before, commitTransaction is an event and is wrapped with a Notify. A pre-defined rule is triggered for the commitTransaction, it has a priority of -1 differentiating it from all other rules in the system. It has an immediate coupling mode, the condition of the rule is always true and the action calls function 'process-deferred'. This rule is placed at the head of the process-rule-list and has thread-id 10.

The scheduler is woken up and it continues thread 10 because it has an immediate coupling mode. The action, i.e. function 'process-deferred' is called. This function calls the 'process-deff-rules' function whose operations have been described in detail in chapter-5, after which it blocks on a condition variable. This condition variable is signaled by the scheduler once all rules spawned by this transaction completes execution. 'process-deff-rules' traverses the deferred rule list and awakens thread 9 which corresponds to rule R05 and thread 7 which corresponds to rule R03, as they have the same priority. The condition and action of both rules are executed. This can be seen in figure 6.4.

After execution of rule R05 the node is removed from the list. Thread 7 or rule R03 continues execution. The action calls function 'action_getDate'. Looking at the body of 'action_getDate' we can see that it calls function get_date. This raises

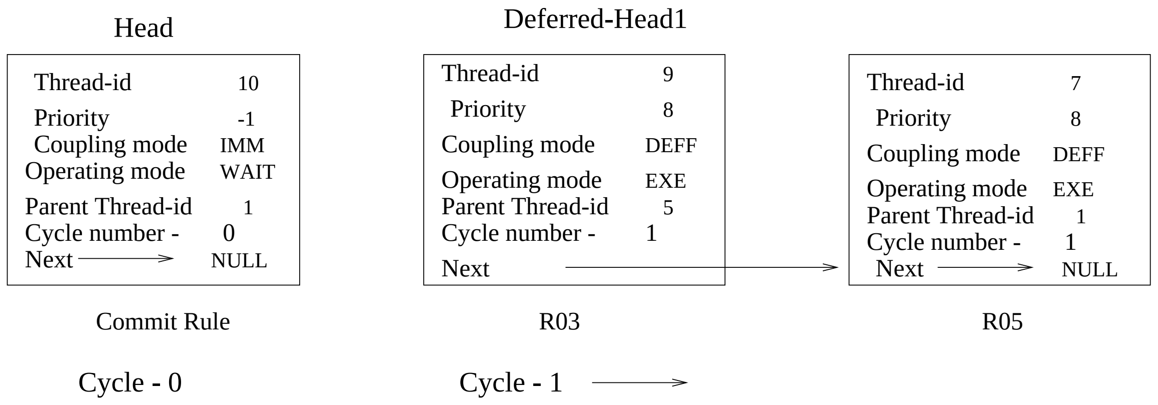

Figure 6.4. Process-rule-list

event, 'event_get_date' and within its Notify, it creates two thread. One with thread-id 11 representing rule R07 and the other with thread-id 12 representing rule R08. Thread 11 has been placed after thread 7 because the rule has an immediate coupling mode. The operating mode of rule R03 is changed to 'WAIT'. Thread 12 or rule R8 is part of the next cycle of deferred rules, in this case cycle-2. Rule R03 executes 'wait-for-immchild' and the scheduler seeing the 'WAIT' calls 'child-recurse'. This function as mentioned before awakens thread 11 or rule R07, which executes condition 'true_condition' and action 'action_update'. This can be see in figure 6.5.

