NESTED TRANSACTIONS FOR CONCURRENT EXECUTION OF RULES: DESIGN AND IMPLEMENTATION

By

RAJESH BADANI

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

UNIVERSITY OF FLORIDA

1993

Dedicated to my
Parents

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser, Dr. Sharma Chakravarthy, for showing me the path of research which was just a fantasy before, and for giving me an opportunity to work on the challenging *Sentinel* project. I am extremely grateful to Dr. Stanley Su and Dr. Nabil Kamel for serving on my committee and for their comments to improve the manuscript.

Many thanks are due to Mrs. Sharon Grant. I remember a sentence that might give insight into her indefatigable efforts in providing a well administered research environment, "Please turn off the coffee pot at night when no one is around," a note above the coffee pot in the Database Systems R&D Center.

I will also take this opportunity to thank all the graduate students at this center for their help and friendship. I am proud of their efforts in establishing this as a preeminent research group of the nation in databases, and also in making it a rendezvous of diverse cultures and experiences.

Last, but not the least, I thank my parents and family for their love. Without their encouragement and endurance, this work would not have been possible. Creativity, critical understanding, logical analysis, dedication to purpose and hard work — I learned from my father — played an indispensable background role in making this work happen.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

NESTED TRANSACTIONS FOR CONCURRENT EXECUTION OF RULES: DESIGN AND IMPLEMENTATION

By

Rajesh Badani

December 1993

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

In active databases, several rules may be applicable and ready to be executed when a event occurs. Production rule systems typically have used various conflict resolution strategies (e.g., arbitrary ordering, user defined priority, most instantiated rule, a new rule) for ordering the execution of rules. One of the rule execution strategies could be to execute all rules in parallel. When rules defined in a database are executed in parallel, and they access/modify shared data items in conflicting mode, there is a need for a correctness criterion that is consistent with the serializability criterion used for transactions.

The nested transaction model provides an ideal starting point for dealing with concurrent execution of event-condition-action rules in active databases. Although the nested transaction model only supports immediate coupling of rule execution (with the triggering transaction), it has been extended to support deferred and detached coupling modes.

In this thesis, the "flat" transaction model of Zeitgeist has been extended to a nested transaction model to support concurrent rule execution in the immediate

coupling mode. The concurrency control algorithm uses an extended set of attributes to keep track of the retain and grant information for each object. In our approach only nodes along the path from the current transaction to the root (virtual node) of the transaction tree is searched in the worst case (instead of all the nodes in the tree).

The design has been implemented on Zeitgeist–an object-oriented DBMS from Texas Instruments, Dallas. The type of intra-transaction parallelism supported is sibling concurrency. The hash table used for managing locks has been extended to an "anchored hash table" to minimize the search at the time of lock acquisition and commit of a subtransaction.

CHAPTER 1
INTRODUCTION

When multiple users access a database concurrently, their database operations execute in an interleaved fashion. Thus the operations have to be coordinated in order to prevent programs from behaving incorrectly and thus leading to an inconsistent database. This process of coordination is called concurrency control. It gives each user the illusion that s/he is referring to a dedicated database. A transaction is defined as a unit of concurrency control [2], which performs consistent and reliable computations. The consistency and reliability aspects of a transaction can be ascribed to four properties: atomicity, concurrency, isolation, and durability. Together they are referred to as "ACID" property of a transaction [5] and every transaction has to conform to this property. A Database Management System (DBMS) has to guarantee isolated execution of the entire transaction, thus guaranteeing that results obtained in a multiprogramming environment with interleaved execution of transaction will be same as some serial execution of the same set of transactions. Similarly other three properties also have to be guaranteed.

In current DBMS's, transaction management is typically designed for using a single level control structure. Its implementation is optimized to execute simple and short transactions [3]. The single level control (flat) structure of a transaction does not yield optimal flexibility and performance when executing large and more complex transactions. In these conditions, flat transactions have a disadvantage of the whole transaction being the unit of recovery and no intra-transaction parallelism. Thus the major concerns are decomposability and finer grained control of concurrency and recovery.

As a solution to this problem, the concept of nested transaction was proposed by Moss [8]. In nested transactions, properties of flat transactions are enhanced by a hierarchical control structure. Each nested transaction consists of either primitive actions or some nested transactions (called subtransaction of the containing transaction). Thus allowing dynamic decomposition of a transaction into a hierarchy of subtransactions and thereby preserving all the properties of a transaction as a unit and assuring atomicity and isolated execution for every individual subtransaction. Subtransactions have the same properties as their parents; in other words they may themselves have nested transactions. There are obvious restrictions on a nested subtransaction: it must begin after its parent and finish before it. Subtransactions are similar to flat transactions except that, they lack durability due to the fact that the commit of the subtransaction is conditional upon its parent's commit.

1.1 Advantages of the Nested Transaction Concept

Above mentioned properties of the nested transactions lead to the following advantages [6]:

Intra-transaction parallelism: Nested transactions provide appropriate synchronization between concurrently running parts of the same transaction, thus taking advantage of potential parallelism in a larger and complex transaction.

Intra-transaction recovery: Subtransactions of a nested transaction fail independently of each other and independently of the containing transaction. Thus subtransaction can be independently aborted and rolled back without any side effects to any other subtransaction. This limits the damage to a smaller part of the transaction, making it less expensive to recover. In flat transactions UNDO-recovery [4] yields the state of the previous savepoint. According to Gray and Reuter [4], emulation of nested transaction by savepoints is not as flexible as the general nested transaction model. It corresponds to the case where each subtransaction gets all the locks held

by ones parent transaction [8]. Essentially, a savepoint is a system defined interval, whereas each subtransaction corresponds to an interval, defined by the user, used for the purpose of rollback.

Reusability: In a nested transaction, it is possible to create a new transaction from existing ones simply by inserting the old ones inside the new transaction as a subtransaction. Thus a library of frequently used transactions can be defined and packaged into a nested transaction when necessary.

Distribution of implementation: Robustness of any particular system may be improved in various ways depending on how the distribution is done. Implementation of distributed algorithms embodied by a flexible control structure for concurrent execution is supported by the nested transaction concept. Overall efficiency is thus increased by the distribution of data and also by distributing the processing among various nodes.

Explicit control structure: The reliability of a transaction processing is greatly increased by providing an explicit control structure which allows for the delegation of work, and more importantly, their atomic execution.

System modularity: Design and implementation of independent transaction modules, which are subtransactions, facilitates the safe composition of transaction programs. Other design goals viz. security, encapsulation (information hiding), failure limitation are also achieved by this system modularity.

It can be easily seen that the advantages of nested transaction concept are not fully brought out in centralized DBMS, their potential is exploited more in distributed systems. But the advantages in centralized systems are significant enough and in a wide range of applications.

1.2 Motivation

Event-condition-action or ECA rules form the basis for supporting active capability [1]. The event part of the ECA rule specifies database, temporal, or explicit events; the condition part specifies a database query; and the action part specifies an arbitrary transaction. When the event occurs (is signalled), the condition is evaluated; if the condition is satisfied (i.e., if the query returns a non-empty answer), the action is executed. HiPAC allows decoupling of triggering event, condition evaluation, and the triggered action with respect to triggering transaction. Coupling modes are defined [7] for rules that determine when, relative to triggering event and transaction boundaries, the condition is evaluated and the action is executed.

The active database concept allows for greater flexibility and modularity, if the rules are treated as separate entities from the transaction. These have been modeled based on the fact that every rule triggered can be constituted as a separate transaction. Applications and model semantics govern the proper formalization of these rules. The grouping of detection of Events, evaluation of Condition, and execution of Action, depend on the model.

Most of the currently proposed strategies for executing triggered action essentially consider execution of the triggered action to be part of the triggering transaction. Thus, some rules maybe executed immediately when triggering event occurs, and others may be delayed until the end of the triggering transaction. In either case, the triggered action is executed essentially as a linear extension of the triggering transaction, and the atomicity requirement is applied to the combined execution. HiPAC calls the above immediate and deffered coupling modes, respectively.

While coupled execution describes an important class of rules, HiPAC proposed an execution model in which condition evaluation and triggered actions can be separated into different execution threads from those of triggering transaction. Allowing

condition evaluation and actions to occur in separate transactions permits the triggering transaction to finish more quickly and thereby release system resources earlier and improve transaction response time.

HiPAC also addresses the issue of concurrently executing several rules, triggered by the same event. HiPAC allows the triggered action to be executed concurrently, instead of executing them serially by using a conflict resolution strategy. In addition, HiPAC incorporated the capability for the system to handle rules triggered within the execution of another rule (i.e., nested firing of rules). Capabilities described above can be handled by a Nested Transaction Model. Combining various aspects of rule execution described above, coupling modes were defined for rules, which determine when, relative to the triggering event and transaction boundaries, the condition is evaluated and action is executed. Three modes are defined at which condition C can be evaluated and action A can be executed [7]:

- immediate: immediately when event E occurs and before next operation in transaction T,
- deferred: after the last operation in T and before T commits, or
- decoupled: in a separate transaction.

Depending on above three and some other constraints seven distinct coupling modes were defined:

- Evaluate C and execute A immediately when event E occurs and before the next operation in T.
- Evaluate C immediately when event E occurs and execute A after the last operation in T and before T commits.
- Evaluate C and execute A after the last operation in T and before T commits.

- Evaluate C immediately when event O occurs and execute A in a separate transaction T1.
- Evaluate C after the last operation in T and before T commits, and execute A in a separate transaction T1.
- Evaluate C and execute A together in a separate transaction T1.
- Evaluate C in one separate transaction T1 and execute A in another separate transaction T2.

A proper implementation of Nested Transactions should be able to support concurrent execution of rules and it should also be possible to separate three components of rules as done in some models. A model of Nested Transactions developed in this thesis is based on behavior of the combined execution of the user requests and triggered rules. With this model of nested transactions, we will be able to support:

- Immediate,
- deferred,
- decoupled modes of execution.

Another motivating factor was the goal of providing flexibility to the end-user or application programmer to spawn subtransactions and specify the segments for concurrent operations. This would help in some cases of optimization being provided by the user. Problems encountered in Distributed Databases was another factor affecting our research. Here, one transaction accessing data at multiple sites can spawn subtransactions for each sites. The overall structure of such an implementation is much cleaner, modular, and efficient than most currently existing ones.

CHAPTER 2
RELATED WORK

2.1 Nested Transaction: Model

In this section, unless stated otherwise, the term transaction is used to refer to both subtransactions and top-level transaction. Before providing a detailed discussion regarding concurrency control issues in nested transactions, we first introduce the terminology used and the underlying transaction model. For this purpose, we refer to the framework developed by Moss and follow his terminology defined in [8].

A transaction may contain any number of subtransactions, and every subtransaction in turn, may be composed of any number of subtransactions. This results in an arbitrarily deep hierarchy of nested transactions. The nesting of transactions can be represented by a *transaction tree*, which displays only the static aspects of the calling hierarchy. The transaction at the root of the tree, i.e. the one not enclosed in any transaction is called the *top-level transaction* (TL-transaction); others are called *subtransactions*. A transaction's predecessor in the tree is called a *parent*; subtransactions at the next lower level are its *children* . We can also think in terms of *ancestors* and *descendants*. The ancestor (descendant) relation is a reflexive transitive closure of the parent (child) relation. The set of descendants of a transaction T is called the *sphere of T*. We will be using the term *superior (subordinate)* for the non-reflexive version of ancestor (descendant). Thus non-descendants of a transaction T are all transactions in the universe except the transactions in the sphere of T. In the following we will be using the term 'transaction' to denote both the TL-transaction and the subtransactions.

