In: The VLDB Journal, Vol. 2, No. 1, 1993, pp. 39-74.

Concurrency Control Issues in Nested Transactions

Prof. Dr. Theo Härder
University of Kaiserslautern
D-6750 Kaiserslautern
Fed. Rep. of Germany
Phone: ++49-631-205-4031
Fax: ++49-631-205-3558
e-mail:haerder@informatik.uni-kl.de

Prof. Dr. Kurt Rothermel
University of Stuttgart
D-7000 Stuttgart 80
Fed. Republic of Germany
Phone: ++49-711-7816-434
Fax: ++49-711-7816-424
e-mail:rothermel@informatik.uni-stuttgart.de

Index Terms: Nested Transactions, Concurrency Control, Locking, Object Hierarchies

Concurrency Control Issues in Nested Transactions

Theo Härder
University of Kaiserslautern
D-6750 Kaiserslautern
Fed. Rep. of Germany

Kurt Rothermel
University of Stuttgart
D-7000 Stuttgart 80
Fed. Rep. of Germany[1]

Abstract

The concept of nested transactions offers more decomposable execution units and finer grained control over concurrency and recovery than 'flat' transactions. Furthermore, it supports the decomposition of a 'unit of work' into subtasks and their appropriate distribution in a computer system as a prerequisite of intra-transaction parallelism. However, to exploit its full potential, suitable granules of concurrency control as well as access modes for shared data are necessary.

In this paper, we investigate various issues of concurrency control for nested transactions. First of all, the mechanisms for cooperation and communication within nested transactions should not impede parallel execution of transactions, neither among parent and children nor among siblings. Therefore,a model for nested transactions is proposed allowing for effective exploitation of intra-transaction parallelism. Starting with a set of basic locking rules, we introduce the concept of 'downward inheritance of locks' to make data manipulated by a parent available to its children. To support supervised and restricted access, this concept is refined to 'controlled downward inheritance'.

The initial concurrency control scheme was based upon S-X locks for 'flat', non-overlapping data objects. In order to adjust this scheme to the needs of practical applications, a set of concurrency control rules is derived for generalized lock modes described by a compatibility matrix. Moreover, these rules are combined with a hierarchical locking scheme to improve selective access to data granules of varying sizes. After having tied together both types of hierarchies (transaction and object hierarchies) it could be shown how 'controlled downward inheritance' for hierarchical objects may be achieved in nested transactions. Finally, problems of deadlock detection and resolution in nested transactions are considered.

1. This research war carried out while the authors were visiting scientists at the IBM Almaden Research Center, San Jose, California.

1. Introduction

When multiple users access a database simultaneously, their data operations have to be coordinated in order to prevent incorrect results and to preserve the consistency of the shared data. This activity is called concurrency control and should provide each concurrent user with the illusion that he is referencing a dedicated database. The classical transaction concept [Eswaran76] defines a transaction as the unit of concurrency control, that is, the database management system (DBMS) has to guarantee **isolated execution** for an entire transaction. This implies that its results derived in a multi-programming environment should be the same as if obtained in some serial execution schedule. Other important transaction properties are *atomicity*, *consistency*, and *durability* as defined in [Härder83]. In a DBMS, the component responsible for achieving these properties is transaction management which includes concurrency control as a major function.

In current DBMSs, transaction management is typically designed with a single level control structure; its implementation is optimized to execute short transactions with only a few data references [Anon85]. Two-phase locking is, by far, the most common method for controlling concurrency among transactions and has been accepted as a standard solution [Bernstein81, Gray78]. When running on a centralized DBMS, transaction granularity as well as locking protocols usually obtain satisfactory performance; for high performance transaction systems, special concurrency control methods are considered to be mandatory to increase the level of parallelism [Gawlick85, Reuter82], however, without requiring changes to the transaction concept.

When executing more complex transactions involving, for example, sequences of joins and sort operations in a relational DBMS, it turns out that single level transactions do not achieve optimal flexibility and performance. Especially in distributed systems, it is highly desirable to have more general control structures supporting reliable and distributed computing more effectively. Major concerns are more decomposable and finer grained control of concurrency and recovery. As a solution to these problems, the concept of nested transactions was proposed by Moss [Moss85] where single level transactions are enriched by an inner control structure. Such a mechanism allows for the dynamic decomposition of a transaction into a hierarchy of subtransactions thereby preserving all properties of a transaction as a unit and assuring *atomicity and isolated execution* for every individual subtransaction. As a consequence, subtransactions may be distributed in a system among various (processor) nodes performing subtasks of the entire transaction. These prime aspects of nested transactions - *decomposition of a 'unit of work' into subtasks and their distribution* - lead to the following advantages in a computing system and, in particular, in a distributed DBMS:

Intra-transaction parallelism

The larger a transaction is, the more inherent parallelism may be anticipated during its execution. To take advantage of this inherent concurrency in the application, suitable granules of concurrency control as well as access modes (e.g. locking modes) are necessary. In environments enabling parallel execution, the nested transaction concept embodies an appropriate control structure to support supervised and, therefore safe intra-transaction concurrency, thereby increasing efficiency and decreasing response time.

Intra-transaction recovery control

An uncommitted subtransaction can be aborted and rolled back without any side-effects to other transactions outside its hierarchy. Hence, the concept of nested transactions contributes to a considerable refinement of the scope of in-transaction UNDO as compared to single level transactions where UNDO-recovery necessarily yielded the state of 'begin of transaction' (BOT). It may be further refined by adding an appropriate savepoint concept to nested transactions [Härder87, Rothermel89].

Explicit control structure

When parallel and asynchronous activities are to be coordinated for a single 'unit of work' (from an external point of view), the introduction of a powerful explicit control structure allowing for the delegation of pieces of work and their atomic execution appears to be mandatory. Such a structure will greatly reduce the complexity of programming and enhance the reliability of transaction processing.

System modularity

Subtransactions facilitate a simple and safe composition of a transaction program whose modules may be designed and implemented independently. This system modularity serves other design goals as well: encapsulation (information hiding), failure limitation, and security.

