IMPLEMENTATION OF PARALLEL NESTED TRANSACTIONS FOR NESTED RULE EXECUTION IN ACTIVE DATABASES

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCE
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Yücel Saygın
September 1996

IMPLEMENTATION OF PARALLEL NESTED TRANSACTIONS FOR NESTED RULE EXECUTION IN ACTIVE DATABASES

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER

ENGINEERING AND INFORMATION SCIENCE

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Yücel Saygın

September, 1996

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Asst. Prof. Dr. Özgür Ulusoy(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Varol Akman.

Prof. Dr. Varol Akman

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

[illegible]

Asst. Prof. Dr. Tuğrul Dayar

Approved for the Institute of Engineering and Science:

[illegible]

Prof. Dr. Mehmet Baray, Director of Institute of Engineering and Science

ABSTRACT

IMPLEMENTATION OF
PARALLEL NESTED TRANSACTIONS FOR
NESTED RULE EXECUTION
IN ACTIVE DATABASES

Yücel Saygın
M.S. in Computer Engineering and Information Science
Supervisor: Asst. Prof. Dr. Özgür Ulusoy
September, 1996

Conventional, passive databases, execute transactions or queries in response to the requests from a user or an application program. In contrast, an Active Database Management System (ADBMS) allows users to specify actions to be executed when some specific events are signaled. ADBMSs achieve this feature by means of rules. Execution of rules is an important part of an ADBMS which may affect the overall performance of the system. Nested transactions are proposed as a rule execution model for ADBMSs. The nested transaction model, in contrast to flat transactions, allows transactions to be started inside some other transactions forming a transaction hierarchy. In this thesis, implementation issues of parallel nested transactions, where all the transactions in the hierarchy may run in parallel, are discussed for parallel rule execution in ADBMSs. Implementation of nested transactions has been performed by extending the flat transaction semantics of OpenOODB using Solaris threads. A formal specification of the proposed execution model using ACTA framework is also provided.

Key words: Active Databases, Nested Transactions, execution model, Solaris Threads, rule execution, ACTA.

ÖZET

PARALEL İÇ İÇE YUVALANMIŞ HAREKETLERİN AKTİF VERİ TABANI KURALLARININ İŞLEME KONMASINDA UYGULANMASI

Yücel Saygın
Bilgisayar ve Enformatik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Dr. Özgür Ulusoy
Eylül, 1996

Klasik veri tabanlarında, hareketler veya sorgular kullanıcının talebine karşılık işleme konur; buna karşılık, aktif veri tabanları, belli olayların sinyal edilmesi sonucu işleme konacak eylemlerin kullanıcı tarafından belirlenmesine izin verir. Aktif veri tabanları, aktif özelliklerini kurallar sayesinde gösterir. Kuralların işleme konması aktif veri tabanının önemli bir parçasıdır, ve tüm sistemin performansını etkileyebilir. İç içe yuvalanmış hareket modeli, bir kural işleme modeli olarak önceden sunulmuştur. İç içe yuvalanmış hareket modeli, normal hareket modelinden farklı olarak, hareketlerin içinde başka hareketler başlatılmasına izin verir, böylece bir hareket hiyerarşisi oluşur. Bu tezde paralel iç içe yuvalanmış hareketlerin uygulanmasından bahsedilmektedir. Paralel iç içe yuvalanmış hareketlerde, hiyerarşinin içindeki bütün hareketler paralel olarak çalışabilir ve bu şekilde sistemin verimliliği arttirilmiş olur. Paralel iç içe yuvalanmış hareketlerin uygulanması OpenOODB'nin düz hareket modeli genişletilerek gerçekleştirilmiştir. Solaris thread'leri, hareketlerin paralel çalışması amacıyla kullanılmıştır. ACTA adlı formal çerçeve yapısı kullanılarak, sunulan çalışma modeli formal olarak açıklanmıştır.

Anahtar kelimeler: Aktif Veri Tabanları, İç İçe Yuvalanmış Hareketler, işleme koyma modeli, Solaris Thread'leri, kural işleme, ACTA.

To my parents and my brother

ACKNOWLEDGMENTS

I am very grateful to my supervisor, Asst. Prof. Dr. Özgür Ulusoy for his invaluable guidance and motivating support during this study. His instruction will be the closest and most important reference in my future research. I would also like to thank Prof. Alejandro Buchmann for his guidance, Dr. Juergen Zimmerman, and Alin Deutsch for their technical support, my colleague Uğur Çetintemel who always provided me with technical and moral support, Prof. Sharma Chakravarthy for providing me with a platform for integrating my implementation, Shashi Neelakantan and Hyoungjin Kim for their technical support during the integration, my colleagues Aynur Akkuş, Bilge Say and Murat Bayraktar for their moral support, my friends Gökhan Tür, Dilek Z. Hakkani and A. Kurtuluş Yorulmaz who were always with me with their invaluable moral support, my family for their moral support and patience during the stressful moments of my work, and last but not the least, Tahsin Mertefe Kurç, who was always ready for help with his priceless technical knowledge and experience.

Finally, I would like to thank the committee members Prof. Dr. Varol Akman and Asst. Prof. Dr. Tuğrul Dayar for their valuable comments, and everybody who has in some way contributed to this study by lending moral and technical support.

Contents

1 Introduction **1**

2 Rule Execution in ADBMSs **5**

2.1 Overview of ADBMS Rule Execution 5

2.2 Nested Transactions for Rule Execution 8

2.2.1 An Introduction to Nested Transactions 8

2.2.2 Concurrency Control and Recovery Issues in Nested Transactions 9

3 Execution Model **11**

3.1 ACTA 14

3.2 Axiomatic Definition of Parallel Nested Transactions in ACTA . 19

3.3 A Formal Model for Rule Execution in Active Databases using ACTA 21

3.3.1 Coupling Modes 22

3.3.2 A Formal Model Using ACTA 24

4 Implementation of Nested Transactions **27**

4.1 Previous Work 27

4.2 Implementation 28

4.3 Controlling Concurrent Access to Common Data Structures 35

4.4 Implementation of the Locking Protocol for Nested Transactions 37

4.5 Deadlock Detection 39

4.6 Integration of Our Implementation of Parallel Nested Transactions to Sentinel 43

5 Conclusions and Future Work 45

A Sample Pseudocodes 47

A.1 Processing Lock Requests 47

A.2 Deadlock Detection 51

Chapter 1

Introduction

Conventional, passive, databases execute queries or transactions only when explicitly requested to do so by a user or an application program. In contrast, an active database management system (ADBMS) allows users to specify actions to be executed when specific events are signaled. The concept of active databases has been originated from the production rule paradigm of Artificial Intelligence (AI). The AI production rule concept has been modified for the active database context so that rules can respond to the state changes caused by the database operations [HW92]. An active database implements reactive behavior since it is able to detect situations, which may occur in and out of the database, and to perform necessary actions which were previously specified by the user. In the absence of such an active mechanism, either the database should be polled or situation monitoring should be embedded in the application code. Neither of these approaches is completely satisfactory. Frequent polling degrades performance of the system and infrequent polling may deteriorate the timeliness of system responses. Embedding situation monitoring in the application code is error prone and reduces the modularity of the application [Day88].

Active databases is now an actively researched area since it has many applications that cannot be supported in a time-critical and efficient manner by a conventional database. Applications of active databases cover a wide range of areas like authorization, access logging, integrity constraint maintenance,

RULE1: Inventory Control
Event: $Update\ Quantity\_On\_Hand(item)$
Condition: $Quantity\_On\_Hand(item) < Threshold(item)$
Action: $Submit\_Order(item)$

RULE2: Access Logging
Event: $Update\ User\_Accounts$
Condition: $True$
Action: $Insert\ Into\ Security\_Log\ Values(User, Timestamp)$

RULE3: Power Plant Control
Event: $river \rightarrow updateWaterLevel(x)$
Condition: $x < 37 \ \wedge river \rightarrow getTemp() > 24.5$
$\wedge\ reactor \rightarrow getHeatOutPut() > 10000$
Action: $reactor \rightarrow reducePlannedPower(0.05)$

Figure 1.1: Sample Rules

alerting, network management, air traffic control, computer integrated manufacturing, engineering design, plant and reactor control, tracking, monitoring of toxic emissions, and any other application where large volumes of data must be analyzed to detect relevant situations [Day88], [BDZ95]. Active database systems are proposed for system level applications as well, like supporting different transaction models [CA95].

In a typical ADBMS, system responses are declaratively expressed using Event-Condition-Action (ECA) rules [Day88]. An ECA rule is composed of an *event* that triggers the rule, a *condition* describing a given situation, and an *action* to be performed if the condition is satisfied. One of the most important concerns in ADBMS research is event, condition, and action specification. Another significant research issue in active databases is event detection. Among typical events in an ADBMS are data modification operations (e.g.,insertions and deletions), method invocations on objects, external events (e.g., application signals), temporal events, and transaction related events (e.g., begin, abort transaction) [BZBW95]. Some sample rules for a subset of the applications listed at the beginning of this section are provided in Figure 1.1. The first and second rules deal with the handling of inventory control and access logging, respectively [HLM88]. The third rule is related to power plant control which is specified in a rule language called REAL [BDZ95]. First rule is fired when an update of the quantity on hand of an item occurs, condition is checked to see whether the quantity on hand of an item goes below the threshold value for that item, and in the action part of the rule, some amount of that item is ordered. Second rule is fired when the user accounts are updated, condition is true, meaning that the action will be executed anyway, action for that rule is an insertion of some information into the security log about the user and time. For the third rule to be fired, water level of the river in concern should be updated, some condition about the temperature of the river and heat output of the reactor is checked, and the planned power of the reactor is reduced as an action. Basic events, such as the ones presented in the sample rules, can also be combined to form composite events by using an event algebra. Efficient event detection is of particular importance especially when the number of events to be monitored is large. Rule execution is also a significant concept in ADBMS research. Section 2.1 of Chapter 2 provides a detailed discussion of rule execution in ADBMSs.

Efficient rule execution is also important from the performance perspective of the whole system. The occurrence of an event can start the execution of some rules by firing them. If the condition part of a fired rule is satisfied, then the action part of that rule is executed. Coupling modes between event and condition, and condition and action determine when the condition will be executed relative to the occurrence of the event, and when the action will be executed relative to the condition, respectively. During the condition evaluation and action execution of a rule, some other rules may be fired. This situation may go on recursively and is called nested (or cascaded) rule firing. Nested transaction model [Mos85] is considered as a suitable tool to implement rule execution since it can handle nested rule firing well. In the nested transaction model, some transactions may be started inside some other transactions forming a transactions hierarchy. The transaction at the top of the hierarchy is called a top-level transaction, and the other transactions are called subtransactions. Subtransactions can be executed in parallel which is a desirable situation if subtransactions are performing tasks that can be overlapped. Concurrency control of parallel nested transactions is discussed in [HR93].

In this thesis, we describe a parallel execution model for rule execution in ADBMSs based on nested transactions [Mos85]. The execution model is formally specified using ACTA which is a framework for specifying extended transaction models [CR91]. An implementation of parallel nested transactions for nested rule execution is described. The locking protocol in [HR93] is implemented which allows us to control the concurrency among all the transactions in the transaction hierarchy running in parallel. Implementation has been performed by extending the flat transaction semantics of OpenOODB using Solaris threads. OpenOODB is an open (i.e., extendible) object oriented database management system developed by Texas Instruments [WBT92]. In our implementation we allow all the transactions in the transaction hierarchy to run in parallel, therefore achieving the highest level of concurrency. Solaris threads are used for running the subtransactions in parallel which provides us with efficient handling of transactions executing concurrently [Sun94]. Our implementation of parallel nested transactions is currently being integrated into Sentinel [CAM93] which is an ADBMS developed at the University of Florida.

A detailed discussion of the issues introduced in this chapter is provided in the following chapters. In Chapter 2 we provide a detailed description of rule execution in active databases. Our execution model for active databases is described in Chapter 3 together with its formal specification. Nested transactions are also discussed in the same chapter. Chapter 4 deals with the implementation issues of parallel nested transactions on OpenOODB using Solaris threads, and discusses the integration of our implementation into Sentinel. Finally in Chapter 5 conclusions and future work are discussed.

Chapter 2

Rule Execution in ADBMSs

This chapter provides an overview of rule execution in Active Databases and describes a model for rule execution in Active Databases which is formally specified in Section 3.3 together with an appropriate transaction model. We introduce the concept of nested transactions together with their concurrency control and recovery properties.