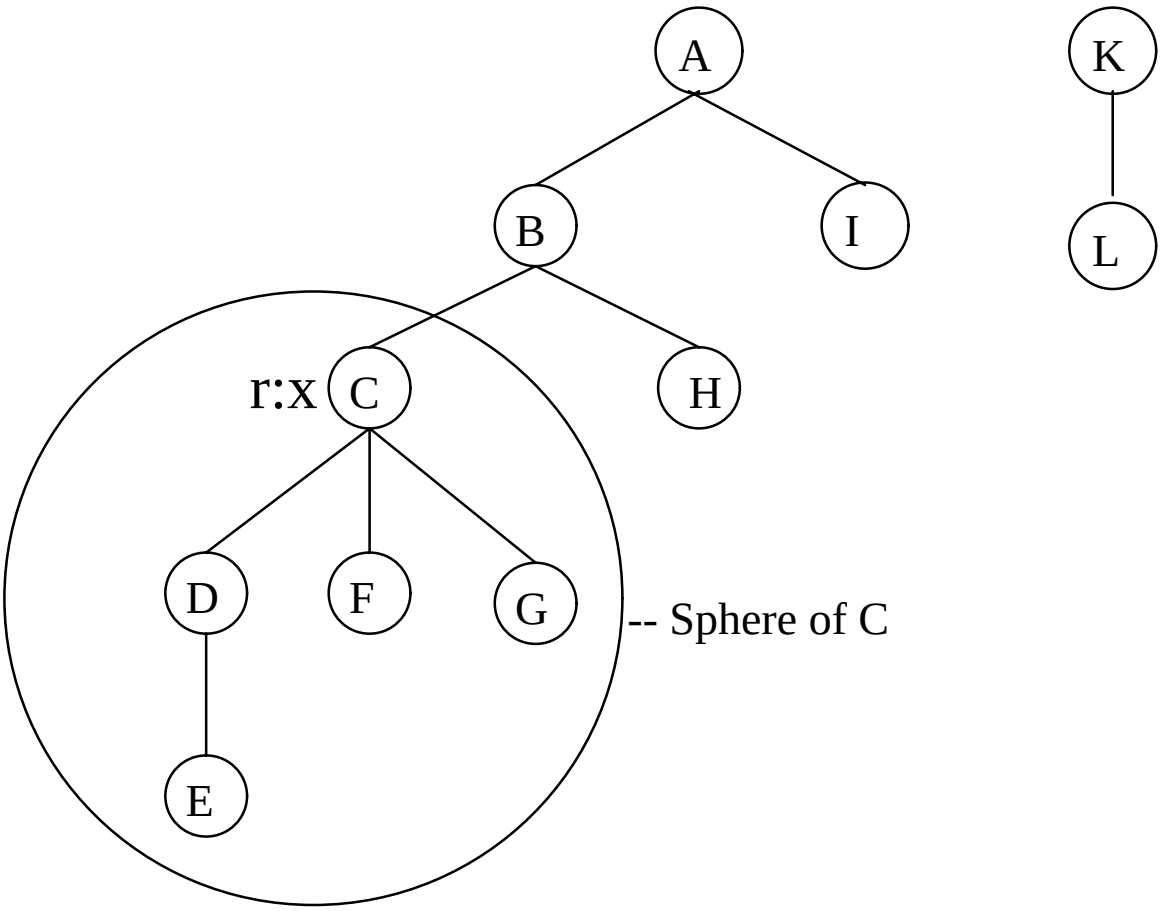


Figure 2.1. Example of Transaction Tree

In the nested hierarchy of a TL-transaction, the node of a tree represents transactions, and an edge represents the nesting relationship between the related transactions. In the transaction tree shown in figure 2.1, the root is represented by the TL-transaction A. The children of the subtransaction C are D, F, and G and parent of C is B. The subordinates of C are D, E, F, and G and the superiors are B and A. Also the descendant and ancestor sets of C additionally contain C itself. Finally non-descendants of C are A, B, H, I and also K, and L of other nested transaction. As indicated in figure 2.1, the sphere of C is the set of descendants of C, (which includes C itself).

We can also think in terms of every transaction being embodied by a single process to facilitate comprehension. This type of nested transaction hierarchy may be explained as a collection of *nested spheres of controls*, where the outer-most sphere is formed by the TL-transaction which incorporates the interface to the outside world (user and other transactions). The ACID properties defined for flat transactions are assumed to remain valid for TL-transactions. Subtransactions can terminate either by committing or by aborting like a TL-transaction. Subtransactions appear atomic to the surrounding transactions and may commit or abort independently. A subtransaction can abort without affecting the outcome of the surrounding transactions, but the commit of the subtransaction is conditional subject to the commit of its superiors; even if a subtransaction commits, aborting one of the superiors will undo its effect. All updates of the subtransaction becomes permanent only when the enclosing TL-transaction commits. Furthermore, it would be unnecessarily restrictive to require consistency to be preserved by a subtransaction, since there are times when the parent transaction needs the results of several child transactions to perform some consistency preserving actions. Thus as opposed to a TL-transaction, it would be

sufficient to provide weaker properties for subtransactions. In fact, atomicity and isolated execution are the only essential properties for nested transactions.

To take advantage of the inherent potential of nested transactions, the degree of intra-transaction parallelism should be as high as possible. There are two primary types of intra-transaction parallelism: parent/child concurrency and sibling concurrency. In the first, a transaction can run concurrently with its children, while in the second, siblings are allowed to run concurrently.

Using the above, we can characterize four levels of intra-transaction parallelism [6]:

Neither parent/child nor sibling concurrency: In this, there can be no more than one transaction active in the sphere of TL-transaction. In other words, there is no intra-transaction parallelism at all. Since all (sub)transactions in a transaction's sphere are executed serially, we do not require any concurrency control among transactions belonging to the same TL-transaction. All systems in which each (sub)transaction is executed by a single process and which only allows for synchronous inter-process communication provide this level of concurrency.

Sibling concurrency: In this, only siblings are scheduled concurrently. Thus a transaction is never executed concurrently with its superiors or subordinates. Only the leaf nodes within a sphere of each TL-transaction will execute concurrently. This type of restricted parallelism enables a transaction to share objects with its ancestors without further concurrency control between them.

Only parent/child concurrency: In this case, a transaction and its children can be executed concurrently, but siblings cannot. Thus in the sphere of TL-transaction only the transactions along one path of the transaction hierarchy can execute concurrently. This kind of restriction simplifies intra-transaction concurrency control in the sense

that only transactions residing in the same path have to be synchronized with each other.

Parent/child as well as sibling concurrency : This level permits arbitrary intra-transaction concurrency, i.e., in principle, all transaction in a TL-transaction's sphere may execute in parallel. Of course, as compared to the degrees of parallelism described above, this degree requires the most sophisticated concurrency control scheme. One disadvantage is of a high possibility of a deadlock (if a subtransaction request's for a lock held by an ancestor transaction), thus enforcing the rule that a subtransaction and a parent transaction have to access disjoint objects. This also provides the maximum amount of concurrency possible for a transaction execution.

In this thesis, we delimit our discussion to sibling concurrency. In most of the database applications parents will require results from its children and thus will have to wait for termination of the children transactions. Thus, sibling concurrency can provide high degree of parallelism, since, at a time any number of children of a transaction can be executed concurrently, while in parent/child concurrency only transactions along a path can run concurrently. Although, Parent/child as well as sibling concurrency would provide maximum parallelism for database applications, it does not seem to be beneficial considering the complexity of its implementation.

2.2 Concurrency Control in Nested Transactions

In this section we discuss the locking rules for nested transactions. We chose locking as a method of concurrency control for the following reasons: firstly, it has been used successfully in most of the commercial applications for a long period of time, and secondly the underlying system, on which we propose to implement the algorithms presented in this paper, also uses locking for concurrency control. We assume three modes of synchronization: (1) read, which permits multiple transactions to **S**hare an object at a time and also allow the lock to be upgraded, (2) write, which

reserves e**X**clusive update access to an object for a single transaction, and finally (3) **R**ead **O**nly, which again permits multiple transactions to share a lock but does not allow locks to be upgraded. Also for concurrency control issues, objects serve as lockable units.

Before describing the details of concurrency control issues, we state the objectives of synchronization [6]:

- Concurrency control among nested TL-transactions has to prevent database updates performed by one transaction from interfering with database updates and retrievals from other transactions. Therefore serializability is required as far as the 'outside world' is concerned, which implies that a transaction (TL-transaction together with its descendants) observe a strict two-phase locking protocol for synchronizing its database accesses with other transactions.

- Within a nested transaction, concurrency control has to ensure effective and safe, but *least restrictive cooperation possible*. Thus, its goal is to provide controlled parallelism between siblings whenever acceptable by data consistency demands.

In what follows, we focus on intra-transaction concurrency control. We start with basic locking rules proposed by Moss [8] and approach more complex rules by stepwise refinement, also evaluating the benefits in doing so and the problems involved. Read-only mode has also been added to the basic rules.

We have four possible lock modes: NL, RO, S, and X. The Not Locked mode (NL) represents the absence of a lock request or a lock on the object. A transaction can acquire a lock on an object in some mode; then it holds the lock in the same mode until its termination (commit or abort) or until it upgrades the lock mode explicitly. Also, besides holding a lock, a transaction can *retain* a lock. Retention concept is required for modeling the correctness of nested transaction execution. When a subtransaction

commits, its (retained and held) locks are inherited by its parent transaction, which in turn retains these locks. A retained lock is a place holder, i.e. unlike a lock 'held' by a transaction, for which a transaction has all the rights to access the locked object (in corresponding mode), a transaction retaining a lock does not have any right to access that object. A retained X-lock denoted by r:X (as opposed to h:X for an X-lock held), indicates that the transactions outside the sphere of the retainer cannot acquire the lock, but the descendants of the retainer potentially can - subject to the locking rules given below. That is, if a transaction retains an X-lock, then all non-descendants of that transaction cannot hold the lock either in X- or in S-mode. If a transaction is a retainer of S-lock, it is guaranteed that a non-descendant of that transaction cannot hold the lock in X-mode, but potentially in S-mode. As soon as a transaction becomes the retainer of a lock, it remains retainer for that lock during its lifetime (i.e., till it commits).

In the example illustrated in figure 2.1, transaction C retains a lock for object O in X-mode. This enforces that all transactions outside the sphere of C cannot acquire O in any mode, but the descendants of C potentially can.

We now describe the *basic* locking rules [6], including the extensions related to read-only mode. The discussions are with respect to an object O.

1. Transaction may acquire a lock in X-mode if

    - no other transaction holds the lock in X- or S-mode, and
    - all transactions that retain the lock in X- or S-mode are ancestors of the requesting transaction.

2. Transaction may acquire a lock in S-mode if

    - no other transaction holds the lock in X-mode, and

- all transactions that retain the lock in X-mode are ancestors of the requesting transaction.

3. Transaction may acquire a lock in RO-mode if

   - no other transaction holds the lock in X- or S-mode, and
   - all transactions that retain the lock in X- or S-mode are ancestors of the requesting transaction.

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

5. When a subtransaction commits, the locks it held or retained are inherited by its parent. After that the parent retains the locks in the same mode (X or S) as held or retained by the child earlier. If the parent already retains the lock, it keeps the more restrictive mode (multiple retainment rule).

In the last rule, if the parent transaction already retains a lock on that object, we must perform a 'union' (least upper bound) of the already retained mode and newly inherited mode. The basic modes are totally ordered as shown below:

$$\text{NL} < \text{RO} < \text{S} < \text{X}$$

Hence NL is least restrictive and X is most restrictive. Using this ordering, we can easily compute a new retain mode for the parent when a child commits:

$$\text{new mode of parent} = \text{MAX}(\text{old mode of parent, child's mode})$$

A transaction's retain mode never decreases when this rule is obeyed. Also, whenever a transaction requests and is granted a lock, it obtains the lock in the restrictive of the requested mode and the mode in which it previously held the lock:

$$\text{new mode} = \text{MAX}(\text{requested mode, old mode})$$

CHAPTER 3
ISSUE IN DESIGN

3.1 Issues Involved

In nested transactions, when a transaction requests a lock on an object, there are a number of factors to be considered before granting a lock:

- Does any transaction or subtransaction hold the lock, in any mode, on that object ?
- Does any transaction or subtransaction retain the lock, in any mode, on that object? If so, which transaction holds the lock, in what mode, and its relationship with requesting transaction?

Granting a lock request depends upon the type of lock held or retained, and who retains it. Also to find all this information, we may have to search each node of the tree exhaustively. We have to do this exhaustive search to find if any transaction in the system holds the lock or retains it in some mode and if the lock is being retained by a transaction, whether that transaction is an ancestor of the requesting transaction.