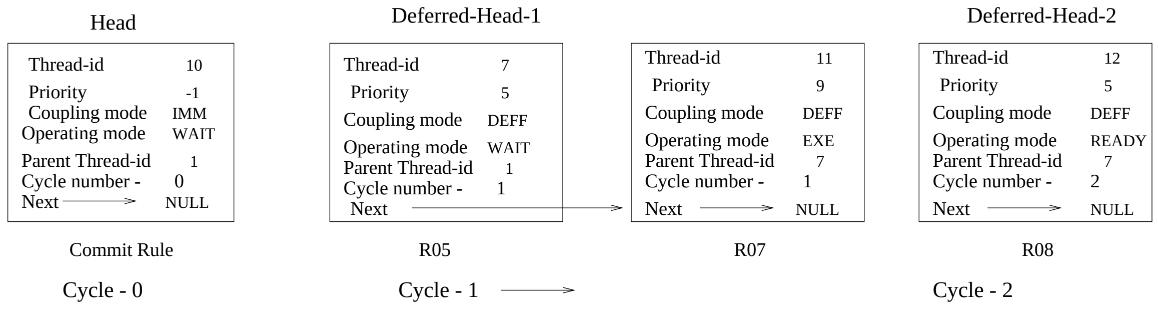

Figure 6.5. Process-rule-list

After it is done the node for rule R07 is deleted and the parent rule R03 continues. Once R03 finishes execution, the scheduler traversing along the process-rule-list finds no more rules in cycle-1. It thus awakens rules in cycle-2. In this case only one rule R08 exists which is continued. Now this becomes the top cycle and the next cycle is NULL. Thus at any time there can only be two cycles of deferred rule execution. This is shown in figure 6.6. Once its condition and action finishes executing, the node is removed from the process-rule-list.

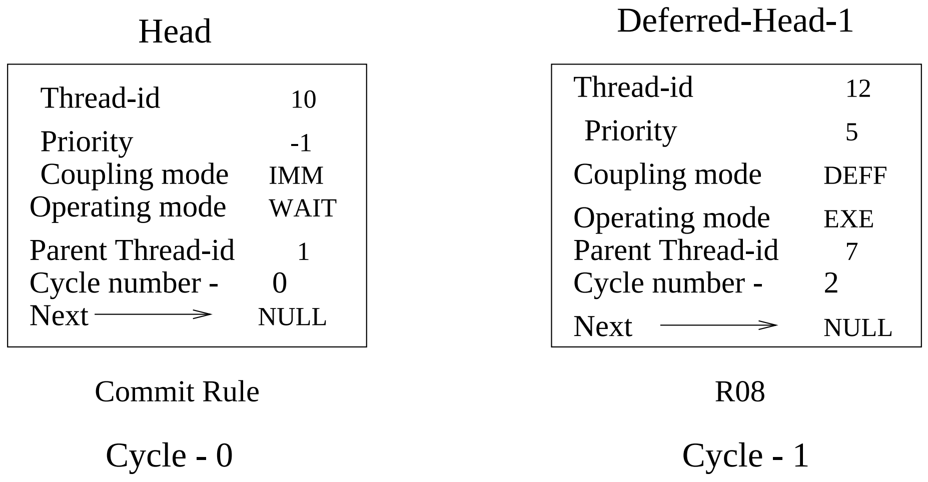

Figure 6.6. Process-rule-list

The scheduler now finds no more rules to execute so it sends a signal to the waiting function 'process-deferred'. The signal is received and the rule finishes execution leaving the process-rule-list empty. The transaction now commits and the application proceeds.

CHAPTER 7
PERFORMANCE ANALYSIS

This chapter outlines some of the tests done to measure performance of the system using the rule scheduler. The scheduler allows concurrent execution of rules if they have the same priority. These tests try to show the improved performance achieved by multi-threading the rule execution.

To show the effect of multi-threading we need to run an application on machines with more than one processor. One option would be to run the same application on a single processor machine, then on a machine with 2 processors and so on. The problem with this approach is that the performance differences are harder to measure owing to the different configurations of the processors on each machine. Moreover we need to have machines with different number of processors to run them on.

The other approach is to run them on one machine and simulate different processing conditions. This is the approach taken here. The tests were run on a machine with eight processors. To simulate the different number of processors, the application was modified such that at run time it was possible to restrict the number of concurrent threads allowed to run.

The application itself consists of a simple event triggering thousand rules. The rules were defined as follows :

```
for(int j=0; j<1000; j++)
  {
    RULE *test_rule = new RULE("test_rule",MedicalRecord_event_date,
                               true_cond,action_compute,RECENT);
    temp_rule->set_priority(10);
    temp_rule->set_mode(IMMEDIATE);
  }
```

The above rule definition creates 1000 rule definitions for the event 'event_date' of class MedicalRecord. The condition of the rule is 'true_cond' and the action is 'action_compute', the priority is set to 10 for all the rules and all of them have an immediate coupling mode.