2.1 Overview of ADBMS Rule Execution

Rules in Active Databases consist of an event, a condition and an action. If the event is missing, then the result is a condition-action (CA) rule or a production rule; if no condition is specified, then the resulting rule is an event-action (EA) rule or simply a trigger [PD95]. When an event is detected, the system searches for the corresponding rules. The condition part of the rule triggered by that event is evaluated and the action is taken if the condition is satisfied. One event may cause more than one rule to be fired. Handling of multiple rules fired by an event is also an important task of rule execution. New events may also occur during rule execution which may cause triggering of other rules. This is called *cascading rule firing*. Efficient handling of cascading rule firing improves the performance of the whole system and is a desirable situation.

The action part of a rule may be executed in one transaction immediately as a linear extension of the triggering transaction. This is called *coupled execution* [HLM88]. We can give Starburst as an example of coupled execution of rules [AWH92]. In Starburst, rules are based on the notion of transitions. A transition is a database state change resulting from the execution of a sequence of data manipulation operations. Rules are activated at assertion points. There is an assertion point at the end of each transaction and users may specify other assertion points within a transaction. The state change resulting from the database operations issued by the user since the last assertion point creates the first relevant transition which triggers a set of rules. A rule r is chosen from the set of triggered rules such that no other triggered rule has precedence over it. Condition of r (if it has any) is evaluated . Action part of r is executed provided that its condition evaluates to true; otherwise another rule is chosen. After the execution of r's action, rules that are not considered up until now are triggered only if their transition predicates hold with respect to the predicate created by the composition of the initial transition and the execution of r's action. Rule processing terminates after all triggered rules are executed.

Although coupled execution is useful in some cases, it degrades the performance of the system by increasing the response time of transactions. If we allow actions to be executed in separate transactions, then the triggering transaction can finish more quickly and release resources earlier, and this way transaction response times can be improved. We may also want the condition part of the rule to be executed as a separate transaction since conditions which are queries on the database can be long and time consuming. Allowing conditions and actions to be executed in separate transactions is called *decoupled execution* [HLM88].

It is also important to specify when the condition will be evaluated relative to the triggering event and when the action will be executed relative to the condition evaluation. This is achieved by defining *coupling modes* for conditions and actions. There are three basic coupling modes: *immediate*, *deferred*, and *detached* (or *decoupled*) [Day88]. Basic coupling modes between event and condition are illustrated in Figure 2.1. If the condition is specified to be evaluated in *immediate* mode, then it is executed right after the triggering operation

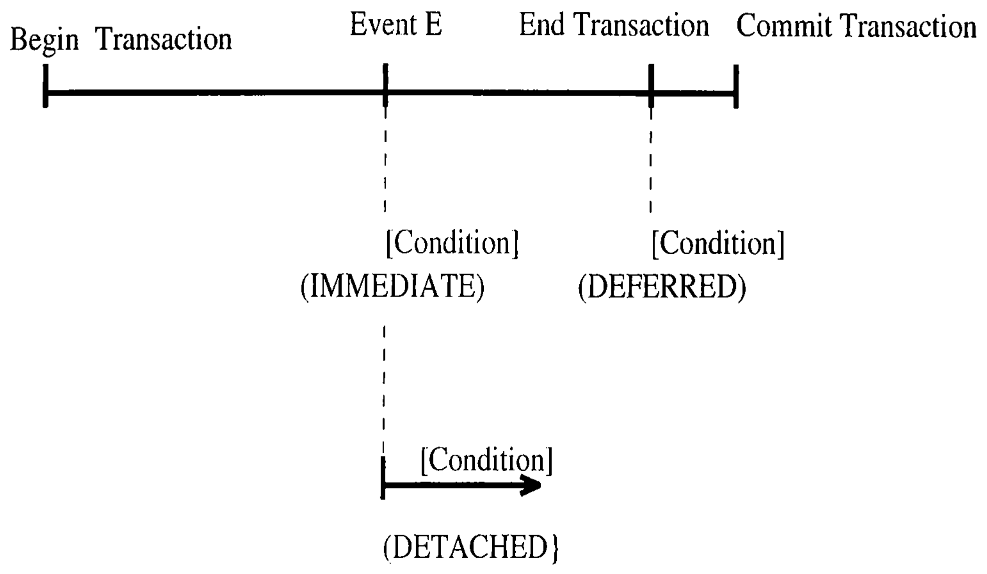

Figure 2.1: Basic Coupling Modes Illustrated

that caused the event to be raised. If the action part is specified to be executed in immediate mode then it is executed immediately after the evaluation of the condition. In case the condition is specified to be in *deferred* mode, its evaluation is delayed until the commit point of the transaction, and similarly if the action is in deferred mode relative to the condition, again it is executed right before the transaction commits. Finally, in *detached mode*, condition is evaluated or action is executed in a separate transaction. Detached mode can further be classified into four subcategories: *detached coupling*, *detached causally dependent coupling*, *sequential causally dependent coupling*, and *exclusive causally dependent coupling* [Buc94]. In *detached coupling* there is no dependency between the triggering and triggered transactions. In *detached causally dependent coupling*, the triggered transaction can commit only if the triggering transaction commits. In *sequential causally dependent coupling*, the triggered transaction can start executing only if the triggering transaction commits. Finally, in *exclusive causally dependent coupling*, triggered transaction commits only if the triggering transaction fails.

There is a special technique used for the execution of transactions fired in deferred mode [HLM88]. Deferred transactions are executed in cycles. In cycle-0, deferred transactions that have been fired up to that point are executed. Transactions spawned during cycle-0 in immediate mode are executed as a linear extension of their parents as usual. Execution of the deferred transactions that are fired during cycle-0 by another deferred transaction is postponed to the next cycle, which is cycle-1. Again deferred transactions that are fired in cycle-1 are postponed to the next cycle, which is cycle-2, and so on. This process continues until there are no deferred transactions left.

2.2 Nested Transactions for Rule Execution

Nested transactions are considered to be suitable for rule execution in ADBMSs. In the following subsections, we will discuss nested transactions together with the concurrency control and recovery issues.

2.2.1 An Introduction to Nested Transactions

Traditional transactions have only one branch of execution. In the Nested Transaction Model, transactions can have multiple branches of execution. A nested transaction may either consist of a set of primitive actions or other nested transactions; i.e., it is recursive. Nested transactions form hierarchies which can be represented as trees and standard tree notions like parent, child, ancestor, descendant, superior, and inferior also apply to them. The root of the tree is called a root or top-level transaction. The root may have one or more children, similarly children of the root may also have other children. By dividing transactions into smaller granules, we localize the failures into subtransactions. Subtransactions can abort independently without causing the abortion of the whole transaction hierarchy. When a transaction aborts, all of its descendants are also aborted, but other transactions are not affected. Nested transactions are also very useful in terms of system modularity. If we consider a transaction hierarchy as a big module, its subtransactions may be designed and implemented independently as submodules, also providing encapsulation and security [HR93].

2.2.2 Concurrency Control and Recovery Issues in Nested Transactions

Using nested transactions we can exploit the parallelism among subtransactions since subtransactions can be executed in parallel. There can be four different kinds of parallelism:

1. only sibling
2. only parent-child
3. parent-child and sibling
4. no parallelism (i.e., sequential execution)

In the first case, where only sibling parallelism is allowed, parent stops its execution while its children are running concurrently. In the second case, only parent-child parallelism is allowed where parent and child run concurrently while the other children wait. In the third case, all transactions in the hierarchy can run in parallel. In the fourth case, we have no parallelism at all (i.e., transactions in the hierarchy are executed sequentially) [HR93]. In our model, we will assume parent-child and sibling parallelism since it provides us with the most flexible model of parallelism.

When transactions are executed concurrently, serializability is used as the correctness criterion, and it is ensured by two-phase locking. A child transaction can potentially access any object in the database. When a subtransaction commits, the objects modified by it are delegated to its parent transaction. We used a locking protocol, which is described in Chapter 3, for concurrency control in nested transaction execution.

In nested transactions, ACID properties (i.e., atomicity, consistency, isolation, and durability) are valid for top-level transactions, but only a subset of them holds for subtransactions [HR93]. A subtransaction may commit or abort independent of other transactions. Aborting a subtransaction does not affect other transactions outside of its hierarchy, hence they protect the outside world from internal failures. If we had packed all subtransactions into one big flat transaction then we would have to abort the whole transaction.

Recovery of nested transactions is similar to the recovery of flat transactions. Standard recovery algorithms like versioning or log-based recovery can be used. Log-based recovery for nested transactions is discussed in [Mos87] and [RM89]. [RM89] introduces a model called ARIES/NT and this has several advantages over the recovery model provided in [Mos87] . The biggest drawback of the recovery model of [Mos87] is that it does not use Compensation Log Records (CLRs) which are necessary for performance reasons. A detailed description of CLRs is provided in [RM89]. Implementation of our execution model has been built on OpenOODB which uses EXODUS as storage manager whose recovery component is implemented based on ARIES [MHL$^+$92], and ARIES/NT is provided for nested transactions as an extension to ARIES. Therefore, ARIES/NT is the most suitable recovery scheme that can be adopted to our transaction execution model.

Chapter 3

Execution Model

Our rule execution model is based on the nested transaction model. The nested transaction model implicitly assumes that the subtransactions are spawned in immediate mode. In our execution model, transactions may spawn subtransactions in any coupling mode specified by the system. Each rule is encapsulated in a transaction. When a rule r_1 fires another rule r_2, then depending on the coupling mode, r_2 is encapsulated in another (sub)transaction and executed in the specified coupling mode. If the coupling mode is immediate or deferred, then r_2 is executed as a subtransaction of r_1. If the coupling mode is one of sequential causally dependent, detached causally dependent or, exclusive causally dependent, then r_2 is executed as a top-level transaction. The overall structure of the currently executing rules in the system forms a forest consisting of trees whose roots are the rules fired in one of the detached coupling modes. As stated earlier, both parent-child and sibling parallelism are allowed which provides us with the maximum concurrency among subtransactions. Top-Level transactions are executed in parallel. All nested transaction semantics apply among the individual rules in the nested transaction tree. Abort and commit dependencies among the top-level transactions are enforced by the transaction manager.

The concurrency control algorithm used in our execution model is based on the notion of nested concurrency control. Harder and Rothermel [HR93] have extended Moss's nested transaction model to contain downward as well as upward inheritance of locks. We have employed in our model the locking protocol provided in [HR93]. The protocol is composed of the following locking rules:

- Rule 1: Transaction T may acquire a lock in mode M or upgrade a lock it holds to mode M if
  - no other transaction holds the lock in a mode that conflicts with M, and
  - all transactions that retain the lock in a mode conflicting with M are ancestors of T.

  A transaction holds a lock on an object if it has the right to access the locked object in the requested mode. In contrast, a transactions retains a lock on an object to control the access of the transactions outside the hierarchy of the retainer and cannot be accessed by the transaction retaining the lock.

- Rule 2: When subtransaction T commits, the parent of T inherits T's locks (held and retained). After that, the parent retains the locks in the same mode as T held or retained them before.

- Rule 3: When a top-level transaction commits, it releases all locks it holds or retains.

- Rule 4: When a transaction aborts, it releases all locks it holds or retains. If any of its superiors hold or retain any of these locks, they continue to do so.

- Rule 5: Transaction T, holding a lock in mode M, can downgrade the lock to a less restrictive mode, M′. After downgrading the lock, T retains it in mode M.

These locking rules can be used with different types of coupling modes. A transaction spawned in detached causally dependent mode should be able to use its parent's locks in the same way as a subtransaction spawned in immediate or deferred mode. Since the transaction spawned in detached causally dependent mode should abort if its parent aborts, it can use its parent's locks without causing any problem in the recovery.

Both shared and exclusive lock modes are available to transactions in our execution model.

Active database recovery is still an open research area. There are only a few papers on active database recovery which mainly focus on the recovery of events (e.g., [HEKX94], [Zuk95]).

Active database recovery in general deals with :

- Recovery of events.
- Recovery of aborted rules. Possible solutions to the recovery of aborted rules are listed in [HEKX94] as:
  - ignore the aborted transaction,
  - abort the triggering transaction,
  - retry the triggered transaction or start a different one,
  - reset the triggering transaction to the point of the occurrence of the event that caused the rule to be triggered.

The above recovery options may be left to the user decision, or they may be handled automatically depending on the implementation. The most reasonable approach would be to let the user specify the recovery mode of the rule during the rule definition and make the recovery mode an attribute of the rule.