The discussion on nested transaction so far suggests that, along with lockmodes (i.e. eXclusive, Shared etc.) we also need to store the retaining information about the transaction. Thus we introduce additional fields and to differentiate them from lockmodes we call it *holdmodes*. Nested transaction can be implemented with retain() and lockmodes alone, but with this limited information, when a transaction requests for a lock on an object, in any mode, we have to perform an exhaustive search and identify the nodes inside and outside the sphere of control using this object. We have

to search the transaction tree of the requesting transaction and if no entry was found for that object, then search the transaction trees of all the transactions in the system.

Depending on the applications, we can have two types of transaction tree structures; bushy and deep tree structures. If the transaction tree structure of an application is bushy, then exhaustively searching the tree structures, of requesting transaction and if required all the transactions in the system, every time a lock is requested, may be affordable (in terms of the search time required). But for deep trees, it is likely to be inefficient as this needs to be done for acquiring each lock. Thus, we require a methodology which restricts this search.

One way to limit the search is to store all the information at the root node. Thus we will have all the lockmodes and retain() information at the root node, which can be used for checking the availability of locks. If the lock is available, then find out the retaining information on the lock, and if the lock is being retained, we need to find whether that node is an ancestor of the requesting node (using the basic rules discussed earlier). This will limit the search to a great extent, as all we need to check at every request is the information at the root node. The memory space required will also be in acceptable limits. But this scheme has problems as discussed below.

We identified two problems with above scheme :

- Distributed nature of nested spheres of control.
- Distributed nature of disjoint spheres of control.

3.1.1 Distributed Nested Spheres

Firstly, we describe nested spheres of control and discuss the related problem. As was seen in figure 2.1, whenever a transaction retains a lock, a sphere of control is formed. We also discussed that all the transactions in the sphere of control can acquire a lock on an object in any mode. Figure 3.1 illustrates the case where an

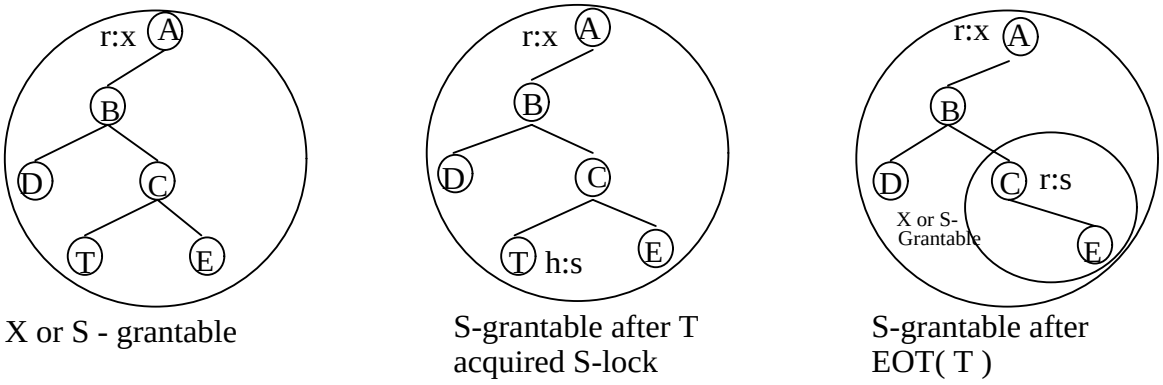


Figure 3.1. Example of Nested Spheres

S-lock is granted in a transaction hierarchy whose root retains the lock in X-mode (which can be a result of a child of transaction A acquiring a lock in X mode and later committing). While this hierarchy was 'X or S-grantable' before the S lock was given to subtransaction T, it remains only 'S-grantable' thereafter. After commit(T), a part of this hierarchy becomes 'X or S-grantable' again. Thus we have nested spheres of control.

In figure 3.1, transaction D is in the sphere of control of A. Thus by checking the ancestors, transaction D will find itself in sphere of control of A, and as A retains a X-lock it will find that it can acquire a lock in any mode. But if we take a closer look, transaction C also retains a lock on same object and has formed a new sphere of control within the original sphere of control. Thus by basic locking rules, transactions in the inner sphere of control should get a lock in any mode, but transactions outside should get lock on that object only in S-mode (as retain of transaction C is in shared mode). Thus transactions D should be allowed to take a lock on that object only in S-mode. This is contradictory to basic locking rules which state that a transaction inside a sphere of control can get lock in any mode. This indicates that the retain mode of inner circle need to be taken into account for accessing data in a particular mode. The solution here is to allow the most restrictive mode to prevail. i.e. for

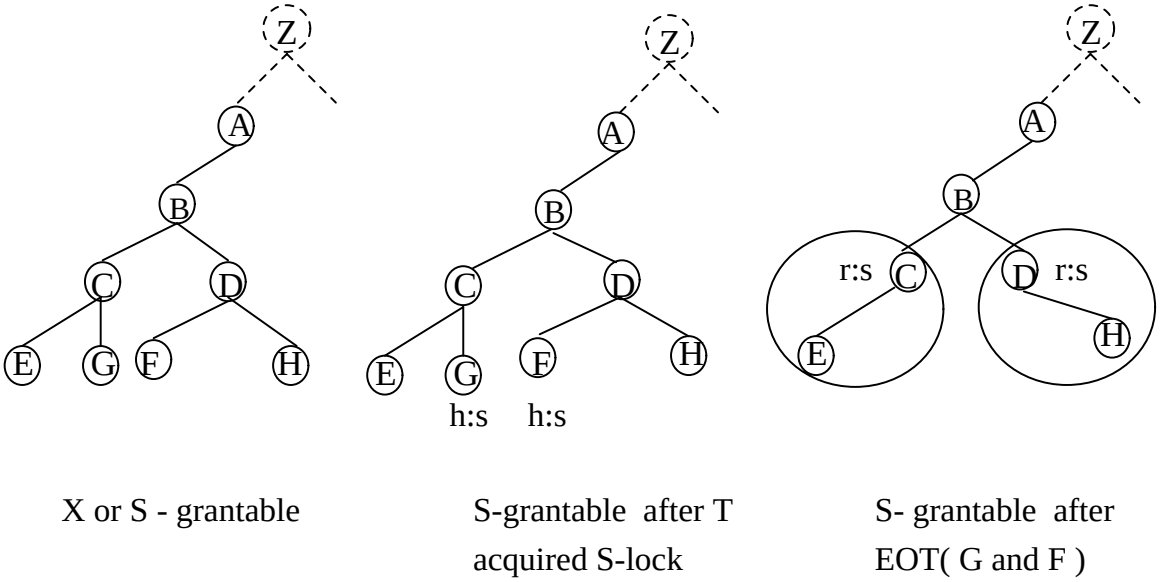


Figure 3.2. Example of Disjoint Spheres

this example, we should allow transaction D to take a lock on that object only in S-mode and **not** in X-mode. Thus we need to know if some subtransaction or ancestor is retaining the lock on the same object as you do. Thus, we need to store more information along with retain in holdmodes (explained in detail in next section).

3.1.2 Distributed Disjoint Spheres

The second phenomenon is of disjoint spheres of control. Consider the case when two siblings spawn children, and these children want to acquire lock on same object (say in S-mode). In figure 3.2 transactions C and D spawn children E, G, and F, H respectively. Suppose transactions G and F take S-lock on the same object. If now both G and F commit, they form two disjoint spheres of control. There are two disjoint spheres, the sphere of control of C will include C and E, and that of D will include D and H. Now if transactions E and H check the information (e.g., lockmode) at the ancestors, they find themselves in sphere of control and may acquire a X-lock on same object (according to basic locking rules no transaction can acquire a X-lock, but this reduces concurrency; (see detailed discussion in section 5.2). This

is due to the fact that two disjoint spheres are formed with C and D as root nodes and both of which have retained a lock on the same object. The solution to this is, in addition to storing more information, we need to store holdmodes at the nearest common ancestors (again discussed in more detail in next section). Note that in case of eXclusive locks, disjoint spheres don't form at all.

All of the above discussion points in the direction of keeping additional information to infer the nature of locks held without having requiring a search on the entire tree. We propose to keep lockmodes and holdmodes at all the ancestors of a particular transaction. Thus we will need information at the nearest common ancestor or at the root transaction. Hence, if the transaction wants a lock on an object, it will first check its own transaction tree (i.e. it's own modes, modes at parent node and modes in its path to root transaction), and if need be check the root nodes of all the transactions in the system (root transaction being common ancestor of all the transactions in the tree will have all the lock and hold modes). Our approach eliminates unnecessary check on the entire tree.

To further improve the efficiency, we have introduced the concept of a virtual transaction. As shown in the figure 3.2, Z is the virtual transaction and it is the immediate parent of all the top-level transactions in the system. If there is no entry for an object in requesting transaction's tree, then instead of searching the roots of all the transactions in the system, we have to search only the virtual transaction, further limiting the search time. It also makes the concept of nested transactions simpler in a way that now there is only one virtual TL transaction in the system !

3.2 Modeling the Design

In this section we introduce the fields of holdmodes, which will be used to keep the current status of the locks on objects used by active transactions in the system.

Lockmodes can specify whether the lock is being held in Shared, eXclusive, or Read-Only modes. Holdmodes consists of five fields as follows :

hold(x)/ holdsub(x)/ retain(x)/ retainsub(x)/ grantmode

These are the extensions made for the nested transaction model. The fields of holdmode defines the status of a lock and its availability to the transactions in the sphere of control and to the transactions outside. The arguments given in the values represents the object in question and not the mode of lock. The algorithm proposed in this thesis use these fields along with lockmodes to determine the availability of lock in a particular mode. Below, we define each field and discuss it in detail.

3.2.1 Holdmodes

We have five different fields in holdmode, which are defined as follows :

Hold(T): This field indicates whether the transaction in question is holding a lock on a particular object T. The lock can be held in three modes; eXclusive, Shared, and Read-Only, represented by X, S, and RO, respectively. A lock in RO-mode cannot be upgraded to any other mode, and a lock in S-mode can be upgraded to X-mode. Also, when a transaction spawns a child, all the locks held by the parent *may* be inherited by the child.

Holdsub(T): Holdsub field of the holdmode gives us the information about the number of subtransactions holding the lock on an object T. It is a numerical value. If the lock is held in Shared or Read-Only mode, it will indicate the number of (sub)transactions holding the lock on an object. This information will be used by subtransactions in deciding whether to acquire a lock or not. This field along with grantmode and retain fields will help determine whether a lock can be granted (and the mode in which it can be granted) in presence of nested and disjoint spheres of control discussed earlier. If lock is being held in eXclusive mode, then this value will be one.

Retain(): When a subtransaction commits, all its locks are inherited and retained by its parent. According to basic locking rules, other transactions in the system can be granted a lock on an object. Retain field of the holdmode can take values S, or X, depending on the mode in which the object was retained. Retain in the holdmode field indicates the presence of a sphere of control, of which it is a root. Note that there is no need to retain the Read-Only mode.

Retainsub(): Retainsub field of the holdmode indicates the number of descendent transactions retaining the lock on an object. Thus it is a numerical value indicating the number of subordinate transactions retaining the lock on an object. By checking this field of the parent or ancestor, it is possible to obtain the information about the existence of one or more of spheres of control. This information will help in the two cases of spheres of control discussed earlier.

Grantmode: This particular field of the holdmode gives us the information about the available lockmode, i.e. the lockmode in which the lock request can be granted. It can take four different values –, X, S, RO. Where '–' in grantmode field has the meaning of a lock being grantable in **any mode**, 'X' means that lock is grantable in **no mode**, 'S' means that lock is grantable in **shared mode only**, and 'RO' has the meaning of a lock being grantable in **read-only mode**. As explained earlier, due to the problems of nested spheres of control and disjoint spheres of control (and combination of both), retain alone is not enough to grant a lock on an object. In the example of nested spheres of control (figure 3.1) discussed earlier, we can now set the grantmode field of all ancestors of transaction C to 'S'. Thus when transaction D requests for an X-lock on an object, we can check the grantmode for that object at B, and the lock will be denied, as the grantmode says that the lock is grantable only in S-mode.