The condition 'true_cond' always returns a true. The action 'action_compute' is as follows :

```
void action_compute(L_OF_L_LIST *n1_list)
{
  int i,j;
  printf("In action-compute, thread %d \n",thr_self());
  for (i = 1 ; i<10000 ; i++)
  {
    for(j = 1; j < 1000; j++)
    {
      i*j%i;
      i*i/j*j;
      i^(j%10);
    }
  }
}
```

The action function is just a CPU intensive process involving nested loops and mathematical computations, this ensures that the threads run for a considerable amount of time bringing out any concurrency control issues.

The application is first run such that just one thread is allowed to execute at one time. To allows this to happen, the scheduler had to be modified to allow only a specific number of rules (threads) to continue at one time, even if more rules were eligible to execute. This first case emulates a uni-processor environment where there is no possible concurrent execution. Next the number of threads allowed to execute concurrently was increased from two and so on upto fourteen. For each execution scenario the CPU time and the total time for the application to execute fully were calculated.

The results are plotted on the graph in figure 7.1.

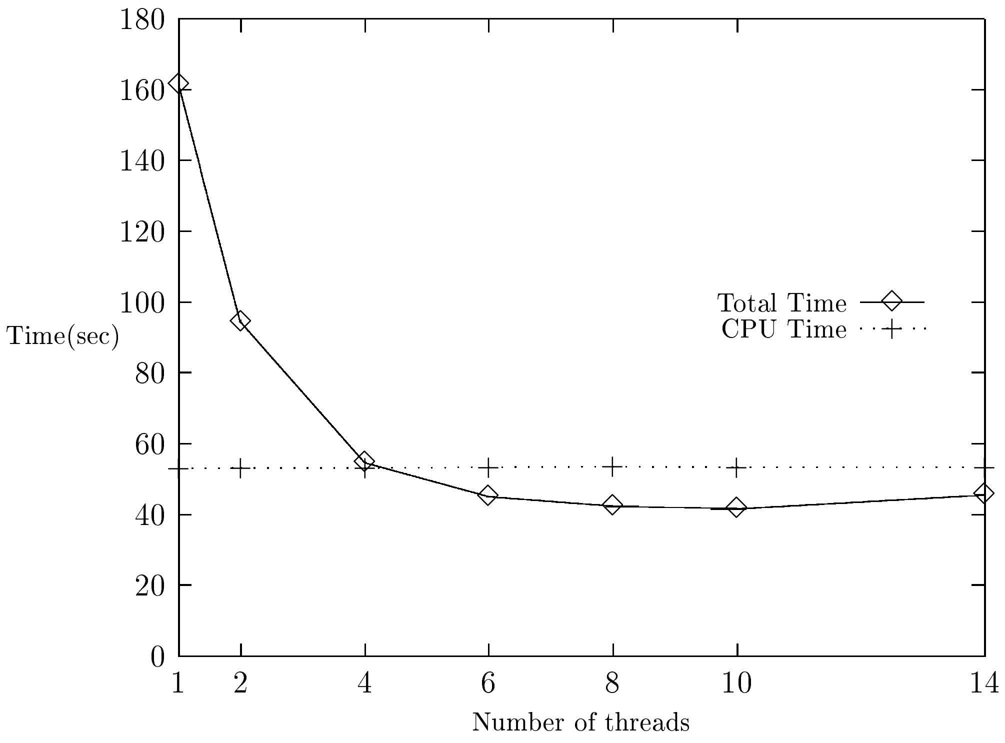

Figure 7.1. Performance analysis 1

The chart shows that the CPU time is almost exactly the same for all the test scenarios. The total time taken though shows significant improvement from a single processor mode to a dual processor mode. This improvement in time steadily increases all the way till a six processor mode. After which the the time taken by increasing the number of concurrently executing threads does not show any improvement. This is so because the machine, as mentioned before, has six processors and irrespective of how many threads are started, only six can execute concurrently at one time.

To test the effect of I/O operations on rule processing, the action of the rule was changed as follows:

```
void action_compute(L_OF_L_LIST *n1_list)
{
  int i,j;
  printf("In action-compute, thread %d \n",thr_self());

  TestClass *testc1= (TestClass*) OpenOODB->fetch("objs1");
  TestClass *testc2 = (TestClass*) OpenOODB->fetch("objs2");
  TestClass *testc3 = (TestClass*) OpenOODB->fetch("objs3");
}
```

Each rule triggered fetches 3 objects from the database. What we are trying to see here is, if the I/O operations serialize the rule execution as they all have to fetch the same object at the same time. The performance graph got from the experiment is shown in figure 7.2.