The execution model described in this thesis does not include recovery. As we discussed in Section 2.2.2, ARIES/NT [RM89] can be adopted as a recovery model for nested rule execution without significant modifications. Recovery of transactions spawned in immediate or deferred mode can be handled as described in [RM89]. Abort dependencies must be considered during rollbacks. Among the detached coupling modes, only the detached causally dependent mode requires some extension to the model of ARIES/NT. Assume that a transaction A spawns a transaction B in causally dependent mode. It means that abortion of B should be succeeded by the abortion of A which should be enforced by the recovery model. Other detached coupling modes do not require any extension to the recovery model. Transactions spawned in one of those modes are treated as top level transactions, and their recovery is performed by following the top-level transaction abortion steps.

3.1 ACTA

ACTA is a transaction framework that can be used to formally describe extended transaction models [CR94]. Using ACTA, we can specify the interactions and dependencies between the transactions in a model. ACTA characterizes the semantics of interactions (1) in terms of different types of dependencies between transactions (e.g., commit dependency and abort dependency) and (2) in terms of transactions effects on objects (their state and concurrency status, i.e., synchronization state) [CR94]. Effects of transactions on objects are specified using two sets associated with each transaction: a *view set* which contains the state of objects visible to that transaction and a *conflict set* which contains operations for which conflicts need to be considered. ACTA framework consists of four basic blocks which are *history*, *dependencies* between transactions, *view* and *conflict sets* of transactions, and finally *delegation*. *History* represents the concurrent execution of a set of transactions and contains all the events invoked by those transactions, also indicating the partial order in which these events occur. ACTA captures both of the effects of transactions on other transactions and their effects on objects through constraints on histories. Transaction models are defined by a set of axioms. These axioms are invariant assertions about the histories of transactions belonging to the particular model. The whole history is denoted by H, and H_{ct} denotes the current history. We may also project the history to obtain a subhistory that satisfies some criterion. Invocation of a general event ϵ by transaction t is denoted by ϵ_t. There are three possibilities that can affect the occurrence of an event:

1. an event ϵ can occur only after the occurrence of another event ϵ' (denoted as $\epsilon' \to \epsilon$)

2. an event ϵ can occur only if a condition c is true (denoted as $c \Rightarrow \epsilon$)

3. a condition c can require the occurrence of an event ϵ (denoted as $\epsilon \Rightarrow c$).

$H^{(ob)}$ is the projection of the history H with respect to the object ob. Two operations p and q conflict in a state produced by $H^{(ob)}$, denoted by $conflict(H^{(ob)}, p, q)$, if and only if,

$$\begin{aligned}
&(state(H^{(ob)} \circ p, q) \neq state(H^{(ob)} \circ q, p)) \\
&\vee\ (return(H^{(ob)}, q) \neq return(H^{(ob)} \circ p, q)) \\
&\vee\ (return(H^{(ob)}, p) \neq return(H^{(ob)} \circ q, p))
\end{aligned}$$

where $state(s, o)$ represents the state produced after the operation o is applied to the state s, and $return(s, o)$ represents the output produced when the operation o is applied to state s, and $\circ$ denotes function composition.

There are two types of events in ACTA:

- Object Events: Invocation of an operation on an object is called an object event. An invocation of an operation p on an object ob by transaction t is shown by $p_t[ob]$ and OE_t is the set of object events that can be invoked by transaction t. Effects of $p_t[ob]$ are made permanent by invoking a commit operation on this object, which corresponds to the event $Commit[p_t[ob]]$. When we want to discard the operation performed on an object, we abort it, and the corresponding event is $Abort[p_t[ob]]$.

- Significant Events: Invocation of a transaction management primitive like begin, abort, spawn, or commit is called a significant event. SE_t is the set of significant events corresponding to transaction t. Events related to the initiation of a transaction are called *initiation events* and are denoted by IE_t. Events that are related to the termination of a transaction are called *termination events* and are denoted by TE_t.

$Delegate_{t_i}[t_j, p_{t_i}[ob]]$ denotes that transaction t_i delegated the responsibility of committing or aborting the operation $p_{t_i}[ob]$ to transaction t_j. A set of operations may be delegated by $Delegate_{t_i}[t_j, DelegateSet]$. Initially, the responsibility of committing or aborting an operation belongs to the transaction that invoked the operation, unless it is delegated to another transaction.

$ResponsibleTr(p_{t_i}[ob])$ identifies the transaction that is responsible for committing or aborting the operation $p_{t_i}[ob]$ with respect to the current history.

Now we can formally define the access set of a transaction t by:

$$AccessSet_t = \{p_{t_i}[ob] | ResponsibleTr(p_{t_i}[ob]) = t)\}$$

An object ob behaves correctly if and only if

$$\forall t_i, t_j \in T, t_i \neq t_j, \forall p, q(return\_value\_dependent(p, q) \\ \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])) \wedge \neg((Commit[p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \\ \vee (Abort[p_{t_i}[ob]] \rightarrow q_{t_j}[ob])) \Rightarrow ((Abort[p_{t_i}[ob] \in H^{(ob)}) \Rightarrow \\ (Abort[q_{t_j}[ob]] \in H^{(ob)}))$$

where $return\_value\_dependent(H^{(ob)}p, q)$ is true if $conflict(H^{(ob)}, p, q)$ is true and $return(H^{(ob)} \circ p, q) \neq return(H^{(ob)}, q)$

An object ob behaves *serializably* if and only if

1. $\forall t_i, t_j \in T_{comm}, t_i \neq t_j$
   $(t_i b_{ob} t_j) \Leftrightarrow \exists p, q(conflict(p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob])))$.
2. $\forall t \in T_{comm} \Rightarrow \neg(t b^*_{ob} t)$

where T_{com} is the set of committed transactions; b_{ob} is a binary relationship that occurs due to an access to the object ob by the transactions that take place in the binary relationship; and b^*_{ob} is the closure of b_{ob}.

An object ob is *atomic* if ob behaves *correctly* and *serializably*.

Fundamental axioms of transactions:

1. $\forall \alpha \in IE_t(\alpha \in H^t) \Rightarrow \neg \exists \beta \in IE_t(\alpha \rightarrow \beta)$
   A transaction cannot be initiated by two different events.

2. $\forall \delta \in TE_t \; \exists \alpha \in IE_t(\delta \in H^t) \Rightarrow (\alpha \rightarrow \delta)$
   If a transaction has terminated, it must have been previously initiated.

3. $\forall \gamma \in TE_t(\gamma \in H^t) \Rightarrow \neg \exists \delta \in TE_t(\gamma \rightarrow \delta)$
   A transaction cannot be terminated by two different termination events.

4. $\forall ob \forall p(p_t[ob] \in H) \Rightarrow ((\exists \alpha \in IE_t(\alpha \rightarrow p_t[ob])) \wedge (\exists \gamma \in TE_t(p_t[ob] \rightarrow \gamma)))$
   Only in-progress transactions can invoke operations on objects.

Here we list a subset of standard dependencies between transactions that are defined in ACTA which we have used for specifying our execution model for Active Databases.

- Commit Dependency (denoted as $t_j \; CD \; t_i$). If transactions t_i and t_j both commit then t_i should commit before t_j. This can be shown axiomatically as:

  $Commit_{t_j} \in H \Rightarrow (Commit_{t_i} \in H \Rightarrow (Commit_{t_i} \rightarrow Commit_{t_j}))$.

- Abort Dependency (denoted as $t_j \; AD \; t_i$). If t_i aborts then t_j should also abort:

  $Abort_{t_i} \in H \Rightarrow Abort_{t_j} \in H$

- Weak-Abort Dependency (denoted as $t_j \; WD \; t_i$). If t_i aborts and t_j has not yet committed, then t_j aborts:

  $Abort_{t_i} \in H \Rightarrow (\neg(Commit_{t_j} \rightarrow Abort_{t_i}) \Rightarrow (Abort_{t_j} \in H))$

- *Exclusion Dependency* is denoted by $t_j \; ED \; t_i$, and ensures that if t_i commits , then t_j must abort. We can state this formally as:
  $Commit_{t_i} \in H \Rightarrow Abort_{t_j} \in H$.

These dependencies may be the result of the structural properties of transactions. For example, in nested transactions child transactions are related to their parent by commit and weak-abort dependencies.

Axiomatic definition of the standard nested transaction model is provided in [CR94]. Since we are utilizing the extended model of nested transactions described in [HR93], which is different from the standard nested transaction model of [Mos85], we need to modify the axiomatic definitions provided for nested transactions in [CR94]. First of all, the standard ACTA term $ResponsibleTr$ is separated into two notions, namely $Responsible\_retainTr$ and $Responsible\_selfTr$. With respect to this modification, $Responsible\_self Tr(p_{t_i}[ob])$ identifies the transaction which actually invoked an operation on the object ob. The transaction identified by $Responsible\_selfTr(p_{t_i}[ob])$ is also responsible for the commit or abort of this operation. $Responsible\_retainTr(p_{t_i}[ob])$ identifies the transaction to which the responsibility of committing or aborting this operation is delegated.

Since our notion of responsible transaction is different, the semantics of delegation should also be modified. According to this modification,

> $Delegate_{t_i}[t_j, p_{t_i}[ob]]$ denotes that transaction t_i delegated the retain or self responsibility of committing or aborting the operation $p_{t_i}[ob]]$ to transaction t_j as retain responsibility.
>
> Finally, the access set of a transaction is modified as:
> $AccessSet_t = \{p_{t_i}[ob] | Responsible\_selfTr(p_{t_i}[ob]) = t$
> $\vee\ \ Responsible\_retainTr(p_{t_i}[ob]) = t\}$

In the next section, we provide the axiomatic definition of the extended parallel nested transaction model. These definitions have been obtained by modifying the axiomatic definitions of standard nested transactions provided in [CR94].

3.2 Axiomatic Definition of Parallel Nested Transactions in ACTA

Assume that t_0 is the root transaction, t_p is a root or a subtransaction, and t_c is a subtransaction of t_p. $Ancestors(t)$ is the set of all ancestors of transaction t, $Descendants(t)$ is the set of all descendants of transaction t, and $Parent(t)$ contains the parent of t.

1. $SE_{t_0} = \{Begin, Spawn, Commit, Abort\}$
2. $IE_{t_0} = \{Begin\}$
3. $TE_{t_0} = \{Commit, Abort\}$
4. $SE_{t_c} = \{Spawn, Commit, Abort\}$
5. $IE_{t_c} = \{Spawn\}$
6. $TE_{t_c} = \{Commit, Abort\}$
7. t_p satisfies the fundamental axioms of transactions that are listed in the preceding section.
8. $View_{t_p} = H_{ct}$
   That is, t_p sees the current state of objects in the database.
9. $ConflictSet_{t_0} = \{p_t[ob] | Responsible\_selfTr(p_t[ob]) \neq t_0, Inprogress(p_t[ob])\}$
   Conflict set of t_0 consists of all operations performed by different transactions for which it is not self-responsible (i.e., t_0 did not actually invoked the operation).
10. $\forall ob \, \exists p \, p_{t_p}[ob] \in H \Rightarrow (ob \ is \ atomic)$
    All objects on which t_p invokes an operation are atomic objects.
11. $Commit_{t_p} \in H \Rightarrow \neg(t_p b_N^* t_p)$
    Transaction t_p can commit only if it is not part of a cycle of b relations that are results of conflicting operations.

12. $\exists ob, p, t\,(Commit_{t_p}[p_t[ob]] \in H \Rightarrow Commit_{t_p} \in H \wedge Parent(t_p) = \phi)$
If an operation p invoked on an object ob is committed by transaction t_p, then t_p should also commit and it should be a top-level transaction.

13. $(Commit_{t_p} \in H \wedge Parent(t_p) = \phi) \Rightarrow \forall ob, p, t(p_t[ob] \in AccessSet_{t_p}$
$\Rightarrow Commit_t[p_t[ob]] \in H)$
If a top-level transaction commits then all the operations for which it is responsible must also be committed.

14. $\exists ob, p, t\,(Abort_{t_p}[p_t[ob]] \in H \Rightarrow Abort_{t_p} \in H)$
If an operation p invoked on an object ob in transaction t is aborted by transaction t_p, then t_p must also abort.

15. $Abort_{t_p} \in H \Rightarrow \forall ob, p, t(p_t[ob] \in AccessSet_{t_p} \Rightarrow Abort_{t_p}[p_t[ob]] \in H)$
If t_p aborts then all the operations for which it is responsible must abort.

16. $Begin_{t_p} \in H \Rightarrow (Parent(t_p) = \phi \wedge Ancestor(t_p) = \phi)$
Begin operation implies that a top-level transaction starts its execution.