3.2.2 Meaningful Combinations

In the previous section we described five fields of holdmode. Each of these five fields can take different values, thus giving rise to numerous combinations. However, there are only sixteen meaningful combinations, which are described as follows :

1.	–	/	–	/	–	/	–	/	–	
2.	Hold	/	–	/	–	/	–	/	–	
3.	–	/	HS	/	–	/	–	/	S | X | RO	
4.	–	/	–	/	Retain	/	–	/	–	
5.	–	/	–	/	Retain	/	RS	/	S | X	
6.	–	/	–	/	–	/	RS	/	S | X	
7.	–	/	HS	/	–	/	RS	/	S | X	
8.	–	/	HS	/	Retain	/	RS	/	S | X	
9.	–	/	HS	/	Retain	/	–	/	S | X | RO	
10.	Hold	/	HS	/	–	/	–	/	S | X | RO	
11.	Hold	/	–	/	Retain	/	–	/	–	
12.	Hold	/	–	/	Retain	/	RS	/	S | X	
13.	Hold	/	–	/	–	/	RS	/	S | X	
14.	Hold	/	HS	/	–	/	RS	/	S | X	
15.	Hold	/	HS	/	Retain	/	RS	/	S | X	
16.	Hold	/	HS	/	Retain	/	–	/	S | X | RO	

- **– / – / – / – / – :** This is the initial state, lock tables are initialized to this value to start with.

- **Hold / – / – / – / – :** When the transaction is granted an lock on a object, or is granted a upgrade on an lock, the hold field in its holdmodes is changed from '–' to 'Hold'. This indicates that a transaction is holding a lock on a particular object.

- **– / HS / – / – / S | X | RO :** As a transaction is granted a lock on an object, the holdsub field of all the ancestors of that transaction is incremented by one. This indicates that certain number (depending on the number in HS field) of subtransactions are holding a lock on an object. Also, the grantmode for all the ancestors need to be upgraded. The new value of the grantmode will be the most restrictive of the mode acquired and the mode already present. Thus we can infer that if the grantmode field is X then the HS field will be either zero or one.

- **– / – / Retain / – / – :** When a transaction releases a held lock, its *immediate* parent will inherit that lock and retain it. For example if only one subtransaction (of a parent) was holding a lock on an object, and it releases that lock, its parent's mode will change from '–/1/–/–/S | X' to above and the HS field of all the ancestors of parent will be decremented by one.

- **– / HS / Retain / – / S | X | RO :** In the above case if more than one transaction was holding a lock on the same object, we have to keep the HS and grantmode field intact. The restriction of grantmode still applies, and certain number of subtransactions are still holding a lock on that object. But as one of them have terminated we have to decrement the HS field of all the ancestors of terminating transaction.

- **– / HS / – / RS / S | X :** Starting from the above case, we know that the parent of the committed transaction retained the lock. If parent retained that lock for first time, we have to update the RS field of all the ancestors of the parent. Every thing else remains the same. In other case we do not increment the RS field of ancestors as it has already been done previously.

- **– / HS / Retain / RS / S | X :** If one of the subtransactions (of the transaction which retained the lock in fourth case) retains the lock, then the holdmode will change from the fourth case to the one above. Still we are assuming that some of the subtransactions are holding a lock on that object, and thus the HS field has a positive value.

- **– / – / Retain / RS / S | X :** If there were no subtransactions holding a lock on the object in question, then the HS field will be 0 and thus absent.

- **– / – / – / RS / S :** When the above becomes true, the holdmode of the ancestors of the parent will change to the one above.

Cases 10-16 are similar to cases 3-9, except that the former has the parent transaction holding the lock at the time of spawning child transactions. In this case, child transactions don't automatically get all the locks held by parent transaction, but has to request for each individual lock it requires. Thus locks are inherited on demand and is in accordance with rules proposed in this thesis.

CHAPTER 4
SPHERES OF CONTROL

When a transaction acquires a lock on an object and commits, all its locks are inherited by its immediate parent, which retains the locks in the same mode as it was held by the subtransaction. When a transaction retains a lock, a sphere of control is formed. This sphere will include all the children (and their descendants) of the retaining transaction. Depending on the basic locking rules and the retaining mode, transactions outside the sphere can acquire a lock on an object. There are three different ways in which spheres of control can interact with each other.

- Overlap of Spheres.
- Nested Spheres.
- Disjoint Spheres.

The first case of overlap of two spheres is not possible as it implies that the same subtransaction is spawned by two parents, which is not allowed in the nested transaction model.

4.1 Nested Spheres

In this section we describe the approach proposed in this thesis for supporting nested spheres of control.

4.1.1 Dealing with Nested Spheres of Control

Not allowing nested spheres to form involves not allowing a descendent to retain a lock if an ancestor has a retain on that object. Thus if a parent retains a lock on a object then none of its children can retain a lock on that object, alternatively, if one

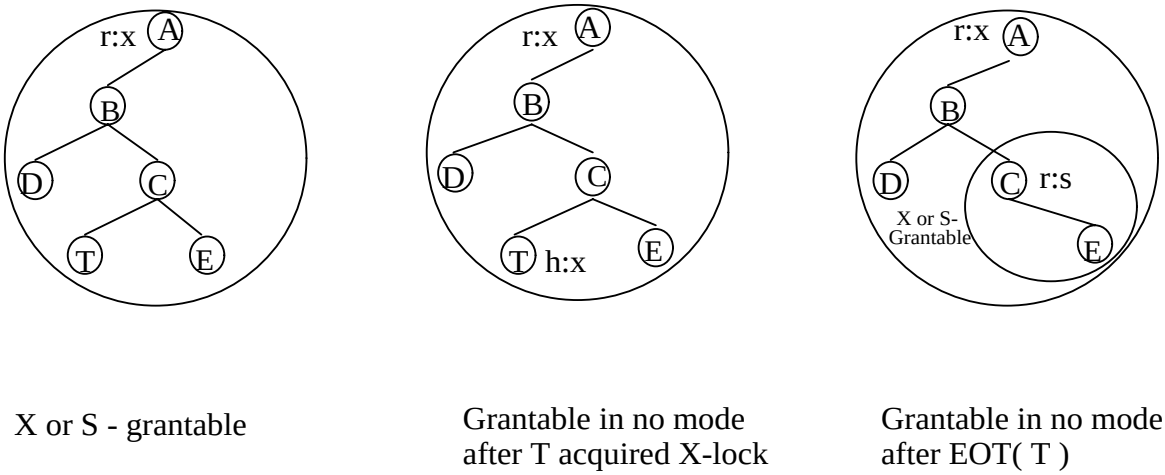


Figure 4.1. Example of Nested Spheres

of the child transactions retains a lock, then none of its superiors can retain a lock on that object. To implement this, we will need information stating that one of the superiors or children is retaining a lock. This information is not difficult to manage, as we already have retainsub field, indicating that, if any of its subordinates retain a lock on an object. To restrict a child from retaining a lock on an object (on which one of its superiors retains a lock), we might have to add one more field, retainpar, which has to be set at each child when a superior retains a lock on an object. Thus we might have to do a lot of update of holdmodes even when a lock is inherited.

The second problem is the enforcement of serializability. In figure 4.1 , transaction T has acquired an X-lock on an object and after commit of transaction T, transaction C retains that lock. If we don't allow transaction C to retain that lock (as transaction A is already retaining a lock on same object), then we are not allowing the inner sphere of control to form. Thus now transactions D and E are in same the sphere of control and there is no way we can distinguish between them (according to basic locking rules, transaction D should be allowed access to that object in no mode and transaction E should have access in any mode). Hence if transaction D requests, it can acquire a lock in any mode on that particular object and thus writing an

independent copy. If both D and C commit, there will be two different values for the same item(lost update problem). This needs to be avoided, requiring support for nested spheres of control.

4.1.2 Supporting Nested Spheres of Control

Consider our previous example (shown in figure 4.1). It sketches the case where an S-lock is granted in a transaction hierarchy whose root retains the lock in X-mode. While this hierarchy was 'X or S-grantable' before S lock was given to subtransaction T, it remains only 'S-grantable' thereafter. After commit(T), a part of this hierarchy becomes 'X or S-grantable' again. Thus we have nested spheres of control.

In figure 4.1, transaction D is in the sphere of control of A. Thus by checking the ancestors, transaction D will find itself in sphere of control of A, and as A retains a X-lock it will find that it can acquire a lock in any mode (if we do not keep extra information or do exaustive search). But if we take a closer look, transaction C also retains a lock on same object and has formed a new sphere of control within the original sphere of control. Thus by basic locking rules transactions in the inner sphere of control should get a lock in any mode, but transactions outside should get lock on that object only in S-mode (as retain of transaction C is in shared mode). Thus transactions D should be allowed to take a lock on that object only in S-mode. This is contradictory to what we stated earlier. We resolved this situation by allowing the most restrictive mode to prevail and store it in grantmode field. In other words, in this case we should allow transaction D to take a lock on that object only in S-mode and **not** in X-mode.

As soon as the nested sphere of control is formed, grantmode permissions in all its superiors (thus also including superiors belonging to outer sphere of control) should be changed from any mode to shared mode only. Thus allowing transaction D to take lock only in shared mode even though it belongs to a sphere of control.

The case we considered had outer sphere with retain in exclusive mode and inner in shared mode. There are different combinations possible:

- 1. Inner sphere with retain in exclusive mode : In this case transaction D of figure 4.1 should not be allowed to take lock in any mode on that object. Thus we have to change the grantmode field at all the superiors from any mode to no mode.

- 2. Outer sphere with retain in shared mode and inner with retain in shared mode : In this case again transaction D should not be allowed to take the a lock in any mode, but should be restricted to shared mode only. This again can be achieved by placing S in the grantmode field of all superiors.

- 3. Outer sphere with retain in shared mode and inner with retain in exclusive mode : Here transaction D should not have any access to that object, which can be done by placing X in grantmode field of all superiors.

4.2 Disjoint Spheres of Control

We know that more than one transaction can acquire the S-lock on an object. When these transactions commit, their S-lock is inherited by the immediate parent. A sphere of control is formed and includes all its descendants . In figure 4.2, transactions G, F, and Q acquire the S-lock on an object. When they commit, they form disjoint spheres of control. Sphere of control of transaction C includes C itself and transaction E, similarly that of D includes D and H, and that of P includes P and R. Thus here we have three disjoint spheres of control. Now if transaction E and H check the ancestors, they find themselves in sphere of control and may acquire a X-lock on same object (according to basic locking rules no transaction can acquire a X-lock, which reduces concurrency). This is in correct, and should be prevented. There are three issues concerning disjoint spheres of control :

- Do not allow disjoint spheres to form.
- Support disjoint spheres with additional information.
- Merging of disjoint spheres.

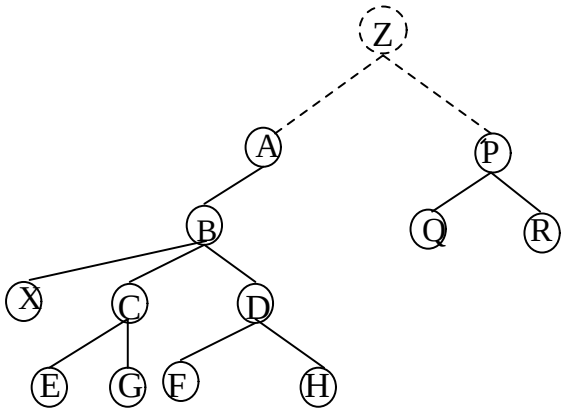
4.2.1 Not Supporting Disjoint Spheres

One solution to this problem, is to somehow restrict the number of spheres to one. In this case, all the subtransactions inside the sphere of control can acquire lock in any mode on that object and no transactions outside can do it. To achieve this, we need to know if any transaction which was holding S-lock on that particular object has already retained it. This approach allows only the transaction which committed first to retain the lock on that object. All other transactions are not allowed to retain a lock on that object, thus not forming spheres of control. This reduces concurrency, as all other transactions (which are not allowed to retain) cannot acquire an exclusive lock on that object.