Distribution of implementation

The concept of the nested transaction supports the implementation of distributed algorithms by a flexible control structure for concurrent execution. Distribution of data and processing, in turn, have a major impact on overall efficiency, in terms of both cost-effective use of hardware (special processors, I/O devices) and responsiveness. Distribution also affects availability (replication of data). Hence, the robustness of the system may be improved in various ways.

In a centralized DBMS, nested transactions have some uses, however, they do not exploit their full potential due to the lack of resources. An obvious advantage is the clearer control structure for the execution of complex transactions supporting the design of more reliable programs. It also allows for the isolated rollback of an uncommitted subtransaction in the case of forced abort or transaction failure. When serializability of transactions controlled by strict 2-phase locking protocols (or equivalent methods) is required, neither lock granules nor lock duration are affected by such an approach. Subtransactions do not release their locks; they are inherited by their parent transaction.

For multi-layered centralized DBMS, some kind of multi-level transaction management was provided where the subtransactions serve as control structures in the various layers. To gain a higher degree of concurrency and more flexible control of lock granules, a so-called multi-level concurrency control was introduced. Furthermore, isolated rollback of subtransactions can be guaranteed. In System R [Astrahan76] this concept is used for two layers: locking is applied twice, on tuples until EOT (long tuple locks) and on pages for the duration of each tuple operation (short page locks for actions). Since tuple operations can be regarded as subtransactions and page locks are released before EOT of the parent transaction, this technique has been called "open nested transaction". The problems involved were discussed in [Gray81].

The generalization of open nested transactions for centralized systems - *called multi-level transactions* - was proposed in [Weikum86, Weikum84, Moss86, Been89] to allow for early release of locks at lower levels of control; however, they rely on compensation operations for subtransactions to be applied in the case of rollback recovery. A detailed description of all aspects of multi-level transaction management including a discussion of performance issues is given in [Weikum91]. Here, we don't want to consider this kind of multi-level structure and concurrency control for centralized DBMS operations.

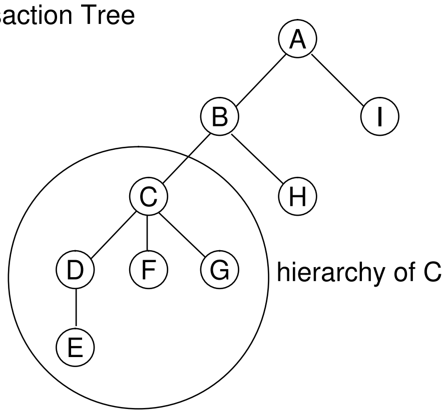
Due to the salient properties supported by nested transactions, many researchers have focussed attention on the design and implementation of them in distributed systems. Our approach is based on the proposals, results and experiences of distributed systems' design, especially as reported in [Allchin83, Jessop82, Liskov85, Müllerl83, Spector83, Walter84]; it tries to adjust the concept of nested transactions and improves its use for distributed DBMS. Our prime goal is its investigation and its conceivable extension for flexible intra-transaction parallelism. Due to space limitations we restrict our discussion to concurrency control and deadlock detection issues. Recovery problems are dealt with in [Moss87, Härder87, Rothermel89]. To facilitate our discussion, we introduce a model for nested transactions; it is designed so as to not prohibit parent/child- nor sibling-parallelism. In Sec. 3, the basic concurrency control model invented by Moss [Moss85] is discussed. In some systems, it has been extended and refined by the concept of downward inheritance enabling transactions to pass on locks to their child transactions. In Sec. 4, we propose a number of generalizations and extensions for concurrency control in nested transactions. The concept of controlled downward inheritance enables a parent to give its child access to shared data and at the same time to restrict its mode of usage. Another refinement allows the use of more general lock modes as compared to the simple S-X lock model. Hence, applications may better adjust their synchronization needs. So far, all efforts are directed towards enhancement of concurrency control in transaction hierarchies operating on 'flat' objects. Since every practical DBMS is forced to use an object hierarchy to provide fine as well as coarse lock granules at reasonable cost, we design a concurrency control protocol which combines object and transaction hierarchies as well as supports controlled downward inheritance. We conclude and summarize our results in the final section.

2. A Model of Nested Transactions

The concurrency control techniques we are going to present in this paper are based on the nested transaction model introduced by Moss [Moss85]. A transaction may contain any number of **subtransactions**, which again may be composed of any number of subtransactions - conceivably resulting in an arbitrarily deep hierarchy of nested transactions. The root transaction which is not enclosed in any transaction is called the **top-level transaction (TL-transaction)**. Transactions having subtransactions are called **parents**, and their subtransactions are their **children**. We will also speak of **ancestors** and **descendants**. The ancestor (descendant) relation is the reflexive transitive closure of the parent (child) relation. We will use the term **superior (inferior)** for the non-reflexive version of the ancestor (descendant). The set of descendants of a transaction together with their parent/child relationships is called the transaction's **hierarchy**. In the following, we will use the term 'transaction' to denote both TL-transactions and subtransactions.

The hierarchy of a TL-transaction can be represented by a so-called **transaction tree**. The nodes of the tree represent transactions, and the edges illustrate the parent/child relationships between the related transactions. In the transaction tree shown in Fig. 1, the root is represented by TL-transaction A. The children of subtransaction C are D, F, and G, and the parent of C is B. The inferiors of C are D, E, F, and G, and the superiors are B and A. Of course, the descendants and ancestors sets of C additionally contain C itself. The hierarchy of C is depicted as the subtree spanned by C's descendants.

Figure 1: Example of a Transaction Tree



The properties defined for flat transactions are **atomicity**, **consistency**, **isolated execution**, **and durability** (ACID-properties) [Härder83]. In the nested transaction model, the ACID-properties are fulfilled for TL-transactions, while only a subset of them are defined for subtransactions. A subtransaction appears atomic to the other transactions and may commit and abort independently. Aborting a subtransaction does not affect the outcome of the transactions not belonging to the subtransaction's hierarchy, and hence subtransactions act as firewalls, shielding the outside world from internal failures. If the concurrency control scheme introduced by Moss is applied, isolated execution is guaranteed for subtransactions. However, to increase intra-transaction parallelism the enhanced schemes proposed in this paper allow transactions belonging to the same TL-transaction hierarchy to share data in a controlled manner. The durability of the effects of a committed subtransaction depends on the outcome of its superiors - even if it commits, aborting one of its superiors will undo its effects. A subtransaction's effects become permanent only when its TL-transaction commits. The consistency property for subtransactions seems to be too restrictive, as sometimes a parent transaction needs the results of several child transactions to perform some consistency preserving actions.