17. $ConflictSet_{t_c} = \{p_t[ob] | Responsible\_selfTr(p_t[ob]) \neq t_c, Inprogress(p_t[ob])\}$
Conflict set of a child transaction t_c consists of those operations for which t_c is not self responsible, since the operations on the object ob for which t_c has the retain responsibility may be conflicting with another transaction which is not an ancestor of t_c. This is due to the fact that subtransactions are executing in parallel with the parent transaction.

18. $Spawn_{t_p}[t_c] \in H \Rightarrow Parent(t_c) = t_p$
If transaction t_c is spawned by transaction t_p then t_p is the parent of t_c.

19. $Spawn_{t_p}[t_c] \in H \Rightarrow (t_c WD t_p) \wedge (t_p CD t_c)$
If transaction t_c is spawned by transaction t_p then t_p cannot commit until t_c terminates, and if t_p aborts then t_c must also abort.

20. $Commit_{t_c} \in H \Leftrightarrow Delegate_{t_c}[Parent(t_c), AccessSet_{t_c}] \in H$
If a child transaction t_c commits, then it should delegate the objects in its access set to its parent.

21. $\forall t \in Descendants(t_p) \forall ob, p, q(p_t[ob] \rightarrow q_{t_p}[ob]) Conflict(p_t[ob], q_{t_p}[ob])$
$\Rightarrow \exists t_c\,((Delegate_{t_c}[t_p, AccessSet_{t_c}] \rightarrow q_{t_p}[ob]) \wedge p_t[ob] \in AccessSet_{t_c})$

Given a transaction t and its ancestor t_p and operations p and q, t_p can invoke q after t invokes p if t_p is responsible for the operation p.

22. $(Ancestor(t_c) = Ancestor(t_p) \cup \{t_p\}) \wedge \forall t(t_p \in Descendants(t)$
$\Rightarrow t_c \in Descendants(t))$
Ancestor set of t_c consists of its parent plus ancestors of its parent, and for all transactions t of which t_p is a descendant, t_c is also a descendant of t.

3.3 A Formal Model for Rule Execution in Active Databases using ACTA

A formal specification of rule execution in Active Databases using ACTA can be provided without significant changes to the standard ACTA primitives. Instead of using a single *Spawn* primitive, we add the primitives *Spawn_Imm*, *Spawn_Def*, *Spawn_Detached*, *Spawn_Caus*, *Spawn_Seq*, and *Spawn_Exc*, which specify the coupling modes in which the subtransactions are spawned. These new primitives will be explained in Section 3.3.1. All coupling modes except the *deferred mode* and *sequential causally dependent mode* can be specified easily using the usual dependencies of ACTA. For the deferred mode, we need to specify a *cycling execution method*, which can be stated as follows:

> Deferred transactions are executed in cycles at the end, but just before the commit of the transaction that spawned them. Cycling execution can start only in a top-level transaction or a subtransaction spawned in immediate mode since deferred subtransactions spawned by another deferred transaction are executed in the next cycle after the commitment of their parent. Subtransactions spawned in immediate mode are executed immediately, which deviates from the standard deferred execution specification.

3.3.1 Coupling Modes

The coupling modes we considered in our execution model are listed below:

- *immediate mode*, which has the same semantics as the creation of a subtransaction in standard nested transaction model. Spawning of an immediate subtransaction is denoted by the primitive *Spawn_Imm*.

- *detached mode*, which has the same semantics as the creation of top-level transactions in the standard nested transaction model. There are no dependencies between the spawning and spawned transaction. Spawning of a detached transaction is denoted by the primitive *Spawn_Detached*.

- *detached causally dependent mode*, in which spawned transaction aborts if the parent aborts, so there is an abort dependency between the spawning transaction and spawned transaction. Spawning of a transaction in this mode is specified by the primitive *Spawn_Caus*.

- *sequential causally dependent mode*, which specifies that a child transaction cannot start its execution until its parent commits. This can be enforced by a *Sequential_Dependency*(SQD) which is provided as an extension to the ACTA dependency set and can be stated formally as:

  $$t_i SQD t_j \Leftrightarrow ((Begin_{t_j} \in H) \Rightarrow (Commit_{t_i} \rightarrow Begin_{t_j}))$$

  The primitive *Spawn_Seq* indicates that a subtransaction is spawned in this mode.

- *exclusive causally dependent mode*, which is denoted by the primitive *Spawn_Exc* and it ensures that the spawned transaction commits only if the spawning transaction aborts. This can be enforced by using a standard ACTA dependency, namely the *Exclusion Dependency* between the spawning and spawned transactions.

- *deferred mode*, which is denoted by the primitive *Spawn_Def*, and a bit more effort is required to specify it in ACTA framework due to the cycling execution method. Assume that t_0 is a top-level transaction, t_p is a top-level or subtransaction, and t_c is a child transaction spawned by t_p in deferred mode.

Case 1: t_p is a top-level transaction or a subtransaction spawned in immediate mode, i.e., t_c is going to be executed in cycle-0. In this case t_c is executed just before the commit of t_p after all other operations of t_p are completed, i.e., all operations of t_p precede all operations of t_c.
Case 2: t_p is a transaction spawned in deferred mode. This means that t_c is spawned during a cycle. Then, every operation performed by t_c should succeed all the operations of t_p and the operations of siblings of t_p that are spawned in deferred mode (i.e., executed in the same cycle).

Discussion

There is an ambiguity in the method described in [HLM88] for the cycling execution of rules fired in deferred coupling mode. If a rule is fired in deferred mode by a transaction during the execution of a cycle, it is executed in the next cycle; but if a rule is fired in deferred mode by a transaction which has been fired in immediate mode then the fate of this transaction, i.e., whether it will be deferred to the next cycle or it will be executed in another execution cycle is left unspecified. We chose to execute these kinds of rules in another cycle before the commit point of the immediate rule.

In our execution model, we consider the coupling modes between the event and condition, and also the condition and action. We can give the option of defining the coupling mode between the condition and action to the user, where the user can select immediate or sequential causally dependent coupling mode. Other coupling modes would not be meaningful due to the relation between the condition and action. In immediate mode, condition evaluation is followed by execution of the action only if the condition evaluates to true. In sequential causally dependent mode, action execution starts before condition evaluation is completed. This improves the concurrency in a system in case there exist abundant resources in the system, which is a reasonable assumption due to the continuous decrease in the prices of system resources. The transaction in which the action is executed can commit only if the condition commits and returns true. The performance impact of the sequential causally dependent coupling mode between condition and action needs further research and testing in a real system.

3.3.2 A Formal Model Using ACTA

Assume that the definitions provided at the beginning of Section 3.2 for the following notations also hold in this section: t_0, t_p, t_c, $Ancestors(t)$, $Decendants(t)$, $Parent(t)$.

Notice that, as one deviation from the standard nested transaction model, there are various spawn events in axioms (1),(4), and (5) corresponding to different coupling modes. Some of the axioms used for the nested transaction model directly apply to our execution model. For example, axioms (7) through (17) which define the semantics of a top-level or a subtransaction which spawns another transaction are the same for both models, therefore we did not include their explanations here. Readers who need more information about those axioms are referred to Section 3.2.

1. $SE_{t_0} = \{Begin, Spawn\_Imm, Spawn\_Def, Spawn\_Detach, Spawn\_Caus, Spawn\_Seq, Spawn\_Exc, Commit, Abort\}$
2. $IE_{t_0} = \{Begin\}$
3. $TE_{t_0} = \{Commit,\ Abort\}$
4. $SE_{t_c} = \{Spawn\_Imm, Spawn\_Def, Spawn\_Detach, Spawn\_Caus, Spawn\_Seq, Spawn\_Exc, Commit, Abort\}$
5. $IE_{t_c} = \{Spawn\_Imm, Spawn\_Def, Spawn\_Detach, Spawn\_Caus, Spawn\_Seq, Spawn\_Exc\}$
6. $TE_{t_c} = \{Commit, Abort\}$
7. t_p satisfies the fundamental axioms of transactions that are listed in Section 3.1.
8. $View_{t_p} = H_{ct}$

9. $ConflictSet_{t_0} = \{p_t[ob] | responsible\_selfTr(p_t[ob]) \neq t_0, Inprogress(p_t[ob])\}$

10. $\forall ob \exists p p_{t_p}[ob] \in H \Rightarrow (ob\ is\ atomic)$

11. $Commit_{t_p} \in H \Rightarrow \neg(t_p b_N^* t_p)$

12. $\exists ob, p, t Commit_{t_p}[p_t[ob]] \in H \Rightarrow Commit_{t_p} \in H \wedge Parent(t_p) = \phi$

13. $Commit_{t_p} \in H \wedge Parent(t_p) = \phi \Rightarrow (\forall ob, p, t(p_t[ob] \in AccessSet_{t_p}$
$\Rightarrow Commit_t[p_t[ob]] \in H))$

14. $\exists ob, p, t\, Abort_{t_p}[p_t[ob]] \in H \Rightarrow Abort_{t_p} \in H$

15. $Abort_{t_p} \in H \Rightarrow (\forall ob, p, t(p_t[ob] \in AccessSet_{t_p} \Rightarrow Abort_{t_p}[p_t[ob]] \in H))$

16. $Begin_{t_p} \in H \Rightarrow Parent(t_p) = \phi \wedge Ancestor(t_p) = \phi$

17. $ConflictSet_{t_c} = \{p_t[ob] | Responsible\_selfTr(p_t[ob]) \neq t_c, Inprogress(p_t[ob])\}$

18. $(Spawn\_Imm_{t_p}[t_c] \in H \vee Spawn\_def_{t_p}[t_c] \in H) \Rightarrow Parent(t_c) = t_p$
If a transaction t_p spawns a child transaction t_c in immediate or deferred mode then t_p is the parent of t_c.

19. $(Spawn\_Imm_{t_p}[t_c] \in H \vee Spawn\_def_{t_p}[t_c] \in H) \Rightarrow (t_c WD t_p) \wedge (t_p CD t_c)$
If a transaction t_c is spawned in immediate or deferred mode by a transaction t_p, then t_c aborts when t_p aborts and t_p cannot commit until t_c terminates.

20. $(Spawn\_Caus_{t_p}[t_c] \in H \vee Spawn\_Detach_{t_p}[t_c] \in H$
$\vee Spawn\_Seq_{t_p}[t_c] \in H \vee Spawn\_Exc_{t_p}[t_c] \in H) \Rightarrow Parent(t_c) = \phi$
$\wedge Ancesstor(t_c) = \phi$
A transaction spawned with detached, detached causally dependent, sequential causally dependent, or exclusive causally dependent mode is a top-level transaction, therefore has no parent or ancestor.

21. $Spawn\_Caus_{t_p}[t_c] \in H \Rightarrow t_c AD t_p$
If a subtransaction t_c is spawned in causally dependent mode by transaction t_p then t_c must abort if t_p aborts.

22. $Spawn\_Seq_{t_p}[t_c] \in H \Rightarrow t_c SQD t_p$
If a subtransaction t_c is spawned in sequential causally dependent mode by transaction t_p then t_c can start its execution only if t_p commits.

23. $Spawn\_Exc_{t_p}[t_c] \in H \Rightarrow t_c EDt_p$
    If a subtransaction t_c is spawned in exclusive causally dependent mode by transaction t_p then t_c can start its execution only if t_p aborts.

24. $(Spawn\_Imm_{t_p}[t_c] \vee Spawn\_def_{t_p}[t_c]) \in H \wedge Commit_{t_c} \in H$
    $\Leftrightarrow Delegate_{t_c}[Parent(t_c), AccessSet_{t_c}] \in H$
    If a subtransaction t_c is spawned in immediate or deferred mode by transaction t_p then t_c must delegate all the operations in its access set to its parent t_p.

25. $\forall t, ob, p, q(t \in Descendants(t_p) \wedge (p_t[ob] \rightarrow q_{t_p}[ob]) \wedge Conflict(p_t[ob], q_{t_p}[ob])$
    $\Rightarrow \exists t_c((Delegate_{t_c}[t_p, AccessSet_{t_c}] \rightarrow q_{t_p}[ob] \wedge p_t[ob] \in AccessSet_{t_c})))$
    Given a transaction t and its ancestor t_p and operations p and q, t_p can invoke q after t invokes p if t_p is responsible for the operation p.