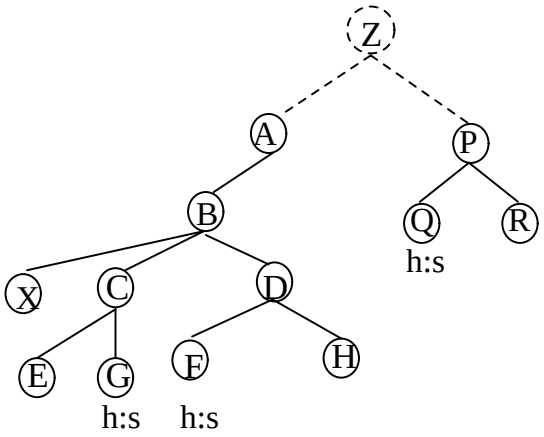
Another approach is not to allow more than one transaction to acquire a S-lock on an object, i.e. similar to exclusive locks, only one transaction is allowed to get a S-lock on an object at a time. Thus only one sphere of control will be formed, and transaction in this sphere can only get lock in any mode and those outside won't be allowed to acquire a lock in any mode. This effectively reduces concurrency among siblings and transactions as a whole. Thus we need to find a solution so that objects can be shared as usual.

4.2.2 Supporting Disjoint Spheres

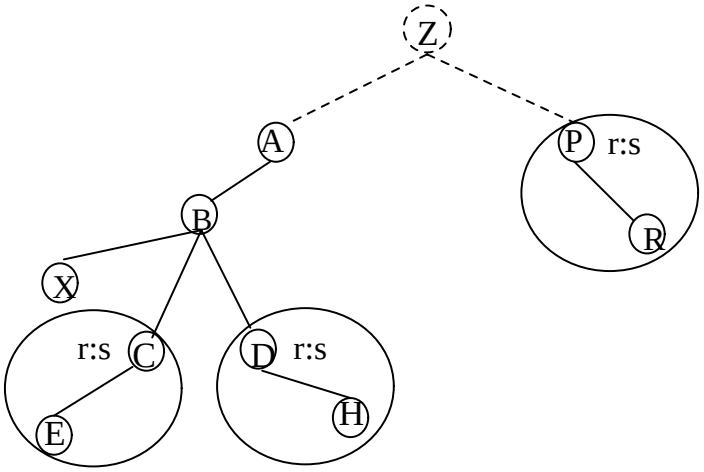
We have seen that to allow for high degree of concurrency we need to support disjoint spheres of control. While supporting disjoint spheres, we need to control the access of an object by transactions in different spheres. One approach is to allow disjoint spheres to be formed and allowing only one of them to upgrade it to an X



X or S - grantable



S-grantable after T, F, Q acquired S-lock



S- grantable after EOT(G and F)

Figure 4.2. Example of Disjoint Spheres

mode. Of course, deadlocks arise when two or more transactions try to upgrade to an X lock. *(We need to emphasize a point here, in the algorithm presented we do not allow transactions in a disjoint sphere to upgrade the lockmode to X: reason being the fact that in our implementation all subtransactions modify a global copy of the object. If each subtransaction can get a local copy of an object, for both S and X mode, we can allow one of the subtransactions in the disjoint spheres to upgrade to X mode.)* Take for example figure 4.2, in this case any one of the transactions E, H, or R can now take a X-lock on that object. After the lock has been acquired, all other transactions in the system are not allowed access to that object. Also suppose transaction E acquired the X-lock, after it commits, transactions in its sphere are allowed access in any mode, but transactions outside are not allowed access in any mode. This is similar to *non-repeating reads* in flat transactions; in our case as the subtransaction commits and is not going to read that object again (if any other subtransaction of the parent wants to read that object it has to issue an explicit request), we can allow transaction E to acquire an X-lock on that object.

Thus we can see that having retain at the parent node does not guarantee that subordinates can get a lock in any mode on an object, but it essentially says that it may get a lock in any mode. The actual mode in which it can acquire depends on the transactions outside.

4.2.3 Merging Disjoint Spheres

Another issue is to try and see whether disjoint spheres can be merged. Merging of two or more disjoint spheres is not possible in all situations. Consider figure 4.2 again, here after the transactions C and D have retained the lock there will be two spheres of control which includes transactions C, E and D, H respectively. Here merging this two spheres is not possible as transaction B has one more child X (or C and D have a sibling), which is not retaining the lock on the same object and

thus does not have exclusive rights on that object. Merging in this case will give transaction X exclusive rights to that object.

But if there was no transaction X or all the children of B had retained the shared lock, then we can definitely merge both the spheres. Merging two or more spheres will also include the immediate parent of all retaining transactions in the sphere, but this does not matter as the parent is suspended.

CHAPTER 5
IMPLEMENTATION

It has been observed that the capabilities of record-oriented data models are limited in capturing the complex structural relationships and the behavioral properties of object in advanced application domain such as engineering, scientific, statistical and military applications. Many semantic models [11] and object-oriented (O-O) data models [12] have been developed to alleviate the limitation of record-oriented data models. The O-O models provide a variety of modeling constructs, which simplify the task of modeling complex data. The main features of complex data models are:

- They support the unique identification of objects by system assigned object identifier.
- They support abstract data types and allow complex objects to be defined in the form of hierarchies.
- They allow encapsulation of structure and behavior within objects and classes.
- They allow definitions of hierarchies and the inheritance of structural and behavioral properties among classes in these hierarchies.

We further extend our work into the Object-Oriented paradigm. In a hierarchy similar to the Object-Oriented design schema, the nested transaction model can be easily adopted. A prototype O-O DBMS called **Zeitgeist** or *free spirit* was used for implementing the nested transactions described in earlier sections [9]. This prototype

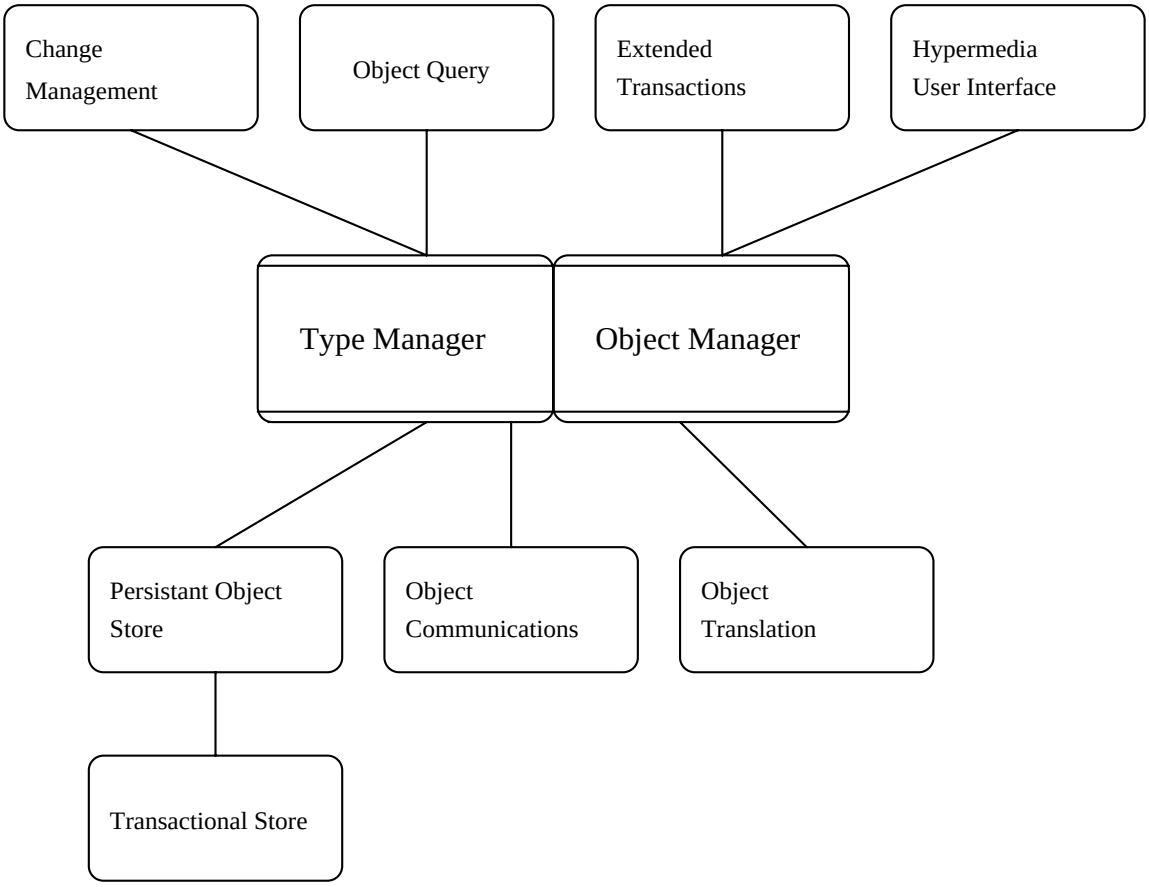


Figure 5.1. Zeitgeist Modules

has been acquired from Texas Instruments Inc., Dallas and was the basis for our implementation.

5.1 Zeitgeist

In this section, we briefly describe the execution model, particularly the Transaction Manager of Zeitgeist. This O-O DBMS originally used two kinds of storage structure for its persistent storage. The Open O-O database architecture, on which Zeitgeist is based, is shown in figure 6.1.

Zeitgeist, has the following components. It has a Persistent Object Store (POS), an Object Management System (OMS), a primitive OQL, and a two-level flat Transaction Manager.

C++ been used to define the O-O conceptual schema and has also been used to manipulate the internal object structure. To make objects persistent the language has been slightly modified to P-C++ which is pre-processed to conform to our needs.

The persistent storage in the original Zeitgeist was either file-based structure on the Unix platform, or Oracle relational DBMS. File based storage structure does not support concurrency control or recovery. At University of Florida an interface was developed to use Ingres as a storage manager for Zeitgeist [10]. The multi-level transaction model consists of the Zeitgeist transaction manager and the transaction manager provided by the Ingres. The relational coupling provided by the storage of translated objects from Zeitgeist is unique. A well defined interface on two levels of abstractions is available. The first is Zeitgeist's own transaction manager, and the other level of abstraction is the transaction manager of underlying storage manager, which is Ingres in our case.

We extend the 'flat' transaction model of Zeitgeist. As a result, some of the implementation decisions are influenced by the existing data structure and the algorithm. Zeitgeist consists of the following modules:

- Object Manager.
- Transaction Manager
  - Lock Manager.
- Persistent Object Store (Ingres).

The Zeitgeist transaction manager supports a flat transaction model, and it supports functionalities such as:

- Begin Transaction.
- End Transaction (Commit Transaction).

- Abort Transaction.

Since Zeitgeist uses a multi-level transaction manager, recovery is available only at the lowest level (provided by Ingres).

5.2 Original Implementation

The interface to persistent object store uses three tables *viz.* Groups, Value, and Refto. These three tables are created and initialized using scripts, and can be accessed directly for debugging purposes. The first relation, Groups contains the information for each storage group and the time when it was last modified. The second relation, Value, is the table where the translated form of object instance is stored. The last of the relations, Refto, is used to keep the external references of an object.

5.2.1 Begin Transaction

In Zeitgeist's implementation of transactions, connection to Ingres is done during the construction of an object of class Zeitgeist. All read queries in Zeitgeist transaction are translated to equivalent Ingres queries. Update queries require that object have to be read first (thus brought into process memory) and then the object is modified in process memory. If the transaction is terminated 'normally' or if an explicit *commit transaction* command is given, all the modified objects are written back to persistent object store (POS). In case of explicit *abort transaction* command or 'abnormal' termination of a transaction (e.g. on account of a deadlock), none of the modified objects are written back to POS. Every object modified during a transaction's life span is also flagged, to be able to locate it at the termination of a transaction.