To exploit the inherent potential of nested transactions and their advantages as stated in Sec. 1, the degree of intra-transaction parallelism should be as high as possible. Two kinds of intra-transaction parallelism can be defined, **parent/child-parallelism** and **sibling-parallelism**. If the first kind of parallelism is supported, then a transaction may run in parallel to its children, while in the second kind siblings are allowed to run concurrently. Using both these definitions, we are able to characterize four levels of intra-transaction parallelism:

- **Neither parent/child- nor sibling-parallelism**: At any point in time there is at most one transaction active in a TL-transaction hierarchy, i.e. there is no intra-transaction parallelism at all. Since all transactions in a hierarchy are executed serially, no concurrency control among them is needed. For example, if each transaction is executed by a single process and processes only communicate by means of a (synchronous) remote procedure call mechanism, only this level of "parallelism" can be provided.
- **Only sibling-parallelism**: If only siblings may be performed concurrently, then a transaction never runs in parallel with its superiors. This kind of restricted parallelism enables a transaction to share objects with its ancestors without further concurrency control. For example, in the ARGUS system [Lisk85], the intra-transaction parallelism is restricted to this level.
- **Only parent/child-parallelism**: Since a transaction and its children may run concurrently but siblings may not, in a TL-transaction hierarchy only the transactions along one path of the hierarchy may run in parallel. 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. (This reason hardly justifies such a system design).
- **Parent/child- as well as sibling-parallelism**: This level permits arbitrary intra-transaction parallelism, i.e. in principle, all transactions of a TL-transaction hierarchy may be executed concurrently. Of course, compared to the degrees of parallelism described above, this degree requires the most sophisticated concurrency control scheme. For example, LOCUS [Müller83] and CLOUDS [Allchin83] support this level of parallelism.

As discussed so far, our transaction model does not contain any essential restrictions. Transactions may either be performed entirely on a single processor site or may be distributed over multiple processors located at one or more sites. Moreover, the model does not restrict the kind of data distribution implemented by the underlying system, and hence our considerations apply for data sharing as well as for data distribution approaches [Rahm92]. Since we focus on concurrency control concepts, introduction of further refinements or implementation issues would only burden our discussion.

3. Basic Locking Rules for Nested Transactions

Locking as the standard method of concurrency control in DBMS has been used successfully for a variety of applications over the past decade and longer. Therefore, it is reasonable to choose conventional locking protocols as our starting point of investigation for nested transactions. Conventional locking protocols offer two modes of synchronization - read, which permits multiple transactions to **S**hare an object at a time, and write, which gives the right to a single transaction for e**X**clusively accessing an object (e.g. see [Gray78]). As far as concurrency control is concerned, our data model initially consists of disjoint ('flat') objects O_i which are the **lockable units**.

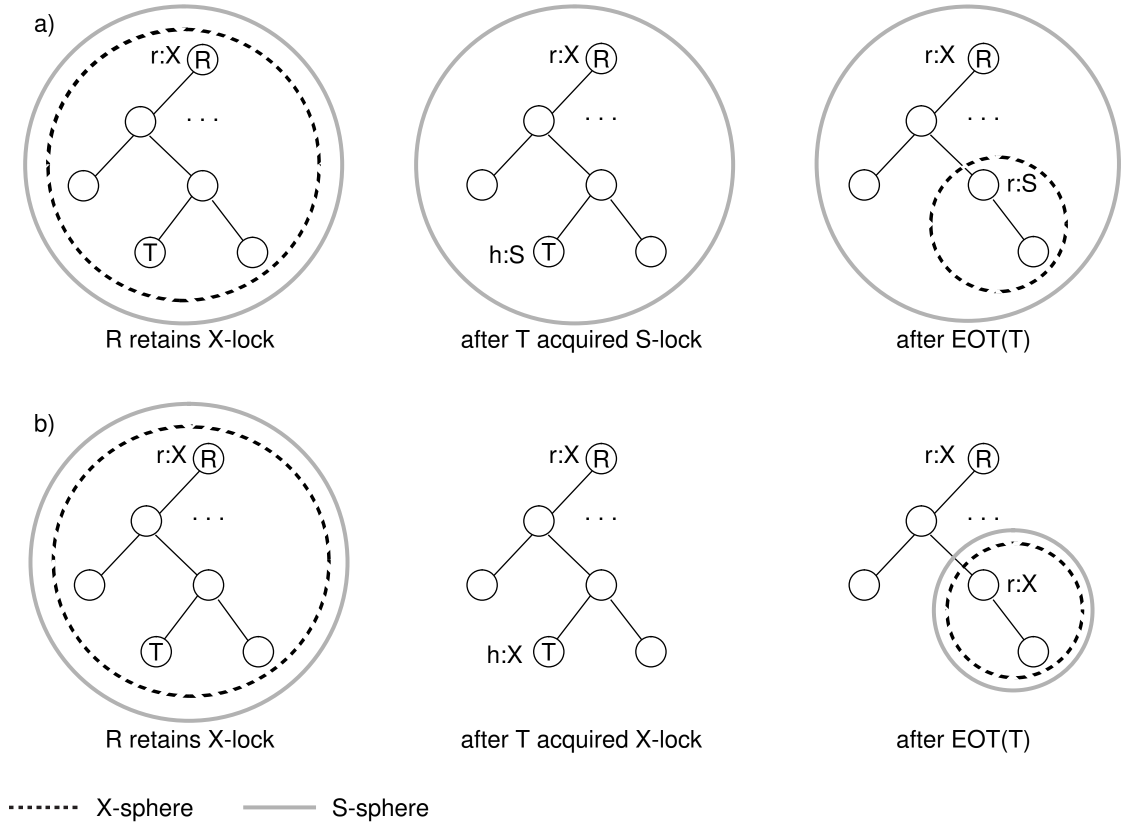
In the next part of this section, we will summarize the locking scheme for nested transactions proposed by Moss [Moss85]. This scheme only allows for upward inheritance of locks, i.e. a transaction can inherit locks from its children, but not vice versa. In the last part, we will extend this scheme such that it supports upward as well as downward inheritance. Both schemes presented in this section have been implemented in several systems.