26. $(Spawn\_Imm_{t_p}[t_c] \vee Spawn\_Def_{t_p}[t_c]) \in H$
    $\Leftrightarrow (Ancestor(t_c) = Ancestor(t_p) \cup \{t_p\}) \wedge \forall t(t_p \in Descendants(t)$
    $\Rightarrow t_c \in Descendants(t))$
    Ancestor set of a transaction spawned in immediate or deferred mode is defined similarly as that with the standard nested transactions.

27. $(Spawn\_Def_{t_p}[t_c] \in H \wedge (Parent(t_p) = \phi \vee \exists t(Spawn\_Imm_t[t_p] \in H))$
    $\Rightarrow \forall p, ob_1, q, ob_2(p \neq Commit \wedge p_{t_p}[ob_1] \in H \wedge q_{t_c}[ob_2] \in H$
    $\Rightarrow (p_{t_p}[ob1] \rightarrow p_{t_c}[ob_2])))$
    This axiom corresponds to Case-1 of the deferred coupling mode execution described in Section 3.3.1.

28. $(Spawn\_Def_{t_p}[t_c] \in H \wedge \exists t(Spawn\_Def_t[t_p] \in H)$
    $\Rightarrow \forall p, q, r, ob_1, ob_2, ob_3, t_1, t_2(p_{t_p}[ob1] \in H \wedge q_{t_c}[ob2] \in H$
    $\wedge\ Spawn\_Def_t[t_2] \in H \Rightarrow (p_{t_p}[ob1] \rightarrow q_{t_c}[ob2] \wedge r_{t_2}[ob3] \rightarrow q_{t_c}[ob3])$
    This axiom corresponds to Case-2 of the deferred mode execution.

Chapter 4

Implementation of Nested Transactions

4.1 Previous Work

Various implementations of the nested transaction model have been provided to date. One such implementation has been performed on the Eden Resource Management System (ERMS) [PN87]. In ERMS, transaction managers are composed hierarchically, i.e., for each subtransaction there is a corresponding transaction manager. For ensuring the serializability, 2-phase locking is used, and a version-based recovery is used for the recovery of sub-transactions. In [DGRV95], the implementation described focuses on nested transactions for client workstations of an OODBMS.

Nested transactions have also been implemented for supporting parallelism in engineering databases [HPS92]. That implementation of nested transactions supports both parent-child and sibling parallelism as in our implementation.

Nested transactions have been implemented for parallel rule execution of Sentinel ADBMS [CAM93] by modifying a prototype OODBMS called Zeitgeist [PP91] without considering recovery of nested transactions [Bad93]. Only sibling parallelism is assumed in that implementation; i.e., parent transaction is suspended while its children are executing in parallel. Deadlock detection is

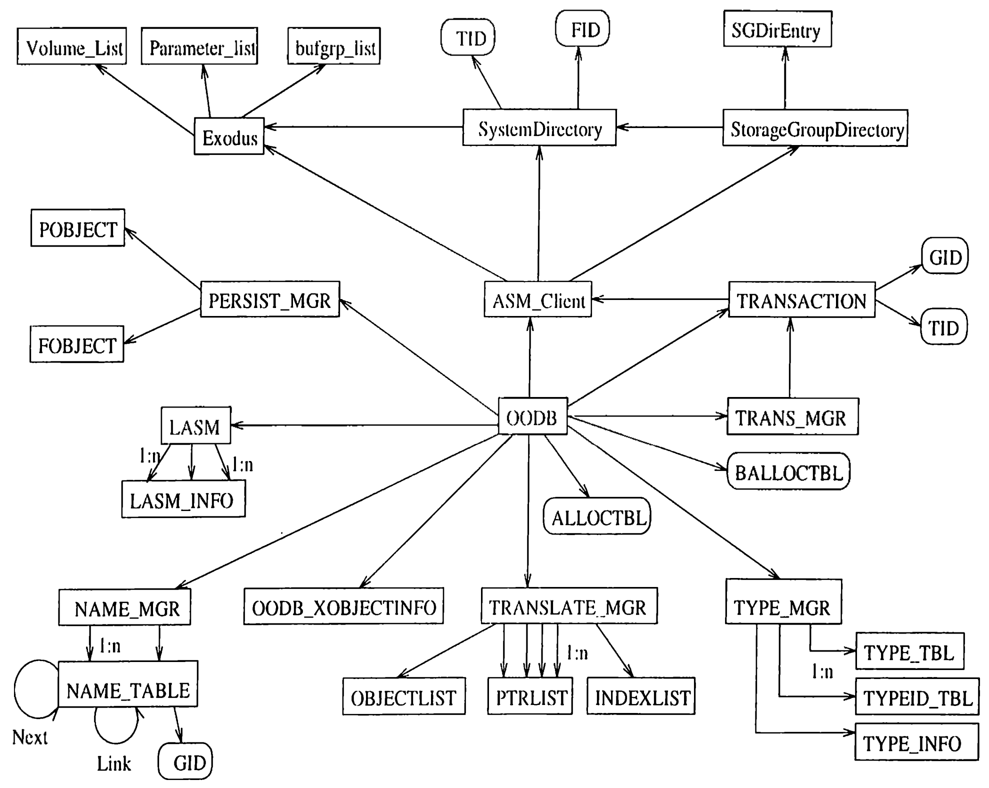

Figure 4.1: OpenOODB Object Relationship Diagram

performed by modifying the transaction manager of Zeitgeist. An analysis of the nested transaction locking protocol is made using the concept of spheres of control. Only the immediate coupling is supported in the implementation of nested transactions.

4.2 Implementation

We implement nested transactions by extending the flat-transaction semantics of OpenOODB [WBT92]. OpenOODB is an open object oriented database management system that can be extended by special constructs called sentries. In our implementation, a component architecture method is used instead of sentries, i.e., a new component is added without significantly modifying the existing ones. Our first task was to construct the object relationships of OpenOODB by examining the class declarations. In Figure 4.1 the whole OpenOODB object relationships diagram is given.

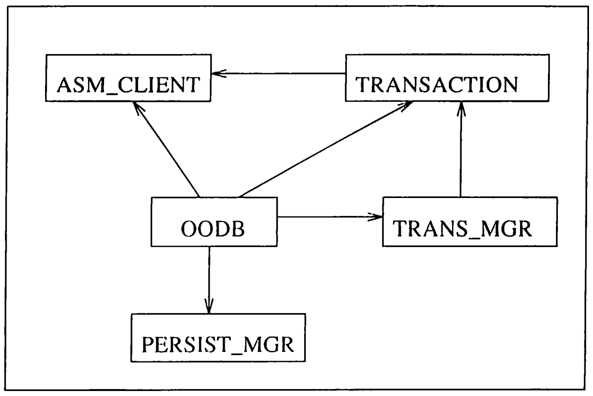

Figure 4.2: Related Object Relationships

Among the components illustrated in this diagram, the ones that need to be considered for our implementation are isolated. These isolated components are shown in Figure 4.2. In this figure, we see that the main OpenOODB object *OODB* has a pointer to each of four objects namely *PERSIST_MGR*, *TRANS_MGR*, *TRANSACTION*, and *ASM_CLIENT* which means that whenever an instance of an object of type *OODB* is created, its constructor creates instances of *PERSIST_MGR*, *TRANS_MGR*, *TRANSACTION*, and *ASM_CLIENT* objects. Furthermore, the constructor of *TRANS_MGR* object creates an instance of *TRANSACTION* object which is also used directly by *OODB*.

To give a flavor of how a transaction is started and objects are fetched from the database, we give a sample application of OpenOODB in Figure 4.3.

As can be seen from the figure, an OpenOODB main object *p_oodb* is created which provides us with an interface to OpenOODB. A transaction is started by using *p_oodb* $\rightarrow$ *beginTransaction* and committed by *p_oodb* $\rightarrow$ *commitTransaction*. Abortion of a transaction is achieved by *p_oodb* $\rightarrow$ *abortTransaction*. Objects are made persistent by *my_obj* $\rightarrow$ *persist*() and are fetched from the database by the *p_oodb* $\rightarrow$ *fetch*(...). OpenOODB fetch operation does not give the flexibility of specifying the lock mode but acquires a default *READ* lock from EXODUS storage manager. To provide the application programmer with more flexibility, we decided to modify the fetch operator of OpenOODB so that it takes the locking mode as a parameter. As a second stage, nested transaction primitives:

```
OODB *p_oodb;
My_Class *my_obj = new My_Class;
My_Class *tmp_obj;
char *obj_name = "obj1";
main()
{
        p_oodb →beginTransaction();
        /* make the object persistent and give a name to it */
        my_obj →persist(obj_name);
        p_oodb →commitTransaction();

        p_oodb →beginTransaction();
        /* fetch the object with the given name */
        p_oodb →fetch(obj_name);
        p_oodb →commitTransaction();
}
```

Figure 4.3: A Simple OpenOODB Application

- *spawn_sub_transaction*
- *commit_sub_transaction*
- *abort_sub_transaction*

are added to the transaction manager of OpenOODB (i.e., *TRANS_MGR* in Figure 4.2). Finally, a Lock Manager is implemented to support the nested transaction primitives that are added to the transaction manager. Among the nested transaction primitives, only the *spawn_sub_transaction* takes parameters. The first parameter of it is the name of the function where the subtransaction is written in, the second one is the spawn-mode. Spawn-mode specifies the coupling mode between the parent and child. For our nested transaction component, we implemented the IMMEDIATE and DEFERRED coupling modes. DETACHED coupling modes are handled by the rule manager of the ADBMS.

Using Figure 4.2 we can describe where our lock manager fits in the object relationships diagram. In that figure, the main object of OpenOODB (i.e., *OODB*), points to a *TRANS_MGR* object which has a *TRANSACTION* object. And the $TRANSACTION$ object has a $LOCK\_MANAGER$ object, i.e., the constructor of the transaction object creates the $LOCK\_MANAGER$ object which can be accessed by $OODB$. This way the constructs implemented in $LOCK\_MANAGER$ and $TRANS\_MGR$ can be used by the application via the OpenOODB interface object, $OODB$.

$LOCK\_MANAGER$ has two main data structures, namely the *Lock_Table* and the *Transaction_Table*. *Lock_Table* is a hash table that is used to keep the lock information of objects that have previously been fetched by a transaction in the transaction hierarchy. *Lock_Table* is hashed by the object name, and given an object, we can reach all the transactions that have a lock on this object with any mode. *Transaction_Table* keeps the transaction hierarchy, wait-for graph and the lock information of the subtransactions. These data structures are shown in Figure 4.4. We can see from the figure that those hash tables are interconnected, that is, we can reach the objects that are held by a transaction given its transaction identifier. This provides us with efficient abort and commit of subtransactions. *Transaction_Table* is hashed by transaction identifiers(*tid*). Given the *tid* of a transaction (from now on we will use the term transaction for both top-level transactions and subtransactions):

- We can reach all the objects held by that transaction in any hold and lock mode. Hold mode of a lock can be *hold* or *retain*, lock mode can be *read* or *write*.
- We can reach the transactions for which the given transaction is waiting.
- We can reach the transactions waiting for the given transaction.
- We can reach all the children and ancestors of this transaction.

For the parallel execution of subtransactions, Solaris threads are used [Sun94]. Solaris is a fully functional distributed operating and windowing environment [Sun92]. Thread is a sequence of instructions executed within the context of a process. Traditional Unix process contains a single thread of control. Solaris provides us with Multi-threaded Programming. Multi-threading separates a process into many execution threads each of which runs independently.