When an object is accessed, Zeitgeist fetches it from POS, and a lock acquired on that object. As a transaction can access an object more than once during its execution, the object manager in Zeitgeist maintains a data structure, encapsulated

object (EO), for every object accessed by a transaction, which keeps track of the lock mode in which the transaction holds that object. Using EO, access to an object is optimized by checking for the object in process memory first, and if the object is not in the memory, acquiring a lock to fetch it from the POS. Subsequent access to the same object is checked for the lock compatibility in EO. If the lock modes are compatible, transaction can access the object: otherwise (new mode > old mode), the transaction has to acquire a fresh lock on that object (upgrade lock). Thus, in no case does the transaction has to request POS for that object, and in most cases transactions will not even have to request a lock from the transaction manager (for those object's already in its memory).

5.2.2 Commit Transaction

At 'normal' termination of a transaction or at an explicit *commit transaction* request by an application, a transaction is committed. At the commit of a transaction, each object in the process memory is scanned to check whether the object has been modified using the flag set earlier. If it is flagged, object is written back to POS. Also, all the locks held by the transaction (say T1) are released and a check is done to find out if other transactions are waiting for the locks on objects held by T1. If other transactions are waiting for a lock, they are released depending on the lock compatibility. Blocking for a lock on an object, is implemented using Semaphores. All the requests for the shared lock on an object block (if necessary) on a single semaphore and requests for an exclusive lock on an object block (if necessary) on a exclusive semaphore.

5.2.3 Abort Transaction

An 'abnormal' termination of a transaction or an explicit *abort transaction* request by a user application results in the transaction being aborted. As all the modification

of an object is done in process memory and nothing is written back to POS (until an explicit commit transaction or when buffers overflow). Thus, all that needs to be done at abort is to release all locks held by the aborting transaction and free all the blocked transactions (blocking on the locks held by the aborting transaction, depending on the concurrency control e.g. if lock being released is Shared then only one transaction requiring and exclusive lock is released from blocked state if no other transaction holds a Shared lock on that object).

5.2.4 Lock Manager

Zeitgeist transaction manager supports three lock modes:

- Read Only (RO),
- Shared (S),
- Exclusive (X).

A transaction is not required to acquire a lock on an object if its request is in read-only mode (This is due to the fact that RO mode is not upgradable). In the other two cases a transaction is required to acquire a lock on an object before accessing that object. Each object in Zeitgeist is assigned a object number (OID) and a storage group number (SG#) (user application has an option to specify its own storage group, if it does not, a default storage group is assigned).

The data structure used by the lock manager is a Hash Table. Hashing is done based on OID and SG#, thus each pair (OID,SG#) hashes to a bucket (all versions of an object hash to same bucket as time stamp is not a hashing parameter). In case of collision, more than one such pair can hash to the same bucket. Each lock held on an object is a node in a particular bucket, thus in case of shared locks, more than one node can exist in a bucket. Also, transactions can hold locks on more than one object, thus nodes belonging to the same transaction (in same or different buckets) are also

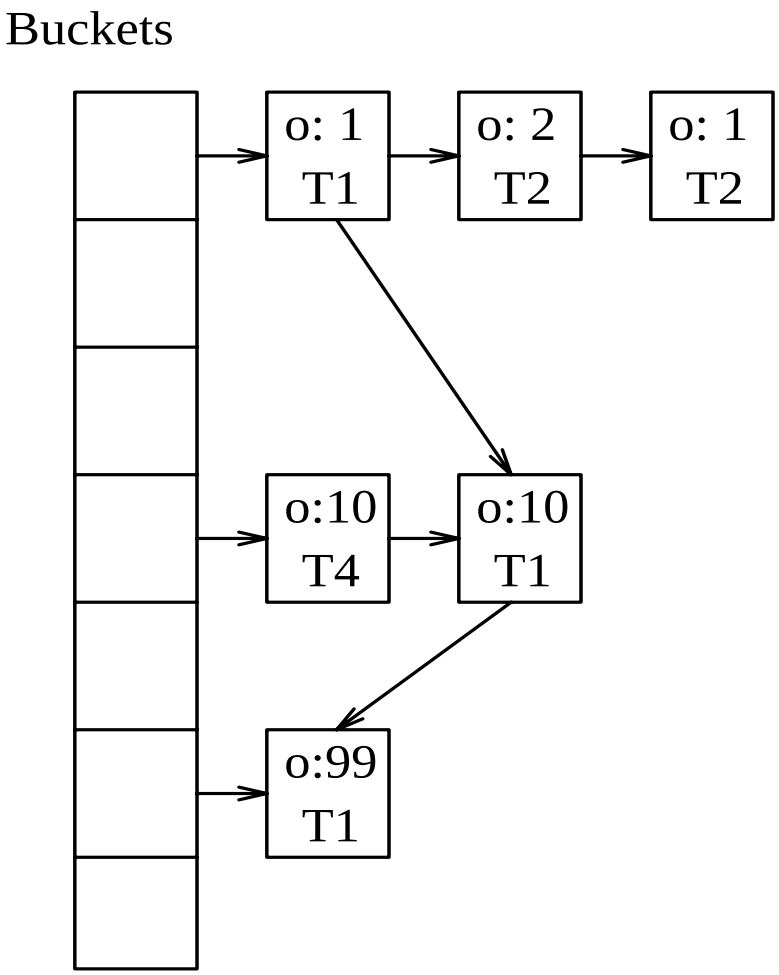


Figure 5.2. Example of Original Hash Table

linked in the hash table structure. Figure 6.2 shows the original implementation of the lock table. It can be seen that transaction T1 has acquired lock on three objects (OID's 1, 10, 99) and that nodes belonging to transaction T1 are also linked together (SG# is not shown).

5.3 Extensions

In this section, unless stated otherwise, the term transaction is used to refer to both subtransactions and top-level transaction. Firstly, two commands were introduced to support nested transactions: begin and end subtransaction. The connection

to Ingres, which was established at the time of construction of an object of type Zeitgeist (i.e. at the beginning of the application and not at begin transaction command), is not changed for top-level transaction (done at the declaration of an object of type Zeitgeist) but for subtransactions it is established at the time of begin subtransaction command. Thus each transaction has its own connection to Ingres and hence an application may have multiple connections to Ingres (This was implemented by using the CONNECT and SESSION clauses of Ingres.). An Ingres session is disconnected at the termination of a transaction (commit/abort), using the DISCONNECT clause.

Subtransactions are implemented using UNIX threads, making each subtransaction a thread of control. Use of UNIX threads offers the following advantages:

- Overhead for spawning a thread is considerably less compared to forking a process.
- Process memory of parent is not copied (fork copies the parent process memory).

SunOS 4.1.2 supports threads as procedures, requiring a scheduler to schedule each thread as required by application. Also, as we have implemented nested transactions for sibling concurrency, a parent has to be suspended until all the children have terminated. This was accomplished by message receive and message send procedure calls from threads library.

5.3.1 Object Manager

In the original implementation, the object manager will acquire a lock and fetch an object from the POS and install it in process memory only in case of first time request. All subsequent requests, for that object and by that transaction, will check EO for that object and decide about the access privileges (i.e. should upgrade request be generated ? or can object be accessed directly ?).

Threads are considered to be a single process (they have same PID), thus top-level transaction and all the subtransactions are considered to be a single process. Now, if parent transaction accesses an object or if any transaction in an hierarchy accesses an object (thus getting object in the process memory), all other transactions can access that object (due to the fact that every attempt to access an object, initially checks for an EO for that object in process memory). Thus transactions without any access privileges on an object can access it. To prevent this from happening, we forced each access of an object to obtain a lock (or access permission) from the transaction manager. As each lock entry along with other information has a TID (which is unique for each transaction/subtransaction) of holder of the lock, the above problem was avoided.

5.3.2 Lock Manager

Unlike in flat transactions where each node in the lock table corresponds to the lock held, in nested transactions nodes existing in the lock table may not correspond to the lock held. In our implementation, if a transaction acquires a lock on an object, then all its superiors will also (along with the transaction itself) be represented by a node in lock table. Nodes corresponding to transactions (along a path from itself to root) not holding a lock on an object give us information regarding holdsub, retainsub, retain, and grantmode in their part of the subtree.

A straightforward implementation of the above increases the number of nodes in a bucket (by a number depending on the depth of the tree). Also, the number of times the lock table is accessed increases by the same amount (each acquire and release would require us to update the information at all the superiors). Also, due to collision, more than one object can hash to same bucket, thus effectively increasing the search time. The number of nodes accessed in worst case can be given by:

$\sum_{i=1}^{m}$ ($\sum_{j=1}^{n}$ **(depth of tree * number of nodes with non-null hold field))**

Here, m is the number of objects hashing to the same bucket and n is the number of top-level transactions (tree is on object basis i.e. one tree per object accessed by a transaction).

We tried the following alternatives to improve efficiency:

- We tried to order the links in a hash table bucket i.e. child link comes before parent. But in doing so we are assuming that the parent is already present in the data structure, or we will have some overhead while inserting the superiors of a transaction (i.e. we simply cannot insert a node at the beginning of the linked list, but have to search for the child node and insert a parent node after the child node in the linked list). Maintaining a specific order in a linked list incurs additional overhead. Thus we tried a different approach as explained below.

- The underlying data structure was extended to an 'Anchored Hash Table' (AHT). An AHT will have an anchor node in each bucket for each object that hashes to that bucket. Thus, if more than one object hashes to a bucket, the bucket will have a linked list of anchors and each anchor in turn will have a linked list structure for that object.

Figure 6.3 shows the new data structure AHT. In the figure, the broken line represents the linked list of nodes belonging to the same transaction. Each anchor node also serves as a virtual node for a particular object. Now the number of nodes accessed in worst case can be given by:

$\sum_{j=1}^{n}$ **(depth of tree * number of nodes with non-null hold field)) + m**

where m and n have the same meaning as earlier.