3.1 Upward Inheritance of Locks

Before describing the locking rules proposed by Moss, we have to introduce some terminology. Possible lock modes of an object are NL-, S-, and X-mode. The **null** mode (NL) represents the absence of a lock request for or a lock on the object. A transaction can **acquire** a lock on object O in some mode M; then it **holds** the lock in mode M until its termination. Besides holding a lock a transaction can **retain** a lock. When a subtransaction commits, its parent transaction inherits its locks and then retains them. If a transaction holds a lock, it has the right to access the locked object (in the corresponding mode), which is not true for retained locks. A retained lock is only a place holder. A retained X-lock, denoted by r:X (as opposed to h:X for an X-lock held), indicates that transactions outside the hierarchy of the retainer cannot acquire the lock, but that descendants of the retainer potentially can. That is, if a transaction T retains an X-lock, then all non-descendants of T cannot hold the lock in either X- or in S-mode. If T is a retainer of an S-lock, it is guaranteed that a non-descendant of T cannot hold the lock in X-mode, but potentially can in S-mode. As soon as a transaction becomes a retainer of a lock, it remains a retainer for that lock until it terminates.

Having introduced this terminology, we can formulate the locking rules now:

Figure 2: Upward Inheritance of Locks



R1: Transaction T 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 T.

R2: Transaction T 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 T.

R3: When a subtransaction T commits, the parent of T inherits T's (held and retained) locks. After that, the parent retains the locks in the same mode (X or S) in which T held or retained the locks previously.[2]

R4: 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.

Obviously, the rules stated above only allow for **upward inheritance** of locks, i.e. a transaction can only inherit its children's locks, but not vice versa. The principle of upward inheritance is exemplified in Fig. 2, where we use the notions X- and S-sphere for describing the implications of this principle. The X-

2. Note, the inheritance mechanism may cause a transaction to (conceptually) retain several locks on the same object. Of course, the number of locks retained by a transaction should be limited to one by only retaining the most restrictive lock.

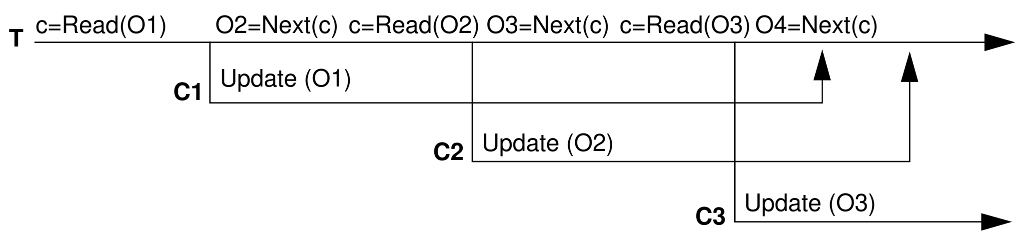
sphere (S-sphere) of an object is defined to be the set of transactions that can potentially lock this object in X-mode (S-mode). In Fig. 2a, the X-sphere of an object disappears entirely when transaction T acquires an S-lock on this object, i.e. no transaction may acquire an X-lock on this object, while each transaction in R's hierarchy may lock the object in S-mode. After commit of T, a new X-sphere is established, which consists of the descendants of T's parent transaction. In Fig. 2b, the X- as well as the S-sphere disappear when T acquires an X-lock. When T commits, a new X- and S-sphere are established. In general, a transaction acquiring a lock on an object may cause the object's X- or S-sphere to shrink, while the termination of a transaction may cause them to grow.

The rules stated above only allow for upward inheritance at commit time, i.e. a transaction may not inherit a child's locks before the latter commits. This restriction guarantees that transactions can see the effects of committed children only, and hence are not affected by failures of children. Furthermore, this restriction ensures that the subtransactions of a transaction tree are serializable. Allowing upward inheritance before commit time would cause transactions to become dependent on the outcome of child transactions, i.e. subtransactions would not act as firewalls anymore [Härder87] such that application code within a subtransaction had to cope with concurrency and recovery issues.

3.2 Downward Inheritance of Locks

We feel that especially the restrictions caused by only allowing upward lock inheritance prevent desirable decompositions of transactions into a set of cooperating subtransactions. For example, assume an application that navigates through an object base and updates each of the accessed objects. A desirable decomposition of the above task is depicted in Fig. 3. Transaction T reads an object and determines the next object to be accessed by applying the *Next* operation on the current object's content. Then T creates a new child transaction, which asynchronously performs an *Update* operation on the current object, while T reads in the next object, on which it acts as described above. This decomposition has some appealing characteristics: (1) Update operations are performed in parallel. (2) If an update operation fails, it does not affect the other operations. A failed operation can be restarted at a later point in time. (3) The update operations are performed in isolation from each other. This is of particular importance if the update of an object may imply updates on other objects. For example, the update of two different objects may imply two updates of the same access path.

Figure 3: Decomposition of an Application



Unfortunately, this decomposition is impossible if the basic locking rules proposed by Moss are applied. To be able to perform the *Next* operation on an object, T must hold an R-lock on this object. Since T must hold this R-lock until it commits, no child of T can ever acquire an X-lock on this object. In other words, once an object has been read by T, it cannot be updated by T's children anymore.