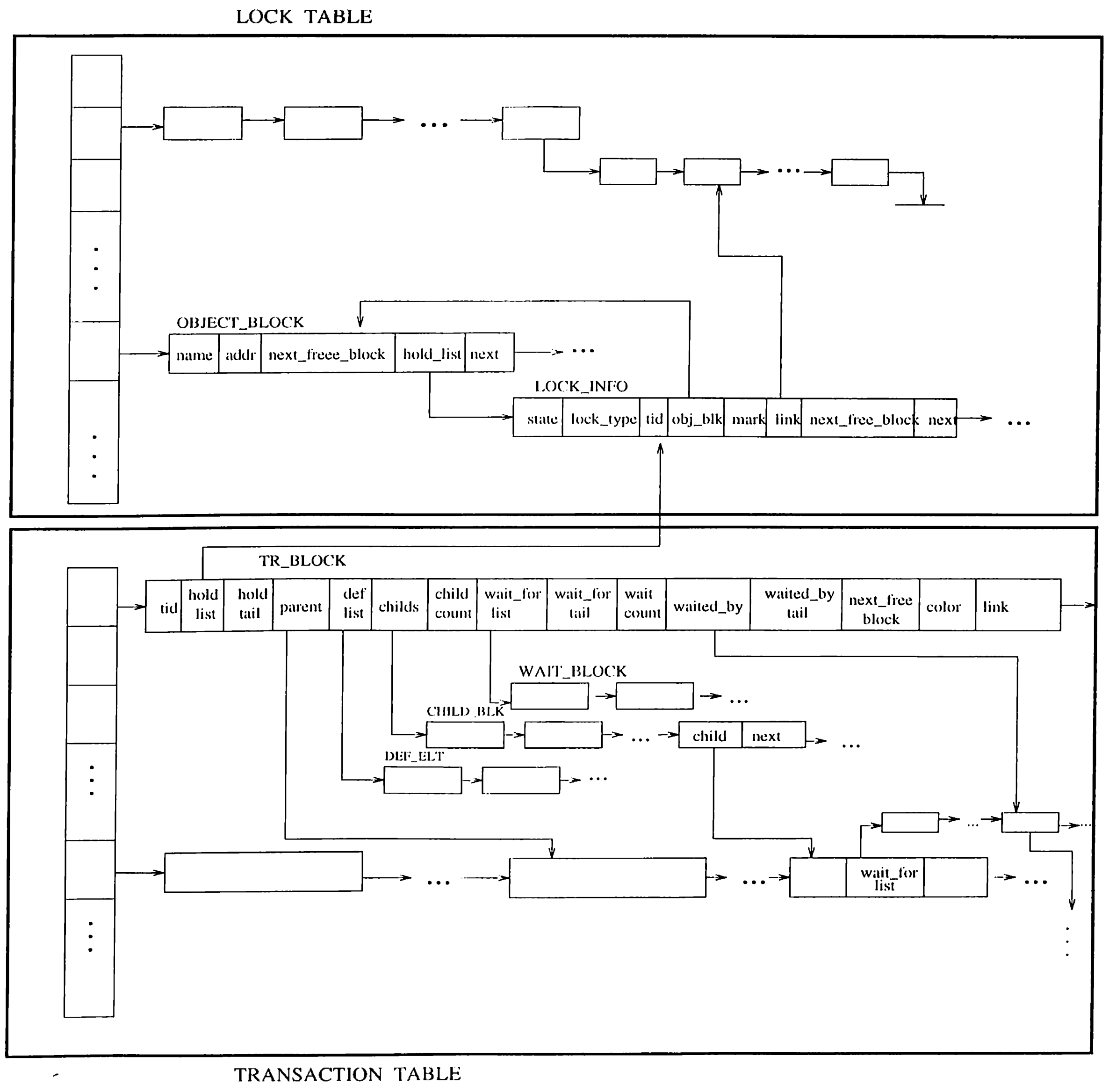

Figure 4.4: Data Structures

Primitive	**Explanation**
thr_create()	create a thread
thr_self()	return the thread identifier of the calling thread
thr_suspend()	block the execution of a thread
thr_continue()	unblock a thread
thr_kill()	send a signal to a thread
thr_exit()	terminate a thread
thr_join()	wait for the termination of a thread

Table 4.1: Thread Primitives Used

Advantages of Multi-threading can be listed as:

- overlap in time, logically separate tasks that use different resources,
- share the same address space,
- provide cheap switching among threads.

Thread primitives used in our implementation are listed in Table 4.1.

Mutual exclusive locks are used to control the concurrent access of different threads to the shared data structures. In Figure 4.5, we give a sample program for the creation of threads.

As can be seen from the figure, using *thr_create*() we execute a given function in a thread. In our implementation, subtransactions are defined as functions in a specified format and are executed concurrently using the *spawn_sub_transaction* primitive provided by our implementation of nested transactions. In Figure 4.6, the *spawn_sub_transaction* primitive executes the transaction, embedded inside a function, in a thread using *thr_create*(). Threads are created in suspended mode so that the necessary information is inserted into the *transaction_table*. Since the *tids* are unique within a Unix process, and top-level transaction boundaries do not exceed the process boundaries, it was very convenient for us to define the *tids* as the transaction identifiers. This way we do not need to pass the transaction identifier to the subtransaction as a parameter. Subtransactions can access their *tids* by calling the thread library function *thr_self*(). This function returns the thread

```
void *myfoo1(void *result)
{
    printf("myfoo1");
    printf("my id is:%d",thr_self());
}


void *myfoo2(void *result)
{
    printf("myfoo2");
}


main()
{
    // define two thread identifiers
    thread_t thr_id1;
    thread_t thr_id2;
    // functions myfoo1 and myfoo2 will be executed
    // in threads that are created in suspended mode
    thr_create(0,0, myfoo1, 0, THR_SUSPENDED,&thr_id1);
    thr_create(0,0, myfoo2, 0, THR_SUSPENDED,&thr_id2);
    // suspended threads are restarted
    thr_continue(thr_id1);
    thr_continue(thr_id2);
    // wait for the created threads to finish their execution
    thr_join(thr_id1);
    thr_join(thr_id2);
}
```

Figure 4.5: A Sample Multithreaded Program

identifier of the calling thread, i.e., the *tid*. When an OpenOODB top-level transaction is created, the *tid* is also inserted into the transaction table for the sake of completeness of the transaction hierarchy.

Since all the subtransactions in the transaction hierarchy can access the data structures in the *LOCK_MANAGER*, we define a global mutex variable. This way, the critical sections of the methods modifying the lock and transaction tables are wrapped by mutex-lock and mutex-unlock. Concurrent access to Lock Manager tables is discussed in Section 4.3 in more detail.

When we look at the data structures, we observe that there are linked lists belonging to both transaction and lock tables which means that deletions and insertions of new blocks to those lists take a lot of time. At this point, we made an optimization by implementing our own memory management via keeping lists of deleted blocks so that they can be used efficiently whenever they are needed. Another optimization was to extend our component with a sort of garbage collection, i.e., when we want to delete a block from the lock or transaction table, we do not delete it physically but mark it as deleted. This technique makes the usage of doubly linked lists unnecessary. Garbage collection is performed during the searches in the lock table.

4.3 Controlling Concurrent Access to Common Data Structures

Since all the transactions can reach the common data structures, i.e., transaction and lock table, there may be inconsistencies during the concurrent updates. To avoid this problem mutual exclusion locks are used. Only one mutex variable is defined which is locked and unlocked by the standard thread library functions *mutex_lock* and *mutex_unlock*, respectively. This way the critical sections of the nested transaction component methods (i.e., the sections where there is an update to the lock or transaction table) are wrapped so that only one transaction at a time can execute its critical section.

```
// create an OpenOODB main object
OODB *p_oodb;
void *sub2(void *res)
{
    // lock the object with name obj3 in WRITE mode
    int rc = p_oodb →fetch_object("obj3",WRITE);
    // in case of an error, abort the subtransaction
    // otherwise, commit the subtransactions
    if ( rc == ERROR )
      p_oodb → sub_abort();
    else
      p_oodb →sub_commit();
}


void *sub1(void *res)
{
    // lock the object with name obj1 in READ mode
    p_oodb →fetch_object("obj1",READ);
    // create a subtransaction in IMMEDIATE mode
    p_oodb →spawn_sub_tr(sub2,IMMEDIATE);
    // commit the subtransaction
    p_oodb →sub_commit();
}


main()
{
    // start an OpenOODB transaction
    p_oodb →beginTransaction();
    // spawn a subtransaction in IMMEDIATE mode
    p_oodb →spawn_sub_tr(sub1,IMMEDIATE);
    // commit the OpenOODB transaction
    p_oodb →commitTransaction();
}
```

Figure 4.6: Sample OpenOODB Application Using Nested Transactions

4.4 Implementation of the Locking Protocol for Nested Transactions

When a transaction requests a lock on an object it specifies the locking mode as well by providing the *fetch_object* method with the *lock_type* parameter. Allowed lock-modes are $READ$ and $WRITE$. A locking protocol for nested transactions is provided in Chapter 3. We implemented this locking protocol considering both READ and WRITE locks. Below we describe the locking protocol.

- A transaction T may acquire a lock in $READ$ mode if:
  - no other transaction holds a lock in $WRITE$ mode,
  - and all the transactions that retain a lock in $WRITE$ mode are ancestors of T.
- A transaction T may acquire a lock in $WRITE$ mode if:
  - no other transaction holds a lock in $READ$ or $WRITE$ mode,
  - and all the transactions that retain a lock in $READ$ or $WRITE$ mode are ancestors of T.
- When a subtransaction T commits, parent of T inherits all the locks (held or retained) that T has. After that, parent retains the locks in the same mode as T held or retained them before.
- When a top-level transaction commits, it releases all the locks it holds or retains.
- When a transaction aborts, it releases all the locks it holds or retains. If any of its ancestors holds or retains any of these locks, it continues to do so.

In Figure 4.7, if T_i retains a $WRITE$ lock on an object O_i and no other transaction inside the sphere S_i (i.e., T_j, T_k, T_l and T_m) has any lock on O_i, then all the transactions inside S_i can acquire a $READ$ or $WRITE$ lock on

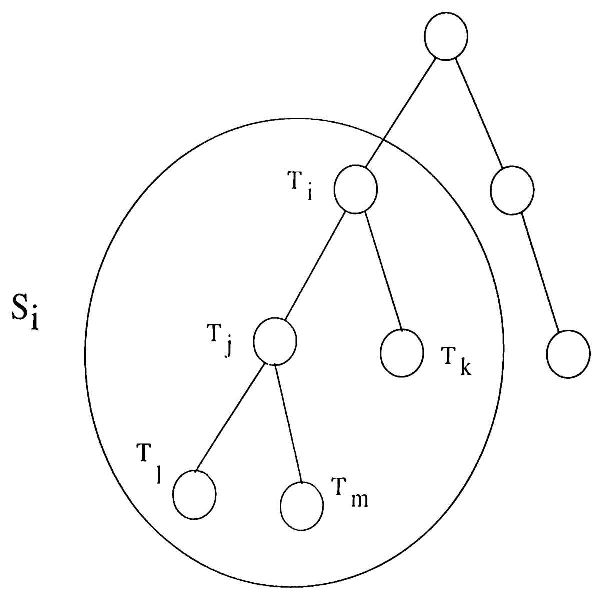

Figure 4.7: Control Sphere of T_i

O_i. If T_j acquires a $WRITE$ lock on O_i, then no other transaction (including the ones inside the sphere S_j in Figure 4.8) can acquire a $READ$ or $WRITE$ lock on O_i.

The *fetch_object* method first checks whether the requested object is in the lock table. If not, it just requests the object from OpenOODB and returns a pointer to it. If the requested object is in the lock table then nested transaction concurrency control protocol is put into action. If the lock can be granted, then a pointer to the object is returned as in the previous case. If the lock cannot be granted with the requested *lock_type*, then for each transaction that has a lock on the object that conflicts with the requested *lock_type*, a node is inserted to the *wait_for_list* of the lock requesting transaction and the same node is appended to the *waited_by_list* of the conflicting transaction. Additionally the *wait_for_count* (i.e., the number of transactions for which the transaction in concern is waiting for) is incremented for each node appended to the *wait_for_list*. Deadlock detection is performed for each node appended to *wait_for_list* . If no deadlock occurs then the transaction that requested the lock is suspended using *thr_suspend*(), otherwise it is aborted. Transactions can be unblocked using the *thr_continue*() function provided by the thread library. Unblocking of a transaction may occur due to the commit or abort of another transaction. When a transaction is aborted, all its locks are released,

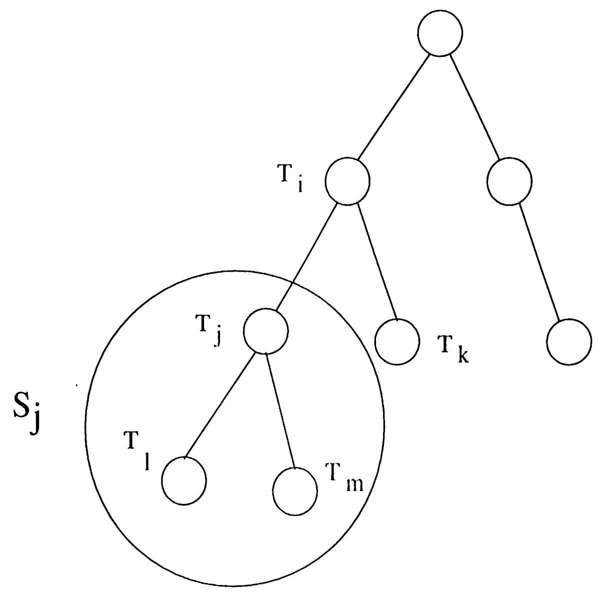

Figure 4.8: Control Sphere of T_j

and all the nodes in the *waited_by* list of that transaction are deleted; while doing the deletions, *wait_for_counts* of the corresponding (blocked) transactions are decremented by one. Blocked transactions whose *wait_for_counts* become zero, are unblocked, and their lock requests are reconsidered. When a subtransaction commits, all the locks held or retained by that transaction are inherited by the parent transaction which may cause some transaction(s) to be unblocked. Those transactions are identified by checking the *waited_by_list* of the committing transaction, and decrementing the *wait_for_counts* of the transactions which are descendants of the transaction inheriting the locks. The transactions whose *wait_for_counts* become zero are unblocked and their lock requests are reconsidered as in the previous case. A sample pseudo-code for the processing of lock requests is provided in Appendix A.