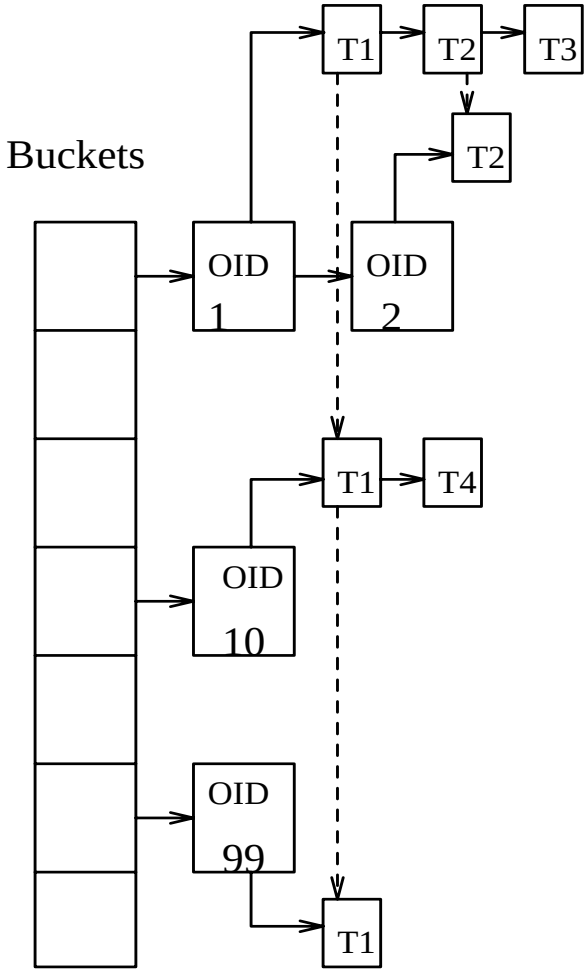


Figure 5.3. Example of Anchored Hash Table

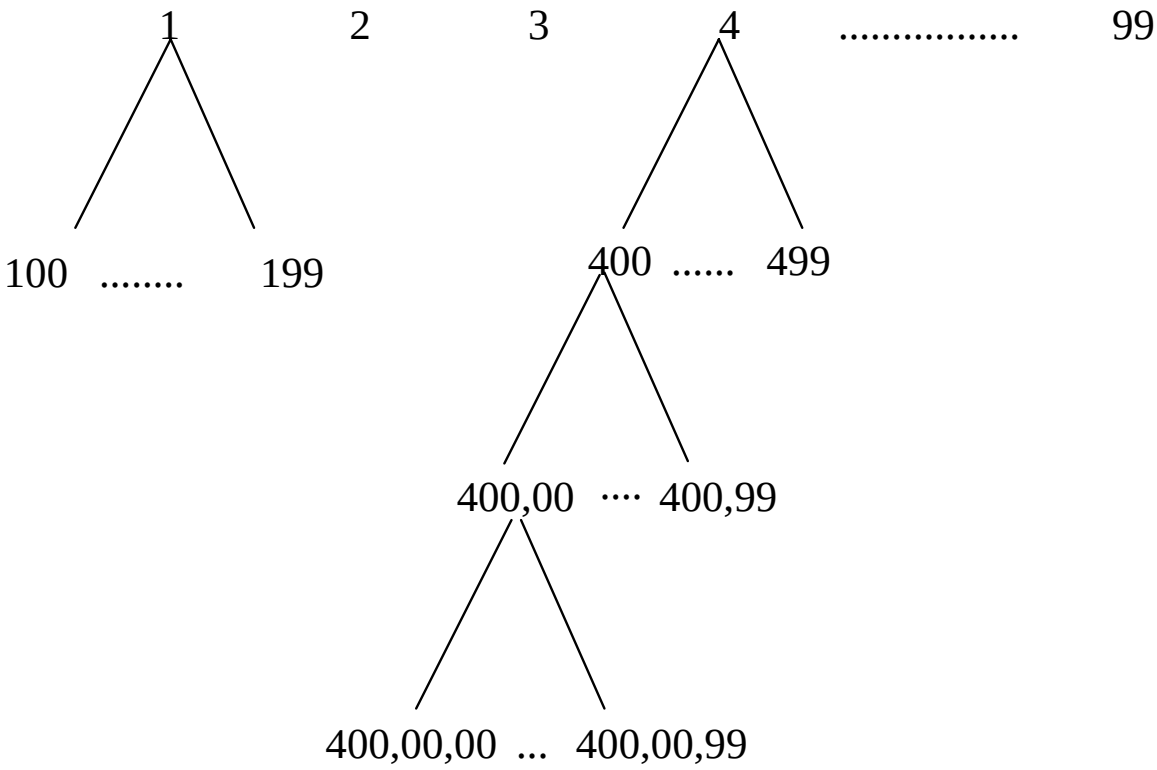


Figure 5.4. Example of TIDs

The access time was further reduced by a scheme, in which child TID was generated using the following formula:

child TID = (parent TID * 100) + number of existing children

Figure 6.4 gives an example of this numbering scheme. Thus, knowing a child TID we can calculate the TIDs of all the superiors and do all the required modifications to all the required links in a maximum of two passes over the linked structure for an object.

5.3.3 Transaction Manager

The transaction manager was modified according to the algorithms presented earlier. Figure 6.5 shows the overall data structure used in Zeitgeist transaction manager. The algorithms which were modified are: set lock, upgrade lock, and release lock.

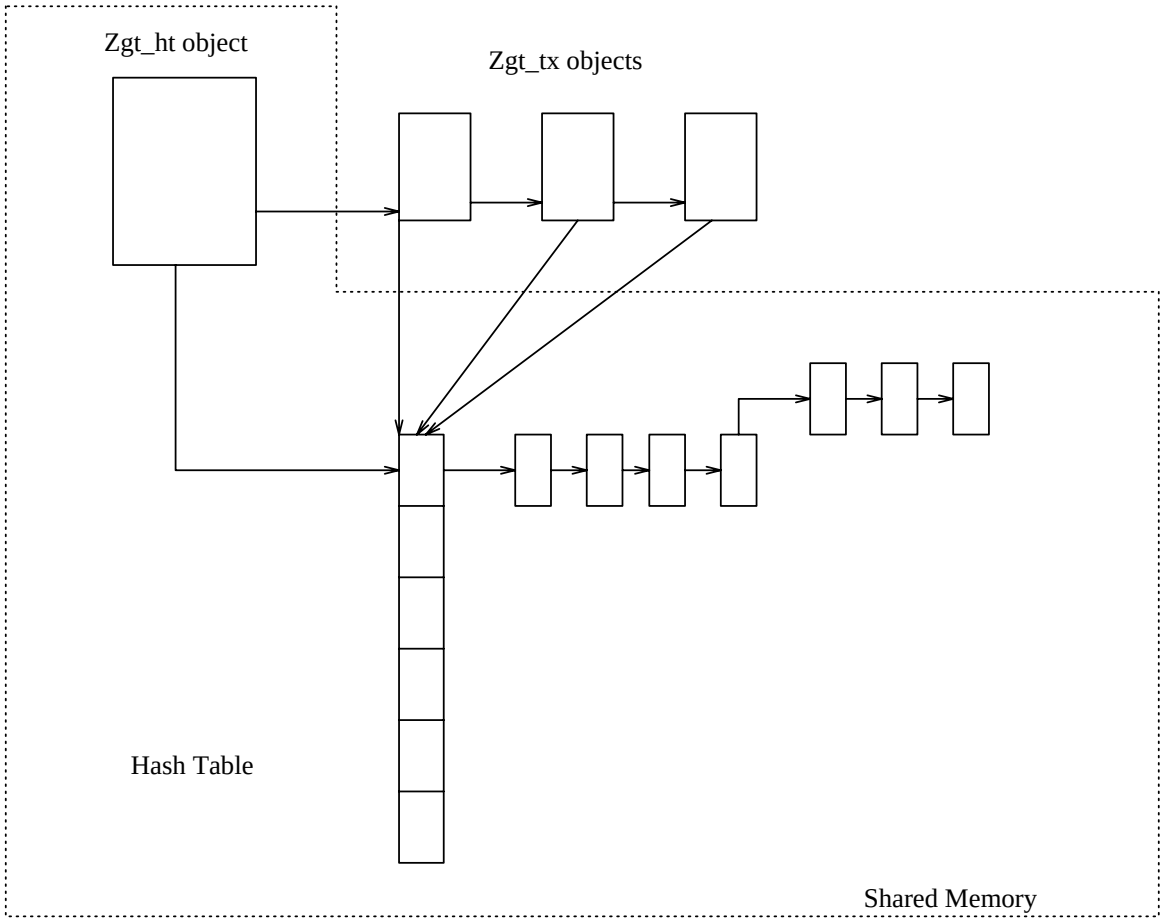


Figure 5.5. Zeitgeist Data Structure

Set Lock and Upgrade Lock

Set lock and the upgrade lock algorithms were modified according to the algorithms presented. When a lock requested for an object by a transaction is already held in a conflicting mode, the transaction is made to wait on that lock. This was originally implemented using semaphores. If the required lock is in shared mode, the transaction waits on a semaphore with all other transactions (which are waiting for the shared lock on same object). In the other case (exclusive mode), each transaction waits on a unique semaphore.

In extended Zeitgeist, even the transactions waiting for a shared lock on an object, cannot be blocked on a semaphore. This is due to the fact that, when the existing lock on that object is released, not all the transactions requiring a shared lock can acquire it (this comes from the sphere of controls phenomena). But they have to be selected on the basis of their relative position to lock releasing transaction (i.e. the requesting transaction has to be in the new sphere of control which was formed due to the release of an exclusive lock). Thus, even for shared lock mode each transaction has to wait on a unique semaphore (so that it can be released selectively). The maximum number of semaphores required earlier was the number of transactions in the system. In the extended version, maximum number of semaphore required is the total number of transactions (top-level + subtransactions) in the system.

Release Lock

Release lock algorithm was modified to incorporate the algorithm presented the in previous chapter. During the release of a lock, we have to free all the compatible lock requests blocked for a lock on that object. In nested transactions, along with compatibility we need to take the spheres of control into account. Thus, before releasing a transaction from the blocked state, we have to check for eligibility of that

transaction to acquire that lock. This is done according to the algorithms presented earlier.

Scheduling Options

As mentioned earlier, we use UNIX threads to implement each subtransaction. Threads in SunOS 4.1.2 are implemented as procedures and thus can be scheduled. Scheduling can be of two types: preemptive and non-preemptive. We have tried both time slicing (preemptive) and priority based scheduling (non-preemptive) of subtransactions.

Preemptive scheduling is the most appealing as it facilitates concurrent execution of threads (subtransactions). CONNECT and SESSION clauses of Ingres have been used to perform preemptive scheduling. Thus each subtransaction has its own private connection to communicate with Ingres. Each connection to Ingres is identified by a session number. Thus a circular buffer has been implemented to put into effect a session, for a transaction to be scheduled next (session number and thread ID are stored). As the CONNECT clause of Ingres cannot handle a second request before first is completed (as server detects that client is sending second request before first request is complete, and gives an error message) preemptive scheduling requires modification to Ingres.

Non-preemptive scheduling can be done using priorities in UNIX threads. Each thread is given a priority according to when we want it to complete relative to the other transactions. Thus we can have subtransactions scheduled in any order desired, at the cost of concurrency. That is, each transaction has to complete or be blocked before another transaction starts. Thus we block each subtransaction before commit and when all the children of a parent have reached this state, all the subtransactions are committed and the parent can continue. This does not demonstrate much

concurrency, but is sufficient to demonstrate the correctness and feasibility of nested transaction algorithms developed in this thesis.

CHAPTER 6
SUMMARY AND FUTURE WORK

In an active database, it is meaningful to execute rules using the same model as that used for transaction execution. Consistent with this philosophy, nested transaction have been proposed for guaranteeing ACID property for concurrent execution of rules. In this thesis, we have developed and implemented a lock-based mechanism for nested transactions for supporting concurrent rule execution as part of Sentinel – an active object-oriented DBMS currently under development at UF.

Employing the basic rules of locking and recovery [8, 6] requires that, in the worst case, every node in the nested transaction tree (as well as every node in other transaction trees) be searched. In this thesis, we have introduced a number of fields under the name *holdmodes* to avoid the exhaustive search for guaranteeing the ACID property. Using our approach, we have shown that the search can be limited to the path from the node requesting the lock to the root of the transaction tree containing that node. Furthermore, we have introduced the notion of a virtual node that serves as the root of all top-level transactions in the system. This further reduces search when a lock request is made for an object currently not used by any transaction in that tree.

The algorithm presented in this thesis handles two cases that are specific to nested transactions: the formation of nested spheres of control and ii) the formation of disjoint spheres of control. These are formed as subtransactions can progress at different rates (spawning subtransactions) and commit. Overlapping spheres of control cannot be formed as there is a unique parent for each transaction. We also modified the deadlock detection algorithm to work correctly with nested transactions.

The current implementation essentially extends the Zeitgeist algorithm for the "flat" transaction model. The structure of the lock nodes as well the hash table (lock table) has been extended. The system can currently be compiled either with flat transactions or with nested transactions.

This work forms the starting point for supporting an execution model for rule processing. Below, we list a number of issues that have not been addressed in this thesis but are currently being investigated as part of the Sentinel project:

- Current implementation supports only the immediate coupling mode; it needs to be extended to support deffered and detached coupling modes (both causally dependent and independent),
- Currently, subtransactions are executed concurrently without using user specified priority information. Prioritised execution of the rules need to be supported,
- Current implementation does not support recovery for subtransactions as Ingres is used as the storage manager. When this is ported to OpenOODB (which uses Exodus as the storage manager), recovery issues need to be addressed,
- Further optimization of current design. For instance, use of a different data structure than the one used currently may be more appropriate,
- Currently, only read-only, shared, and exclusive lock modes are supported. This needs to be extended to include intentional locks,
- Finally, develop algorithms for nested transactions not based on locks (e.g., timestamp based and optimistic).