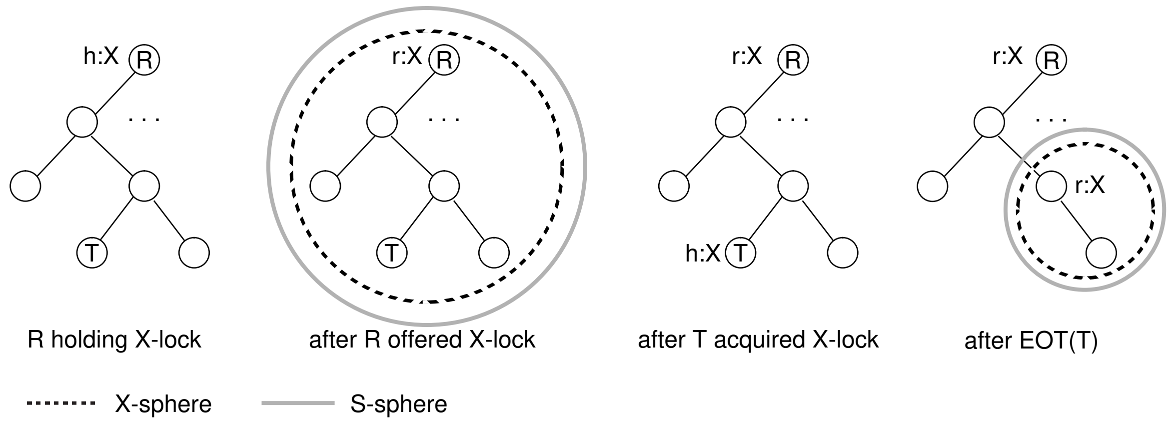
The decomposition required in the example of Fig. 3 is possible as soon as **downward inheritance** of locks is supported by the underlying locking scheme. In such a scheme, subtransactions may inherit locks from superiors, where inheritance of a lock can only take place after the superior holding this lock has explicitly **offered** this lock for downward inheritance. A transaction can offer a lock it holds to the transactions in its hierarchy, which can then acquire the lock according to the locking rules stated above. Consequently, the concept of downward inheritance allows a transaction to make all or a subset of its locks available to its hierarchy. The locking rules proposed by Moss can easily be extended to support downward inheritance by adding a new rule:

R5: A transaction T holding a lock can **offer** the lock (to the transactions in its hierarchy). After offering the lock, T retains the lock in the same mode it held the lock before.

A transaction offering a lock (temporarily) disclaims the right to access the locked object and gives the transactions in its hierarchy the opportunity to lock this object in any mode. Of course, in its hierarchy there might be either at most one transaction holding the lock in X-mode or a number of transactions holding the lock in S-mode. Since the transaction offering a lock still retains the lock in the mode it held the lock before, no transaction from outside its hierarchy can lock the object in a mode that conflicts with the mode of the retained lock. To become a holder again, the transaction must acquire the lock anew, which only succeeds if rules R1 and R2 stated above are fulfilled.

An example for applying the lock offering mechanism is illustrated in Fig. 4. When transaction R offers the X-lock it holds, an S- and X-sphere comprising R's hierarchy is established for the corresponding object, i.e. all descendants of R have the opportunity to lock the object either in S- or in X-mode. As depicted in the example, the object's X- and S-sphere disappear when a descendant of R locks the object in X-mode.

Figure 4: Downward Inheritance of Locks



If downward inheritance of locks is possible, the isolation property of transactions may be violated. While transactions belonging to different TL-transaction hierarchies still cannot interfere, transactions of the same hierarchy may share data. As a consequence, a transaction may see uncommitted data of superiors. This, however, cannot lead to inconsistencies since the effects of the transaction are undone when a superior aborts. On the other hand, a transaction may never see uncommitted data of inferiors, i.e. subtransactions act as firewalls even if downward inheritance of locks is allowed.

A lock offering mechanism similar to the one described above has been implemented in the LOCUS system [Müller83]. Some kind of *automatic downward* inheritance is provided in the ARGUS system [Liskov85]. In this particular approach, concurrency control is considerably simplified, since conflicts among transactions in a hierarchical path are prevented by only allowing sibling-parallelism. Automatic downward inheritance is then implicitly obtained by the rule that a transaction may acquire a lock if each transaction holding this lock is a superior of it.

4. Enhanced Concurrency Control for Nested Transactions

By using the idea of downward inheritance we gain more flexibility of lock inheritance in a given transaction hierarchy, however, with poor control over its specific usage. For this reason, the kind of offering concept has still some shortcomings in situations, where a transaction offering a lock desires to control the mode in which its inferiors can hold the lock. For example, consider the access sequence shown in Fig. 3 once more. With the additional rule R5 it is possible to make the desired decomposition. When transaction T offers the X-lock for O1, O2, O3, etc., then its children C1, C2, C3, etc., can acquire and hold the lock in any mode later on. However, it would be helpful if T could prevent some child Ci from being able to hold the lock in X-mode in order to make sure that Ci cannot change the resp. Oi.

4.1 Controlled Downward Inheritance

The need for controlling the lock mode in which inferiors can access an offered object becomes more obvious if we consider an example from a cooperative design environment [Bancilhon85, Kim84]. Fig. 5 shows a design task which is structured as a three-level transaction hierarchy. Assume, transaction B generates an object O, describing the interface of a work piece. Transactions C and D, which are children of B, design subparts of the work piece and therefore require read access to the interface description. To allow its children to read O, B must offer the lock which it holds on O. If there is no way to control the mode in which children can hold the lock, one of the children may acquire the lock in X-mode which has two undesirable consequences: First, the child can change O, and second, the child blocks its siblings by preventing them from reading O.

To overcome these problems we suggest an extension of the locking rules introduced in the previous section. In the scheme discussed previously, a transaction can offer locks that it holds to its inferiors. In the extended scheme, we will replace the lock offering mechanism by primitives supporting the upgrading and downgrading of locks:

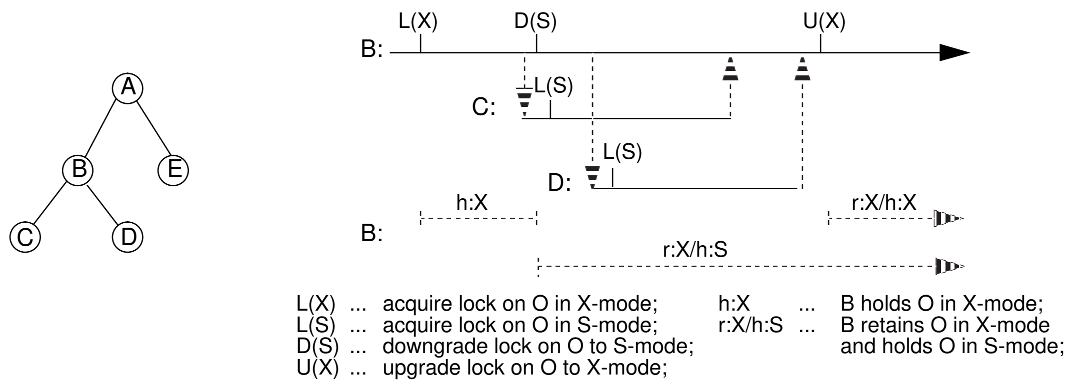
Downgrade

A transaction T holding a lock in mode M can downgrade the lock to a less restrictive mode M'. After downgrading the lock, transaction T holds the lock in mode M'*and* retains the lock in mode M. For example, a transaction holding a lock in X-mode can downgrade the lock to mode S or NL[3].