4.5 Deadlock Detection

Deadlocks may arise among subtransactions in the same transaction hierarchy as well as among subtransactions belonging to different transaction hierarchies. OpenOODB views a transaction hierarchy as one flat transaction; i.e., it is not aware of subtransactions. Lock requests made by a subtransaction is treated by

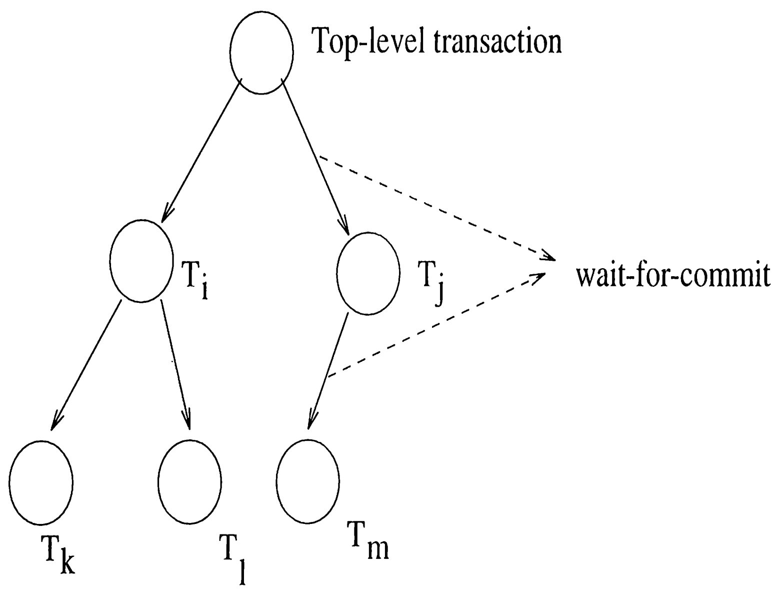

Figure 4.9: Wait-for-commit graph

OpenOODB as if the top-level transaction made the request. If there is a cycle among subtransactions belonging to different transaction hierarchies, then, from the point of view of OpenOODB, it means that there is a cycle among the top-level transactions as well. Therefore, deadlocks that occur among transactions belonging to different transaction hierarchies are resolved by OpenOODB via EXODUS storage manager. Deadlocks among the transactions belonging to the same transaction hierarchy are resolved by the new component managing nested transactions. Wait-for graph data structure is used to detect deadlock occurrences. Deadlock detection for nested transactions is different from the one for flat transactions in that, there are some other wait-for relations besides the wait-for-lock relation. The first wait-for relation associated with nested transactions is wait-for-commit; i.e., a parent transaction should wait for all its children to finish their execution. A wait-for-commit graph is illustrated in Figure 4.9 for a transaction hierarchy where the top-level transaction spawns subtransactions T_i and T_j, T_i spawns subtransactions T_k and T_l, and finally T_j spawns subtransaction T_m.

The second wait-for relation for nested transactions is wait-for-retained-locks. In flat transaction model, when a transaction commits, all the locks it holds are released, and transactions waiting for one or more of those locks can be unblocked immediately provided that they are not waiting for any

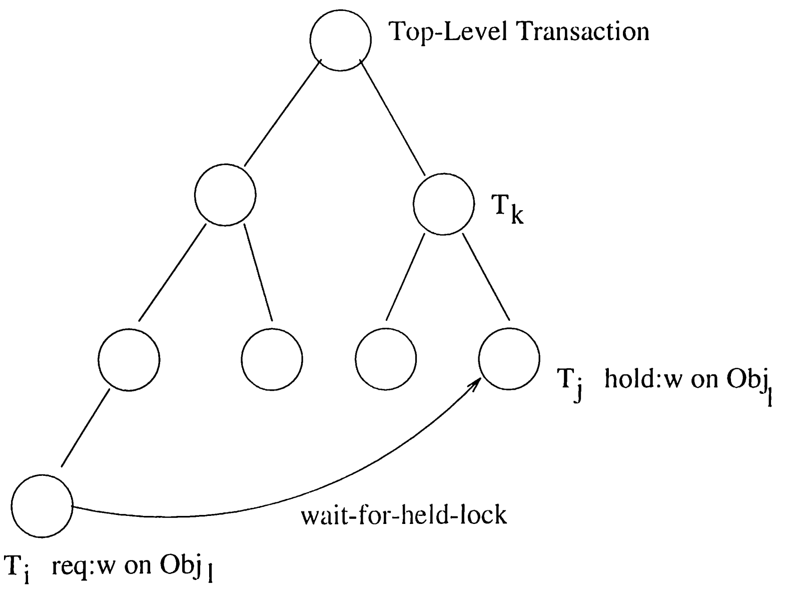

Figure 4.10: Wait-for-retained locks graph (before the commit of T_j)

other transaction. In nested transaction model, locks held by a subtransaction are not released immediately after it commits, but they are inherited by the parent transaction and kept in *retain* mode. This situation is illustrated in Figures 4.10 and 4.11.

In Figure 4.10 we see that transaction T_j holds a write lock on object Obj_l at the time when transaction T_i requests a write-lock on the same object. The arc labeled as wait-for-held-lock depicts this waiting situation. Following the commit of transaction T_j, all the locks that belong to T_j are inherited by the parent transaction, namely T_k, in retain mode. Transaction T_i still have to wait until the commit of T_k. T_i should wait until the first common ancestor of T_i and T_j inherits the lock on object Obj_l, which is the top-level transaction in this case.

Both wait-for-commit and wait-for-retained-locks should be taken into consideration in addition to the classical wait-for relation, for deadlock detection in nested transactions. Whenever a sub-transaction is spawned, a node is appended to the child-list of the parent transaction. The union of those linked lists is also used as the wait-for-commit graph. When a transaction requests a lock and blocks, all the transactions that cause this transaction to be blocked are kept as a linked list (wait-for list) to be used for unblocking the transaction

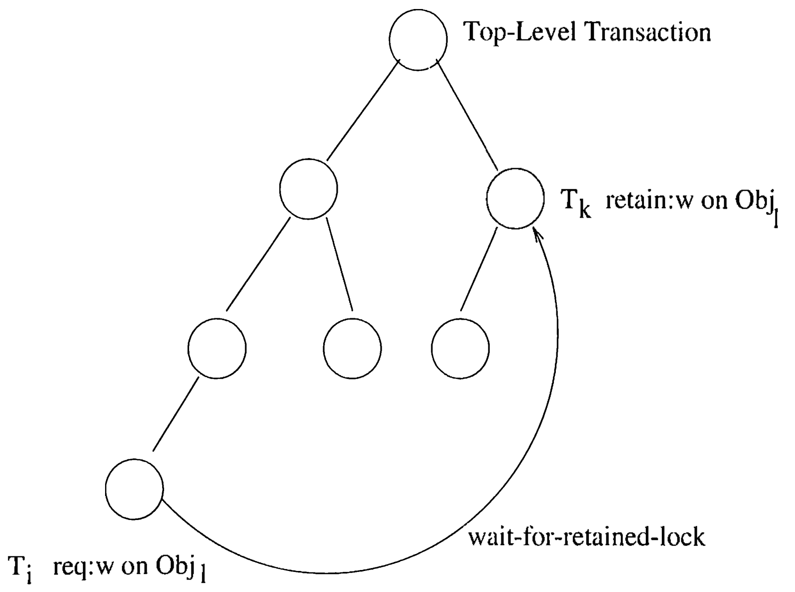

Figure 4.11: Wait-for-retained locks graph (after the commit of T_j)

later. Those lists also represent the wait-for-retained or held locks relationship. During the commit of a transaction, wait-for list of the committing transaction is inherited by its parent.

Deadlock detection is performed before each insertion to a wait-for-graph using the graph coloring technique. If a wait-for arc (T_i,T_j) is going to be inserted to the wait-for graph, deadlock detection algorithm is started assuming that the new link is added. Initially all the nodes of the graph are colored $BLACK$. Deadlock detection algorithm colors T_i as $WHITE$, and marks all the nodes on its way recursively till there are no remaining nodes or a $WHITE$ node is reached. Deadlock detection algorithm uses the child and wait-for lists of the transactions during its traversal. Since all the nodes are initially $BLACK$, a $WHITE$ node reached means that there is a cycle in the graph, implying a deadlock situation. Deadlock algorithm recursively restores the colors of all the nodes it traversed back to $BLACK$.

In case of a deadlock situation, we can abort T_i, T_j or any other transaction in the deadlock cycle. Deciding which transaction to abort is not an easy task. The simplest solution is to abort T_i (i.e., the transaction that requests the lock). Another possibility would be to abort the transaction that caused the lock-requesting transaction to block. Other possibilities for deciding which transaction to abort require the usage of some information about transactions such as the time stamp or the number of subtransactions. We can choose the transaction that has the smallest transaction hierarchy, or we can choose the transaction with the largest timestamp (i.e., the youngest one). As a future work, these alternative techniques could be implemented and their effects on the performance could be observed. We have chosen to use the simplest solution, i.e., to abort T_i. The pseudo-code of our deadlock detection algorithm is provided in Appendix A.

4.6 Integration of Our Implementation of Parallel Nested Transactions to Sentinel

Our implementation of parallel nested transactions is currently being integrated into Sentinel [CAM93] which is an Active Database Management System built on OpenOODB. In Sentinel, rules are treated as objects which means that they can be created, modified and deleted in the same way as other objects. Subscription mechanism is used to reduce the checking overhead of rules; i.e., when an object generates an event, during rule execution, only those rules which have previously subscribed to the object generating the event are checked.

In Sentinel, there is a *rule class* and all the rules in the system are instances of that class. The condition and action parts of a rule are implemented as methods in that rule class. The rule class is shown in Figure 4.12. The rule class is a subclass of the *notifiable class* which means that it can receive and record primitive events generated by reactive (i.e., event generating) objects. Each rule has a *name*, *event−id*, *condition*, *action*, *mode* and *enabled*. The event-id denotes the identity of the event object associated with the rule. Condition, and action are pointers to the condition and action member functions, respectively. The attribute mode denotes the coupling mode, and enabled indicates whether the rule is enabled or disabled.

Declaration of the condition and action functions should be modified so that they can be executed in subtransactions. The modified function declarations

```
class Rule:Notifiable // Rule class made notifiable
{
    char *name; // Rule name
    Event *event-id; //
    PMF *condition, *action; // PMF is a pointer
                             // to a member function
    Coupling mode; // Coupling Mode
    int enabled; // Rule Enabled or not

    public:
        virtual int Enable();
        virtual int Disable();
        virtual Update(Event* eventid);
        virtual int Condition();
        virtual int Action();
        Rule(Event* eventid, PMF condition, PMF action, Coupling mode);
        ~Rule();
```

Figure 4.12: Rule Class of Sentinel

should look like the sample subtransactions provided in Figure 4.6. Furthermore, the rule execution component of Sentinel should be modified so that conditions and actions of rules can be executed in parallel inside the subtransactions.

Chapter 5

Conclusions and Future Work

In this thesis, we described an execution model for active database management systems (ADBMSs). We used nested transactions in our execution model for rule execution and covered some aspects of nested transactions like recovery and concurrency control.

The advantages of our implementation of nested rule execution over the previous implementations can be listed as follows:

- Previous implementations for rule execution assume only sibling parallelism. In our implementation we assume both sibling and parent-child parallelism which is the most flexible kind of parallelism.
- We support deferred mode of execution as well as immediate mode.
- In previous implementations, subtransactions are executed in different processes. In our system, subtransactions are executed in threads which are more efficient than forking other processes.