APPENDIX A
LOCK ACQUIRING ALGORITHM

In this section we are presenting the algorithm for acquiring the locks. This algorithms are based on the design discussed in previous sections.

```
Set_lock()
{
   Search in the lock table of current transaction ;
  if (an entry is found)
    then
      current transaction owns the lock ;
      if (required lock mode <= owned mode)
          Lock already owned ;
      else
           if (requesting transaction retains an exclusive lock)
               Acquire lock ; nonupdate() ;
           else
             upgrade_lock();
  else
      recurse();
}


recurse()
{
  if (virtual node)
        switch(grantmode)
          case(N):
            Acquire lock; update(); break;
          case(S):
            if (required lockmode == S)
               Acquire lock(); update();
            else
              lock not available; wait or return error to caller;
            break;
          case(X):
            lock not available; wait or return error to caller;
  else  /* i.e. not a virtual node  */
      if (parent is retaining)
          switch(retain mode)  /* parents */
            case(S):
              switch(grantmode)
                case(N):
                case(S):
                  if (required lockmode == S)
```

```
                Acquire lock; update();
              else
                lock not available;
                wait or return error to caller;
              break;

           case(X):
             lock not available; wait or return error to caller;

        case(X):
          switch(grant mode)
             case(N):
              Acquire lock; update(); break;
            case(S):
              if (required lock == S)
                  Acquire lock; update();
              else
                 lock not available;
                 wait or return error to caller;
              break;
            case(X):
              lock not available;
              wait or return error to caller;
  else  /* parent is not retaining */
     if (parent is holding)
          switch(hold mode)      /* parents */
             case(X):
                switch(required lock mode)
                   case(X):
                     if (grantmode != N)  /* parents */
                        lock not available;
                        wait or return error to caller;
                     else
                         Acquire lock; update();
                     break;
                   case(S):
                     /* parents grantmode */
                     if ((grantmode==N) || (grantmode==S))
                         Acquire lock; update();
                     else
                        lock not available;
                        wait or return error to caller;
                     break;
                 break;
              case(S):
                switch(required lock mode)
                   case(S):
                          /* parents grantmode */
                     if ((grantmode==N) || (grantmode==S))
                         Acquire lock; update();
                     else
                         lock not available;
```

```
                            wait or return error to caller;
                        break;
                      case(X):
                        if (grantmode==N)
                             if (summation of retains and holds
                                 while traversing the tree upwards
                                 from requesting transaction ==
                                 values at virtual node)
                                  Acquire lock; update();
                             else
                               lock not available;
                               wait or return error to caller;
                        else
                            lock not available;
                            wait or return error to caller;
                        break;
                    break;
          else   /* parent is not holding */
            recurse(with parent node);
}


upgrade_lock()
{
  if (virtual node)
      if (((retainsub-retain_counter)==0) &&
         ((holdsub-hold_counter)==1))
          Acquire lock; non_update();
      else
        lock not available; wait or return error to caller;
  else
       if (lockmode != N) hold_counter++;
       if (retain != N)
           retain_counter++;
           switch(retain mode)
               case(X):
                 if ((holdsub==1) && (retainsub==0)
                     && (retain_counter==1) && (hold_counter==0))
                      Acquire lock; non_update();
                 else
                     lock not available;
                     wait or return error to caller;
                 break;
               case(S):
                 upgrade_lock(with parent node);
                 break;
       else
           if (lockmode != N)
               switch(lockmode)
                  case(X):
                    if ((holdsub==1) && (retainsub==0) &&
                        (retain_counter==0) && (hold_counter==1))
```

```
                        Acquire lock; non_update();
                    else
                       lock not available;
                       wait or return error to caller;
                    break;

                   case(S):
                     upgrade_lock(with parent node);
            else
               upgrade_lock(with parent node);
}



update()
{
  for (all superiors of requesting transaction)
     {
       holdsub = holdsub + 1;
       if (required lockmode>grantmode) /*grantmode of superior*/
          grantmode = lockmode;
     }
}


non_update()
{
   for (all superiors of requesting transaction)
     {
       if (required lockmode>grantmode) /*grantmode of superior*/
          grantmode = lockmode;
     }
}
```

APPENDIX B
ALGORITHM FOR ABORT

In this section we are presenting the algorithms for releasing (aborting the transaction) the locks. This algorithms are based on the design discussed in previous sections.

```
abort(tr_node)
{
 /* find the sphere of control, inner most */
 if (any one of the superiors of aborting transaction retains
     a lock on same object)
     switch(retain mode) /* of inner most sphere */
       case(X) :
         RSupdate() ;
         if (aborting transaction was holding a lock)
             switch(hold mode) /* aborting transaction */
               case(X) :
                 Xretain() ;
                 break ;
               case(S) :
                 Sretain() ;
                 break ;
               case(RO) :
                 ROretain() ;
                 break  ;
         else
             Else() ;
         break ;
       case(S) :
         RSupdate ;
         if (aborting transaction was holding a lock)
             switch(hold mode) /* aborting transaction */
               case(X) :
                 Xretain() ; SSupdate() ;
                 break ;
               case(S) :
                 Sretain() ; SSupdate() ;
                 break ;
               case(RO) :
                 ROretain() ;
                 break  ;
         else
             Else() ;
         break ;
 else   /* aborting transaction was not in any sphere */
     if (aborting transaction was holding)
         switch(hold mode) /* aborting transaction */
           case(X) :
             decrement hs field of all superiors of aborting
             transaction ;
             change the grantmode of all superiors to -- ;
```

```
 /* as aborting transaction was holding in X-mode and was
    not in any sphere */
              break ;
            case(S) :
              decrement hs field of all superiors of aborting
              transaction ;
              at each superior
               {
                if (hs + rs == 0)
                  then
                    change grantmode to -- ;
                  else
                    change grantmode to S ;
               }
              break ;
            case(RO) :
              decrement hs field of all superiors of aborting
              transaction ;
              at each superior
               {
                if (hs + rs == 0)
                  then
                    change grantmode to -- ;
                  else
                    change grantmode to S ;
               }
              break ;
     else
         Else() ;
}


Xretain()
{
 decrement hs field of all superiors of aborting transaction ;
 at each superior until the inner most sphere
 {
  if (rs == 0)
    then
      change grantmode from X to -- ;
    else
      change grantmode from X to S ; /* aborting transaction was
         holding a X-lock, thus there can only be a r:s
         outside its sphere */
 }
}


Sretain()
{
 decrement hs field of all superiors of aborting transaction ;
 at each superior until the inner most sphere
```

```
 {
  if (hs + rs == 0)
      change grantmode from X to -- ;
    else
      if (aborting transaction was retaining in S mode)
          do nothing ! ;
        else
          if (aborting transaction was retaining in X mode)
              change grantmode to S ;
            else
              do nothing ! ;
 }
}


ROretain()
{
 decrement hs field of all superiors of aborting transaction ;
 at each superior until the inner most sphere
 {
  if (hs + rs == 0)
      change grantmode from X to -- ;
    else
      if (aborting transaction retained in S-mode and rs > 0)
          do nothing ! ;
        else
          if (aborting transaction had retained in X-mode
              and rs > 0)
              change the grantmode to S ;
            else
              do nothing ! ;

 }
}


RSupdate()
{
 if (aborting transaction was retaining)
   then
    decrement rs field of all superiors of aborting transaction ;
}

Else() /* aborting transaction was not holding but retaining  */
{
 if (aborting transaction was retaining a lock)
     switch(retain mode)  /* aborting transaction */
       case(X) :
         decrement rs field of all superiors of aborting
         transaction ;
         at each superior
          {
```

```
            if (rs > 0)
              then
                change grantmode from X to S ;
              else
                change grantmode from X to -- ;
            }
          break ;
        case(S) :
          decrement rs field of all superiors of aborting
          transaction ;
           at each superior
           {
            if (rs + hs > 0)
              then
                do nothing ! ;
              else
                change grantmode from S to --  ;
           }
          break ;
}
```

APPENDIX C
ALGORITHM FOR COMMIT

In this section we are presenting the algorithm for releasing the locks. This algorithms are based on the design discussed in previous sections.

```
inherit_lock()   /* commit transaction */
{
  /* at parent of committing transaction */
  if ((retain==N) && (retainsub==0))
    {
       switch(grantmode)
           case(X):
             if (virtual transaction)
                 holdsub = 0; grantmode = N;
             else
                 holdsub = 0; grantmode = N;
                 if (committing transaction was holding or
                     retaining)
                    retain = X;
                 if (committing transaction was holding)
                    both_update();
                 else
                    ri_update();
            break;
           case(S):
             if (committing transaction was holding)
                holdsub = holdsub - 1;
             if (holdsub == 0)
                 if (virtual node)
                     grantmode = N;
                 else
                     grantmode = N;
                     if (committing transaction was holding or
                         retaining)
                          retain = (retain > S? retain:S);
                     if (committing transaction was holding)
                          both_update();
                     else
                          ri_update();
             else  /* holdsub != 0 */
                 if (not virtual transaction)
                      if (committing transaction was holding or
                          retaining)
                          retain = (retain > S? retain:S);
                      if (committing transaction was holding)
                          both_update();
                      else
                          ri_update();
       return(0);
     }
   if ((retain!=N) && (retainsub==0))
     {
```

```
        switch(grantmode)
             case(X):
               holdsub = 0; grantmode = N;
               if (committing transaction is holding or retaining)
                    retain = X;
               if (committing transaction is holding)
                    hd_update();
               break;
             case(S):
               if (committing transaction is holding)
                   holdsub = holdsub - 1;
               if (holdsub == 0)
                   grantmode = N;
               if (committing transaction is holding or retaining)
                   retain = (retain > S? retain: S);
               if (committing transaction is holding)
                   hd_update();
               break;
        return(0);
       }
     if ((retain != N) & (retainsub != 0))
       {
          switch(grantmode)
             case(X):
             case(S):
                if (committing transaction is retaining)
                     retainsub = retainsub - 1; rd_update();
                if (committing transaction is holding)
                     holdsub = holdsub - 1; hd_update();
                if (committing transaction is holding or retaining)
                      retain = grantmode;
                if ((retainsub == 0) && (holdsub == 0))
                      grantmode = N;
          return(0);
       }
    if ((retain == N) && (retainsub != 0))
      {
         if (committing transaction is not retaining)
              ri_update():
         else
              retainsub = retainsub - 1;
         if (committing transaction is holding or retaining)
              retain = grantmode;
         if ((retainsub == 0) && (holdsub == 0))
              grantmode = N;
         return(0);
     }
}

both_update()
{
  for (all superiors of committing transaction)
```

```
        holdsub = holdsub - 1;
        retainsub = retainsub + 1;
}

ri_update()
{
  for (all superiors of committing transaction)
        retainsub = retainsub + 1;
}

rd_update()
{
  for (all superiors of committing transaction)
        retainsub = retainsub - 1;
}

both_update()
{
  for (all superiors of committing transaction)
        holdsub = holdsub - 1;
}
```

REFERENCES

[1] S. Chakravarthy et al. HiPAC: A Research Project in Active, Time-Constrained Database Management, Final Report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, Cambridge, MA, Aug. 1989.

[2] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in Database Systems. *Communications of the ACM 19*, 10(11), Nov. 1976.

[3] Anon et al. A Measure of Transaction Processing Power, Datamation, April issue, April 1985.

[4] J. Gray and A. Reuter. Transaction processing: Concepts and techniques. pages 202–204, 1993. (Chapters 4 for Nested Transactions).

[5] T. Haerder and A. Reuter. Principles of transaction oriented database recovery. *ACM Computing Surveys*, 15(4):287–318, Dec 1983.

[6] T. Haerder and K. Rothermel. Concurrency Control Issues in Nested Transactions . IBM Research Report RJ5803, Aug. 1983.

[7] M. Hsu, R. Ladin, and D. McCarthy. An Execution Model for Active Data Base Management Systems. In *Proceedings 3rd International Conference on Data and Knowledge Bases*, Jun. 1988.

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

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

[10] A. Sharma. On extensions to a passive dbms to support active and multi-media capabilities. Master's thesis, Database Systems R&D Center, CIS Department, University of Florida, E470-CSE, Gainesville, FL 32611, June 1992.

[11] S. Y. W. Su. "SAM*: A Semantic Association Model for Corporate and Scientific-Statistical Databases". *Information Sciences*, 29:151–199, 1983.

[12] S. Y. W. Su, V. Krishnamurthy, and H. Lam. "An Object-Oriented Semantic Association Model (OSAM*)". *Theoretical Issues and Applications in Industrial Engineering and Manufacturing*, pages 242–251, 1989.