Upgrade

A transaction T holding a lock in mode M can upgrade the lock to a more restrictive mode M' if the following condition is satisfied: No other transaction holds the lock in a mode conflicting with M', and all transactions that retain the lock in a mode conflicting with M' are ancestors of T[4]. For example, a transaction T holding a lock in S-mode can upgrade the lock to mode X if no other transaction holds the lock in X- or S-mode, and all transactions retaining the lock in X- or S-mode are ancestors of T.

Figure 5: Decomposition of a Design Task



a) task structure

b) concurrent work steps

In the *extended scheme*, holding and retaining a lock have exactly the same semantics as in Moss's scheme. After downgrading a lock from mode M to mode M', a transaction holds the lock in mode M' and retains it in mode M. Since the transaction retains the lock in M-mode, it prevents transactions outside its hierarchy from holding the lock in a mode conflicting with M. On the other hand, since it holds the lock in mode M' it keeps its inferiors from holding the lock in a mode conflicting with M'. That is, in contrast to the offering of locks described in the basic scheme, downgrade allows a transaction to control how its

3. Note that downgrading to NL does not correspond to a general release of the lock. Release of such locks is limited to the shere of the downgrader.
4. Note, this condition is equivalent to the condition that must be satisfied for a transaction acquiring a lock.

inferiors can hold a lock. For example, if a transaction T downgrades a lock from X- to S-mode, it prevents transactions outside its hierarchy from holding the lock in any mode, and precludes its inferiors from holding the lock in X-mode, but allows its inferiors to hold the lock in S-mode. Of course, downgrading a lock to NL-mode is equivalent to offering a lock in the basic scheme.

As stated above, the holder of a lock can upgrade the lock to a mode which is more restrictive than its current hold mode. This feature allows a transaction to upgrade a lock which it downgraded previously, e.g. a transaction that downgraded a lock to S-mode could again upgrade the lock to mode X as soon as its children have committed. Of course, transactions can also upgrade a lock without having downgraded it before. In the following, we will describe the extended locking rules. Italics will be used to point out the extensions added to Moss's scheme:

ER1: Transaction T may acquire a lock in X-mode *or upgrade a lock it holds to mode X* 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 T.

ER2: Transaction T 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 T.

ER3: When a subtransaction T commits, the parent of T inherits T's (held and retained) locks. After that, the parent retains the locks in the same mode (X or S) as T held or retained them before.

ER4: When a top-level transaction commits, it releases all locks it holds or retains.

ER5: When a transaction aborts, it releases all locks it holds or retains. If any of its superiors hold or retain any of these locks, they continue to do so.

ER6: *A transaction T holding a lock in X-mode can downgrade the lock to mode S or NL. After performing the downgrade operation, T retains the lock in X-mode.*

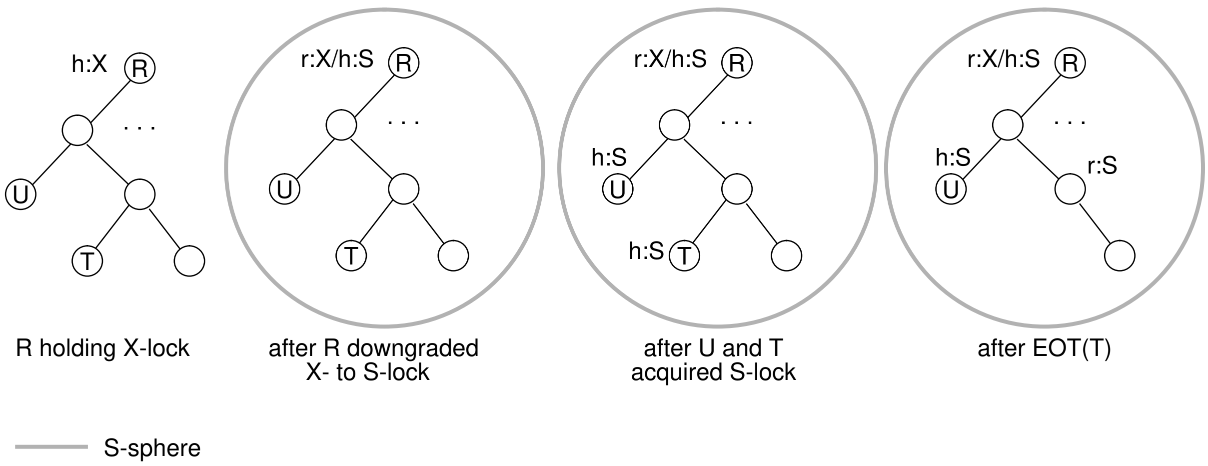
ER7: *A transaction holding a lock in S-mode can downgrade the lock to mode NL. After performing the downgrade operation, T retains the lock in S-mode.*

The mode to which a transaction T downgrades a lock determines the modes in which the transactions of T's hierarchy cannot hold the lock. If the downgraded mode is S, the transactions of T's hierarchy cannot hold the lock in X-mode (since S conflicts with X). If the downgraded mode is NL, then the transactions in T's hierarchy can potentially hold the lock in any mode.

Some examples may help to clarify the key issue of controlled downward inheritance. The effect of offering an X-lock can be depicted in the scenario in Fig. 4. A similar scenario in Fig. 6 illustrates the downgrading of an X-lock to mode S. (Downgrading of S-locks are handled in an analogous manner). The essential issue observed in this example is that only S-locks may be granted within R's hierarchy, i.e. no X-sphere is established when the lock is downgraded to mode S.