In case of a deadlock situation, deciding which transaction to abort is an important issue which may affect the performance of the system. In our system, we used the method that has the simplest implementation; i.e., aborting the transaction that causes the deadlock to occur. Other heuristics for choosing the transaction to abort can be implemented and their effects on the performance can be studied.

Our nested rule execution module was designed in such a way that it can easily be ported to other systems as well, besides OpenOODB. As a future work, our implementation can be ported to different ADBMSs and its benefits and overheads in rule execution can be investigated.

Implementation of the recovery of nested transactions is left as a future work. A log based recovery method, ARIES/NT [RM89], was proposed for nested transaction recovery which is an extension of ARIES. Version based recovery techniques can also be applied which are easier to implement but have some major drawbacks against log-based recovery techniques [MHL$^+$92].

Our implementation of parallel nested transactions is currently being integrated into Sentinel which is an active database management system developed at the University of Florida. Following the completion of the integration, we are planning to investigate the impact of the parallel nested transaction component on the performance of Sentinel.

Appendix A

Sample Pseudocodes

A.1 Processing Lock Requests

```
// given the name of the requested object and requested lock type
// process the lock request of the transaction with identifier tid
$functionprocess\_lock\_request(char * name, LOCKTYPElock\_type, TIDtid)$
{
    // only one transaction at a time can enter this region
    $mutex\_lock(lock\_mgr\_mutex)$
    // get the address of the object using its name
    $obj_blk_ptr = hash\_obj\_name(name)$
    // perform garbage collection on the hold list
    $collect\_garbage(obj\_blk\_ptr \rightarrow hold\_list)$
    $conflict\_found = FALSE$
    $switch\ (lock\_type)$
    {
        $case\ WRITE$ // in case of exclusive lock
        // traverse the $hold\_list$ to check for conflicts
          $l\_info\_ptr = obj\_blk\_ptr \rightarrow hold\_list$
          $while(l\_info\_ptr \neq NULL$
          {
              // if the requested object is held by another transaction
if $(l\_info\_ptr \rightarrow state = HOLD)$
{
    // if the transaction requesting the lock has already acquired it
    if $(l\_info\_ptr \rightarrow tid = tid$ and $l\_info\_ptr \rightarrow lock\_type = WRITE)$
    {
        $mutex\_unlock(lock\_mgr\_mutex)$
        return $(obj\_blk\_ptr \rightarrow addr)$
    }
    else if $(l\_info\_ptr \rightarrow tid \neq tid)$
    }
        if $conflict\_found = FALSE$
        {
            $append\_to\_wait\_for\_list(tid, l\_info\_ptr \rightarrow tid, name, lock\_type)$
            $conflict\_found = TRUE$
        }
        else // if conflict was not found
            $append\_to\_wait\_for\_list(tid, l\_info\_ptr \rightarrow tid, name, lock\_type)$
    }
}
// state is RETAIN
// if the transaction holding the lock is not an acesstor
// of the requesting transaction
else if $(\neg ancestor(tid, l\_info\_ptr \rightarrow tid)$
{
    if $conflict\_found = FALSE$
    {
        $append\_to\_wait\_for\_list(tid, l\_info\_ptr \rightarrow tid, name, lock\_type)$
        $conflict\_found = TRUE$
    }
    else // if conflict was found
        $append\_to\_wait\_for\_list(tid, l\_info\_ptr \rightarrow tid, name, lock\_type)$
}
$l\_info\_ptr = l\_info\_ptr \rightarrow next$
$collect\_garbage(obj\_blk\_ptr \rightarrow hold\_list)$
```

```
} // end while
// If we reached the end of the list without any conflics
$if\ conflict\_found = FALSE$
{
  $acquire\_lock(tid, lock\_type, obj\_blk\_ptr)$
  $mutex\_unlock(lock\_mgr\_mutex)$
}
$else$ // conflict is found
{
  $mutex\_unloc(lock\_mgr\_mutex)$
  // suspend the transaction requesting the lock
  $thr\_suspend(tid)$
}
$case\ READ$ // in case of shared lock
// traverse the $hold\_list$ to check for conflicts
  $l\_info\_ptr = obj\_blk\_ptr \rightarrow hold\_list$
  $while(l\_info\_ptr \neq NULL$
  {
      // if the requested object is held by another transaction
      $if\ (l\_info\_ptr \rightarrow state = HOLD)\ and\ (l\_info\_ptr \rightarrow lock\_type = WRITE)$
      {
          // if the transaction requesting the lock has already acquired it
          $if\ (l\_info\_ptr \rightarrow tid = tid$
          {
              $mutex\_unlock(lock\_mgr\_mutex)$
              $return\ (obj\_blk\_ptr \rightarrow addr)$
          }
          $else\ if\ conflict\_found = FALSE$
              {
                  $append\_to\_wait\_for\_list(tid, l\_info\_ptr \rightarrow tid, name, lock\_type)$
                  $conflict\_found = TRUE$
              }
              $else$ // if conflict was not found
                  $append\_to\_wait\_for\_list(tid, l\_info\_ptr \rightarrow tid, name, lock\_type)$
```

```
                }
            }
            $else\ if\ (l\_info\_ptr \rightarrow state == RETAIN$
                $and\ l\_info\_ptr \rightarrow lock\_type = WRITE$
                $and\ \neg ancestor(tid, l\_info\_ptr \rightarrow tid)$
            {
                $if\ conflict\_found = FALSE$
                {
                    $append\_to\_wait\_for\_list(tid, l\_info\_ptr \rightarrow tid, name, lock\_type)$
                    $conflict\_found = TRUE$
                }
                $else$ // if conflict was found
                    $append\_to\_wait\_for\_list(tid, l\_info\_ptr \rightarrow tid, name, lock\_type)$
            }
            $l\_info\_ptr = l\_info\_ptr \rightarrow next$
            $collect\_garbage(obj\_blk\_ptr \rightarrow hold\_list)$
        } // end while
        // If we reached the end of the list without any conflics
        $if\ conflict\_found = FALSE$
        {
          $acquire\_lock(tid, lock\_type, obj\_blk\_ptr)$
          $mutex\_unlock(lock\_mgr\_mutex)$
        }
        $else$ // conflict is found
        {
          $mutex\_unloc(lock\_mgr\_mutex)$
          // suspend the transaction requesting the lock
          $thr\_suspend(tid)$
        }
}
// end case
$return(obj\_blk\_ptr \rightarrow addr)$
```

A.2 Deadlock Detection

```
// This function is called for deadlock detection
$function\ detect\_deadlock(start\_node, next\_node)$
{
    $start\_node \rightarrow color = BLACK$
    $result = detect\_rest(next_node)$
    $start\_node \rightarrow color = WHITE$
    $return\ result$
}



// This function is called by the main deadlock detection function
$function\ detect\_rest(node)$
{ hspace*0.2cm
                $if\ (node \rightarrow color = BLACK)$
                    $return\ DEADLOCK\_DETECTED$
                $else$
                {
                    $node \rightarrow color = BLACK$
                    // check for the wait-for-commit dependencies
                    $for(tmp1 = node \rightarrow child;\ tmp1 \neq NULL; tmp1 = tmp1 \rightarrow next)$
                    {
                        $result = detect\_rest(tmp1 \rightarrow child)$
                        $if\ (result = DEADLOCK\_DETECTED)$
                        {
                            $node \rightarrow color = WHITE$
                            $return\ DEADLOCK\_DETECTED$
                        }
                }
                   // check for the wait-for-lock dependencies
                   $for(tmp2 = node \rightarrow wait\_for\_list; tmp2 \neq NULL; tmp2 = tmp2 \rightarrow nex$
                   {
                       $if\ (tmp2 \rightarrow mark = UNMARKED)$
                    {
                        $result = detect\_rest(tmp2 \rightarrow tr\_ptr)$
                        $if\ (result = DEADLOCK\_DETECTED)$
                        {
                            $node \rightarrow color = WHITE$
                            $return\ DEADLOCK\_DETECTED$
                    }
                    }
                }
            $node \rightarrow color = WHITE$
            $return\ NO\_DEADLOCK$
            }
}
```

Bibliography

[AWH92] Alexander Aiken, Jennifer Widom, and Joseph M. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, pages 59–68, San Diego, California, June 1992.

[Bad93] Rajesh Badani. Nested Transactions for Concurrent Execution of Rules: Design and Implementation. Master's thesis, Department of Computer and Information Sciences, University of Florida, 1993.

[BDZ95] Alejandro P. Buchmann, Alin Deutsch, and Juergen Zimmermann. The REACH Active OODBMS. Technical report, Technical University Darmstadt, 1995.

[Buc94] Alejandro Buchmann. Active Object Systems. In Asuman Dogac, M. Tamer Ozsu, Alex Biliris, and Timos Sellis, editors, *Advances in Object-Oriented Database Systems*, pages 201–224. Springer-Verlag, 1994.

[BZBW95] A.P. Buchmann, J. Zimmermann, J.A. Blakeley, and D.L. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proceedings of the 11th International Conference on Data Engineering*, pages 117–128, Taipei, Taiwan, 1995.

[CA95] A. Chakravarthy and E. Anwar. Exploiting Active Database Paradigm For Supporting Flexible Transaction Models. Technical report, University of Florida, Computer and Information Science and Engineering Department, 1995.

[CAM93] S. Chakravarthy, E. Anwar, and L. Maugis. Design and Implementation of Active Capability for an Object-Oriented Database. Technical Report UF-CIS-TR-93-001, University of Florida, Department of Computer and Information Sciences, 1993.

[CR91] Panos K. Chrysanthis and Krithi Ramamritham. A Formalism For Extended Transaction Models. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 103–112, Barcelona, Spain, September 1991.

[CR94] Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[Day88] Umeshwar Dayal. Active Database Management Systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pages 150–169, Jerusalem, June 1988.

[DGRV95] Laurent Daynes, Olivier Gruber, Projet Rodin, and Patrick Valduriez. Locking in OODBMS Client Supporting Nested Transactions. In *Proceedings of the 11th International Conference on Data Engineering*, pages 316–322, Tai-pei(Taiwan), March 1995.

[HEKX94] Eric N. Hanson, Kurt Engel, I-Cheng Chen Vijay Kamaswamy, and Roxana Dastur Chun Xu. Flexible and Recoverable Interaction Between Applications and Active Databases. Technical report, University of Florida CIS Department, 1994.

[HLM88] Meichun Hsu, Rivka Ladin, and Dennis R. McCarthy. An Execution Model For Active Database Management Systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pages 171–179, Jerusalem, June 1988.

[HPS92] T. Harder, M. Profit, and H. Schonning. Supporting Parallelism in Engineering Databases by Nested Transactions. Technical Report 34/92, University of Kaiserslautern, Computer Science Department, 1992.

[HR93] Theo Harder and Kurt Rothermel. Concurrency Control Issues in Nested Transactions. *VLDB Journal*, 2(1):39–74, 1993.

[HW92] Eric N. Hanson and Jennifer Widom. An Overview of Production Rules in Database Systems. Technical report, University of Florida, Department of Computer and Information Sciences, October 1992.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwartz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[Mos85] E. Moss. *Nested Transactions.* M.I.T. Press, Cambridge, Mass., 1985.

[Mos87] J. Eliot B. Moss. Log-Based Recovery for Nested Transactions. In *Proceedings of the 13th VLDB Conference*, pages 427–432, Brighton, 1987.

[PD95] Norman W. Paton and Oscar Diaz. Active Database Systems. Technical report, University of Manchester, Department of Computer Science, 1995.

[PN87] Calton Pu and Jerre D. Noe. Design and Implementation of Nested Transactions in Eden. In *Proceedings of Sixth International Symposium on Reliability in Distributed Software and Database Systems*, pages 126–136, Kingsmill-Williamsburg, VA, March 1987.

[PP91] Edward Perez and Robert W. Peterson. Zeitgeist PersistentC++ User Manual. Technical Report 90-07-02, Information Technologies Laboratory, 1991.

[RM89] K. Rothermel and C. Mohan. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 337–346, Amsterdam, 1989.

[Sun92] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. *Getting Started with Solaris*, 1992.

[Sun94] Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. *Multithreaded Programming Guide*, 1994.

[WBT92] David L. Wells, Jose A. Blakeley, and Craig W. Thompson. Architecture of and Open Object-Oriented Database Management System. *IEEE COMPUTER*, pages 74–81, October 1992.

[Zuk95] Olaf Zukunft. Recovering Active Databases. In *Rules in database systems: Proceedings of the Second International Workshop on Rules in Database Systems*, pages 357–371, Athens, Greece, September 1995.