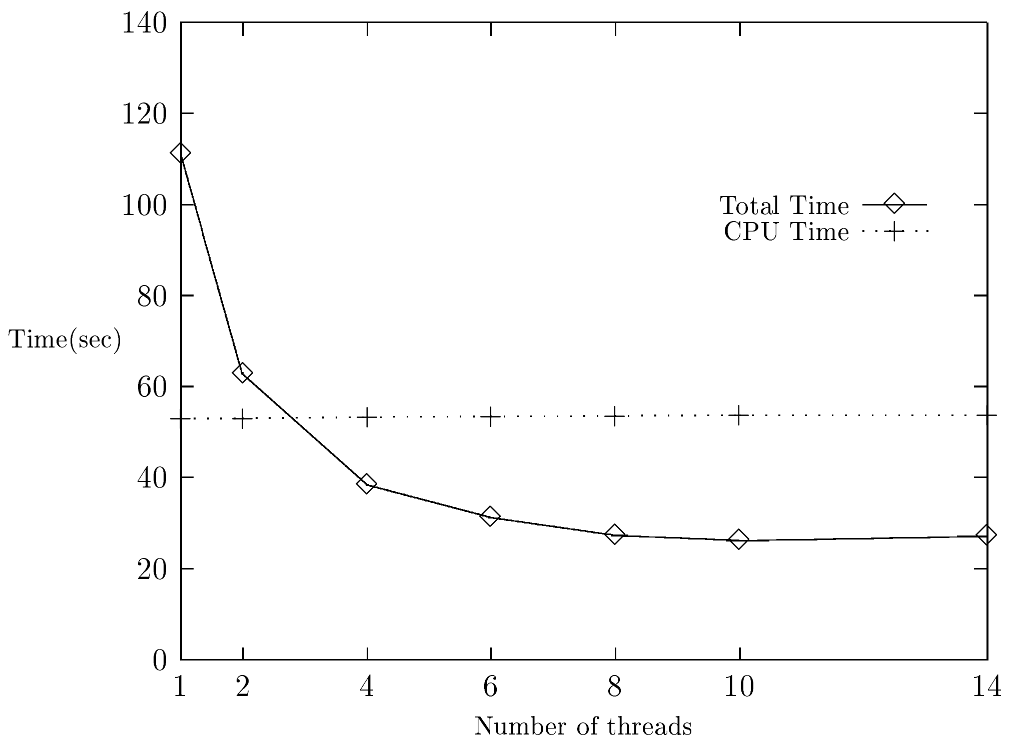

Figure 7.2. Performance analysis 2

We notice from the graph that the performance was not really affected by the I/O operations. This was true because, the storage manager stores the fetched object in a cache after each fetch operation. Subsequent fethces need not get the object from the disk. Instead they are read directly from cache memory and so the performance comes out to be quite similar to the previous experiment.

CHAPTER 8
CONCLUSION

This theses extends the rule processing already present in Sentinel. So far rules are processed serially in some order without using the priority conformation.

The scheduler designed and implemented in this theses supports the semantics of rule execution within a transaction. It allows multiple rules associated with the same event to run concurrently if they have the same priorities. This is achieved by multi-threading the rule execution. Each rule executes as a separate thread.

Every rule can be assigned a priority. This allows a well defined order of rule execution among rules in the same level (or sibling rules). Sibling rules withe same priority execute simultaneously.

Rules can be assigned coupling modes. This allows applications to preserve (or even check) the consistency of a transaction by executing rules at the end of the transaction. This is done to preserve consistency. Such rules can be executed with a deferred coupling mode. Other rules can execute with an immediate coupling mode, where the rules are executed as soon as they are triggered.

Another important topic addressed is the Nested transaction scenario that has been used in this implementation. The most important effect that comes about by using nested-transactions is concurrency control for rules that are executing concurrently. Thus when there are no concurrency issues, the scheduler can safely execute as many threads as possible in parallel.

As part of the theses, a couple of applications were developed to test the working of the scheduler. Multiple levels of nesting was achieved and the performance improved by multi-threading the rule processing.