Given these extended locking rules, the problem described in the design environment example above can be solved very easily (see Fig. 5). After having generated object O, transaction B downgrades the X-lock it holds on O to mode S. Since it then holds the lock in S-mode, C and D are prevented from holding the lock in X-mode which guarantees that they cannot change O nor block each other. Note that since B retains the lock in X-mode after downgrading the lock, transactions A and E cannot hold the lock in any mode, i.e. A and E can neither read nor write O. After commit of C and D, B can upgrade the lock once again. .

Figure 6: Controlled Downward Inheritance



4.2 Correctness Concerns

As stated in Section 3.1, upward lock inheritance at commit ensures that subtransactions act as firewalls in case of a failure and that all subtransactions of a TL-transaction remain isolated. Since we have proposed the concept of controlled downward inheritance, we like to discuss the impact of this concept on the correctness of concurrent executions.

TL-transactions are serializable because

- each transaction in a TL-transaction tree locks each data object before accessing it
- all locks hold by transactions in a transaction tree are released not before the TL-transaction commits.

This locking protocol corresponds to strict 2-phase locking for TL-transactions. It determines their serialization order by the time of their commit, as it holds for single-level transactions.

Now let us discuss the visibility of data changes and their induced dependencies **within** a transaction tree. In Moss's nested transaction model the following holds:

- A transaction may see changes only of those transactions that are committed and it depends on[5]. We say a transaction T depends on a transaction T', if undoing the effects of T' causes the abortion of T.
- Once a transaction T has seen a state of an object, this state will never be seen or changed by another transaction before T commits.

In contrast to that, our model allows for controlled downward inheritance which makes uncommitted data available to inferiors. For this reason, we observe the following properties.

- A transaction T may see changes only of
  - those transactions that are committed and T depends upon and
  - those transactions that are superiors of T
- An object state seen by a transaction T may be changed by inferiors of T.

A transaction may see changes of superiors only if these transactions have downgraded the corresponding locks explicitly. That is, whether or not a transaction may see the effects of superiors can be controlled by the application logic. In terms of failures a transaction seeing changes of superiors causes no problems because, if a superior aborts, this transaction is aborted also. Note that once a transaction has seen an object, the object cannot be changed by a superior (once more) before this transaction commits. If a transaction downgrades a lock, it must be aware of the consequences of the reduced isolation. Downgrading from S- to NL-mode may cause unrepeatable reads from the downgrading transaction's point of view. With X-locks two cases must be considered: Downgrading from X- to NL-mode and from X- to S-mode. In the first case, from the downgrader's point of view lost updates und unrepeatable reads are possible in principle. However, a much more flexible cooperation is enabled where the correctness of execution has to be enforced by application level protocols. In CSCW-like applications, it is even conceivable that this kind of high-level control is based on so-called social protocols between end users. In the latter case, which prevents the inferiors of the downgrading transaction from keeping the downgraded lock in X-mode, neither unrepeatable reads nor lost updates can occur.

An important question is whether the firewall property of nested transactions is in some way affected by the downgrading mechanism. A transaction downgrading a lock does not become dependent on the outcome of its inferiors. When a child fails, its updates possibly on objects with downgraded locks are rolled back. Therefore, the downgrading transaction is not affected and may create another child to do the

5. Remember the effects of a committed (sub)transaction only become permanent when its top-level transaction commits. The reason why a transaction may only see data of committed transactions it depends upon is to ensure that it is aborted when the effects of one of these transactions is wiped out due to a failure. In the transaction hierarchy depicted in Fig. 5a, transaction E may see changes from transaction C not before B has committed. When B commits, E becomes dependent on C as a failure wiping out effects of C causes A and of course E to be aborted.

work.
In summary, the fact that a transaction may see changes from superiors causes no problems nor is the firewall property affected by the downgrading mechanism. Lost updates may only happen when locks are downgraded from X- to NL-mode. In this case, which provides the highest degree of flexibility in terms of cooperation, application-level concurrency control mechanisms are needed to ensure the required form of correctness. Since the application itself can decide how and when to use the downgrading mechanism, it can adapt the level of system supported isolation to its cooperation needs and its facilities for application-specific concurrency control.

4.3 Generalization of Lock Modes

Thus far, we have described and refined a concurrency control scheme for S-X locks on 'flat', non-overlapping objects (e.g. tuples or relations); in particular, we have developed a mechanism for controlled downward inheritance of locks in nested transactions. Closer consideration reveals that the lock modes (comprising only S and X so far) may be enriched by special modes to better adapt concurrency control to access patterns in practical applications. For example, tailored lock modes for frequent kinds of object access could be helpful to more effectively exploit the inherent parallelism of concurrent transactions. Furthermore, the use of semantic knowledge could greatly optimize some contention patterns of data access. However, this requires enhanced lock modes; in particular, it presupposes the ability to introduce user-defined lock modes, e.g. see [Allchin83, Schwarz84].

Such a refinement of lock modes may be easily integrated into our model presented so far. Assume that the data model remains unchanged. Then, our locking rules stated in Sec. 4.1 for S-X schemes can be generalized for basic and/or user-defined lock modes as follows:

GR1: Transaction T may acquire a lock in mode M or upgrade a lock it holds to mode M if

- no other transaction holds the lock in a mode that conflicts with M, and
- all transactions that retain the lock in a mode conflicting with M are ancestors of T.

GR2: When a subtransaction T commits, the parent of T inherits T's (held and retained) locks. After that, the parent retains the locks in the same mode as T held or retained them before.

GR3: When a top-level transaction commits, it releases all locks it holds or retains.

GR4: When a transaction aborts, it releases all locks it holds or retains. If any of its superiors hold or retain any of these locks, they continue to do so.

GR5: A transaction T holding a lock in mode M can downgrade the lock to a (less restrictive) mode M'. After downgrading the lock, T retains it in mode M.

The locking rules stated above allow upward as well as controlled downward inheritance for arbitrary lock modes. If rule GR5 were omitted, we would get a generalization of Moss's scheme which only provides for upward inheritance.

4.4 Use of Hierarchical Locks in Nested Transactions

Let us now reconsider our underlying data model which has some serious drawbacks for realistic concurrency control situations. In particular, the flat object structure which requires disjoint *lockable units* of a given granule makes it impractical for large databases when small granules are needed for some transactions and larger ones for others. To improve selective access to data granules of varying sizes, hierarchical locking schemes have been proposed. In our context, hierarchically structured objects introduce a certain complexity, as we then have to deal with orthogonal transaction and data hierarchies.