APPENDIX A

In this section we present the algorithms to insert and delete a rule node into the rule list as discussed in the previous chapter.

```
void insert_rule(thread_id, int priority, int parent, int coupling_mode)
    //All arguments belong to the new rule node that has to be inserted
    if (coupling_mode == IMM)
      if (parent == top_level_tx)
        if (Header == NULL)
          Header = this
          return
        if (this_priority >= Header->priority)
          this_next = Header
          Header = this_rule
        else
          Search among the top level rules and insert this rule
          based on its relative priority.
      else
        search for parent first among immediate rules and
        place this rule among its siblings relative to its priority
        Change the parents operating mode to 'WAIT'
        If parent not found among immediate rules find it in the
        first cycle of deferred rule and insert likewise, again
    changing the parents operating mode to 'WAIT'
else if (coupling_mode == DEFF)
      Get the head of deferred rules of the first cycle
      if (Def_Header1 == NULL)
        Def_Header1 = this
      else if (parent == top_level_tx)
            if (this_priority >= Def_Header1->priority)
              this_rule_>next = Def_Header1
              Def_Header1 = this_rule
            else
            place it relative to other top level rules
            with respect to its priority
      else if (parent != top_level_tx)
            search for parent among Immediate rules
            if (parent is found)
              insert the rule based on its priority among all
              deferred rules in the first cycle
            else if (parent is not found)
                  this_cycle_num = Def_Header1->cycle_num + 1
                  If (Def_Header2 == NULL)
                    Def_Header2 = this_rule
                    return
                  if (this_priority >= Def_Header2->priority)
                    this_rule_>next = Def_Header2
                    Def_Header2 = this_rule
                  else
```

```
                        insert this rule in the next cycle based on its
                        relative priority


void delete_rule(thread_t this_id)
    if (Header != NULL and Header->tid == this_id)
      temp = Header
      Header = Header->next
      delete temp
    else if (Def_Header1 != NULL and Def_Header->tid == this_tid)
         temp = Def_Header1
         Def_Header1 = Def_Header1->next
         if (Def_Header1 == NULL)
           if (Def_Header2 != NULL)
             Def_Header1 = Def_Header2
             Def_Header2 = NULL
         delete temp
    else
         Search first among immediate and if not found there,
         then among first cycle of deferred rules for this_id and delete it
```

APPENDIX B

The scheduler operates as a Solaris thread. The thread is created at the beginning of the application. The thread operates in an infinite loop but can be made to sleep or waken up by any other thread

'Header' is the head of the immediate rules and 'Def_Header1' and 'Def_Header2' are the heads of the two cycles of deferred rules.

```
void scheduler()
while (1)
    if (Header != NULL and Def_Header1 == NULL and Header->operating mode != REA
      send signal to commit_rule, and free waiting condition
      variable, set def_flag=0
    if (def_flag == 1)
      // this inidcates if deferred rules can be processed
      def_rule = Def_Header1
      if (def_rule->operating_mode == WAIT)
        child_recurse(def_rule->tid)
      this_cycle = def_rule->get_cycle()
      while (def_rule != NULL and def_rule->cycle == this_cyle)
            if (def_rule->operating_mode == READY)
              continue the thread execution using its tid
              change operating mode to EXE
            def_rule = def_rule->next
    switch (Header->operating_mode)
    case Ready :
           continue execution of this thread and change
```

```
            operating_mode to Executing
            search the rule queue for all other top level
            rules with same priority, continue the
            thread execution and change the operating_mode to Executing
            break
case Executing :
            Yield thread execution, this allows
            rules to continue execution
            break
case Wait :
            if (Header->next == NULL)
              Error message and exit
            call child_recurse(this_tid)
            break
default : Error and exit
end case
for all top_level_rules
            if (operating_mode == Wait)
                call child recurse(this_tid)
```

child_recurse takes care of all child rules of the rule with thread_id 'tid' which is passed as an argument to this function

```
child_recurse(tid)
    search for tid in the rule queue
    The search has to be made first in the Immediate rule list
    then in the first cycle of the deferred rules
    for the first child rule do :
        switch (operating_mode)
        case Ready :
            continue the thread, change the operating_mode to
            Executing and do the same for sibling rules with
            the same priority
            break
        case Executing :
            check the operating_mode for sibling rules with
            same prio
            if (operating_mode == Wait)
              call child_recurse(tid)
            break
        case Wait :
            call child_recurse (this_tid)
            // calling child_recurse to process child rules of this rule
            // since the operating_mode is Wait.
            break
        end case
```

'wait_for_immchild()' is called within the Notify. This waits for any rule spawned by the execution of this rule, which has an immediate coupling mode.

```
void wait_for_immchild()
wake_scheduler()
imm_flag = 0
imm_rule = Header
while (imm_rule != NULL)
      if (thr_self() == top_level_tx)
        if (imm_rule != NULL and imm_rule->parent == 1)
          // That is, if the parent is a top_level_tx
          imm_flag = 1
          while (thr_join(imm_rule->tid) == 0)
          // thr_join waits for the completion of the thread specified by tid
          imm_rule = Header
        else if (imm_rule != NULL)
            imm_rule = imm_rule->next
      else if (imm_rule->tid == thr_self())
            imm_rule = imm_rule->next
            if (imm_rule != NULL and imm_rule->parent == thr_self())
              imm_flag = 1
              while(thr_join(imm_rule->tid) == 0)
              imm_rule = Header
            else if (imm_rule != NULL)
                imm_rule = imm_rule->next
      else imm_rule = imm_rule->next
if (imm_flag == 0)
```

```
// If the rule is in the deferred rule list
def_rule = Def_Header1
while (def_rule != NULL)
        if (def_rule->tid == thr_self())
           def_rule = def_rule->next
           if (def_rule != NULL and def_rule->parent == thr_self())
              while(thr_join(def_rule->tid) == 0)
              def_rule = Def_Header1
           else def_rule = def_rule->next
        else def_rule = def_rule->next
```

'process_deff_rules' is called within the commit_rule. This function starts execution of deferred rules.

```
void process_deff_rules()
if (Header == NULL)
  return
else
  if (Header->priority != _1)
    if (Header->next != NULL)
    Give out ALERT message and exit
    // At this point there can be no immediate rules being processed
    // The only immediate rule allowed is the commit_rule, which calls
    // this function as part of its action
  else
    Give out ALERT message and exit
  def_rule = Def_Header1
  while (def_rule != NULL and def_rule->couple_mode == DEF)
        if ( def_rule->operating_mode == READY)
          continue execution of the thread using the tid
          change its operating mode to EXE
        def_rule = def_rule->next
  def_flag = 1
  wake_scheduler()
```

REFERENCES

[1] D. Mishra. Snoop: An event specification language for active databases. Master's thesis, University of Florida, Gainesville, 1991.

[2] S. Chakravarthy, B. Blaustein, A. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Linvy, D. McCarthy, R. McKee, A. Rosenthal. HiPAC: A Research Project in Active Time-constrained Database Management Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA.

[3] L. Brownston, R. Farrell, E. Kant, and N. Martin. Programming Expert Systems in OPS5. Addison-Wesley Publishing Company, Inc., Reading, MA, 1985.

[4] V. krishnaprasad. Event detection for supporting active capability in an OODBMS: semantics, architecture and implementation Master's thesis, University of Florida, Gainesville, 1994.

[5] J. Widom and S. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proceedings of ACM-SIGMOD, pages 259–270, May 1990.*

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

[7] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K Kim. Anatomy of a Composite Event Detector. Technical Report UF-CIS-TR-93-039, University of Florida, Gainesville, FL, December 1993.

[8] 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*, Washington, DC, May 1993.

[9] J. Eliot, B. Moss. Nested Transactions: An Approach to Reliable Computing. The MIT Press, Cambridge, England, 1985.

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

[11] SunSoft. Multithreaded Programming Guide. Sun Microsystems, Inc., Mountain View, CA, 1994

BIOGRAPHICAL SKETCH

Shashi Neelakantan was born on July 1, 1974, in Bombay, India. He received his Bachelor of Engineering from the University of Madras in June 1995, majoring in computer and information science.

He joined the University of Florida in August 1995 to pursue a masters degree in the same field.

He worked as a teaching assistant for the University of Florida from the fall of 1995 until the fall of 1996 and was a research assistant at the Database Systems Research and Development Center until the summer of 1997.

He is presently working as a software engineer for GTE Data Services, Dallas TX.

His research interests include active and object oriented database systems and computer networks.