As mentioned earlier, locking of disjoint partitions of a given size is insufficient for performance reasons in most applications. The choice of lockable units affects locking overhead of a transaction (space for lock control blocks, time to request and release locks) as well as concurrency among transactions. Hence, it implies a dichotomy of *increased concurrency using fine lockable units and higher cost for lock management*. While small granules are appropriate for 'simple' transactions accessing a few tuples, they are intolerable (and hard to implement) for 'complex' transactions accessing a large fraction of the database. Assume, for example, a sequential scan of a relation with 10^6 tuples; having only tuples as lockable units would require 10^6 consecutive lock requests and storing of just as many lock control blocks (of course, in main memory for performance reasons). Hence, coarser granularity locks are sometimes more natural and efficient, e.g. when sorting or reorganizing a relation.

These arguments should convince every DBMS designer that an object hierarchy for locking purposes has to be provided. In fact, every 'practical' DBMS supports such a hierarchy of typically 2, 3, or 4 levels; e.g. System R has a generic 4-level hierarchy: database - segment - relation - tuple [Astrahan76].

An appropriate hierarchical locking scheme was proposed for flat transactions in [Gray76]. Two key ideas allowed for the design of a scheme that could be adapted to a transaction's needs for either locking a few items using a fine lockable unit or locking larger sets of items with larger lock granules:

- A node R in a hierarchy can be locked explicitly. As a result, its entire subtree is implicitly locked, too.
- A transaction locking part of the hierarchy places 'Intention mode' locks along the path to R to avoid a situation where an ancestor node of R is locked in an incompatible mode as compared to R. I-locks merely serve as place holders, signalling the fact that locking of a subtree is done at a lower level of the hierarchy, thereby preventing incompatible locks from being granted for the corresponding nodes.

Besides the known modes S and X, an *Intention Share* mode (IS) and an *Intention eXclusive* mode (IX) were introduced to express a transaction's intent to read and to update or read an object at a lower level of the hierarchy, respectively. A further refinement is the *Share and Intention eXclusive mode* (SIX) which grants an S-lock for the entire subtree to a transaction. In addition, it indicates the transaction's intention to request X-locks explicitly for 'finer' object granules later on. The following table taken from [Gray76] shows the compatibilities among request/lock modes which derive from these semantics:

Compatibility		**Mode of lock**					
		NL	**IS**	**IX**	**S**	**SIX**	**X**
Mode of request	**NL**	yes	yes	yes	yes	yes	yes
	IS	yes	yes	yes	yes	yes	no
	IX	yes	yes	yes	no	no	no
	S	yes	yes	no	yes	no	no
	SIX	yes	yes	no	no	no	no
	X	yes	no	no	no	no	no

For a comprehensive discussion of the precise effects of the lock modes and their compatibilities we refer the reader to the seminal work of J. Gray [Gray78].

Basic Locking Rules for Object Hierarchies

We have now introduced the essential ingredients of both generalized locking rules for nested transactions and appropriate lock modes for an object hierarchy. How can we combine both together? We start with the basic concurrency control model where only upward inheritance is allowed. For the transaction hierarchy, our generalized rules GR1 - GR4 apply. Furthermore, when acquiring a lock on an object O, we have to consider additional rules resulting from the object hierarchies.

As opposed to flat objects, according to Gray et al. an approach for controlling concurrent access to an object hierarchy has to obey the following rules:

1. Instead of locking an object directly, every transaction has to observe a strict *hierarchical protocol* requesting appropriate locks from root to leaf in the object hierarchy - in the following denoted as **root-to-leaf rule**. A lock is granted at each level according to the compatibilities expressed in the above table. As soon as a lock is obtained, a transaction may request another appropriate lock at the same or at the next lower level.

2. Level-to-level transitions should obey the following constraints called **level-to-level rules**:

   - IS held at a node only allows IS and S to be requested on descendant nodes.
   - IX granted for a node carries the privilege to request IS, IX, S, SIX and X at the next level.
   - S and X allow read and write access (respectively) to all descendants of the node without further locking.
   - SIX carries the privileges of S and IX; hence, while S mode allows read only access to all descendants, write access at lower levels may be requested by IX or X at the next level.

As far as acquiring locks is concerned, the rules obtained for the transaction hierarchy and the object hierarchy must be satisfied independently. Following the root-to-leaf rule, transactions must request their locks from root to leaf in the object hierarchy. Whether or not a lock for an object may be granted in a particular mode is decided according to the level-to-level rules, the generalized locking rules (GR1 - GR4), and the lock mode compatibilities depicted in the table above. Since the rules introduced for the object hierarchy are independent of the underlying transaction model and the rules for both hierarchies are applied independently, our protocol and that proposed in [Gray76] for flat transactions only differ in the rules implied by the transaction model.

An example may clarify the issues involved in lock retainment for object hierarchies. In the following scenario, T_1 passes on its hierarchical locks for X-access on relation R to its parent P at EOT(T_1). After having retained the locks, P and its inferiors T_i are qualified to acquire read or write access on R or to tuples of R. The following table shows the locks of T_2 and T_3 for obtaining write and read access on tuples of R.

Object hierarchy	**Before EOT(T_1)**	**After EOT(T_1)**		
		P	**T_2**	**T_3**
Database DB Segment S Relation R $Tuples_i$	h:IX h:IX h:X	r:IX r:IX r:X	h:IX h:IX h:IX h:X on t_1 h:X on t_2	h:IS h:IS h:IS h:S on t_3 h:S on t_4

Upgrading and Downgrading Hierarchical Locks

Although we have succeeded in tying together both hierarchy types (transaction and object hierarchies), we have so far obtained only a more economical and efficient solution of the concurrency problem than compared to the basic approach in Sec. 3.1. Since we cannot make a transaction's objects available to its inferiors, all arguments discussed earlier apply. Therefore, it is desirable to enable controlled downward inheritance in the presence of object hierarchies, too.

Assume, for example, a transaction P holds an SIX-lock on a relation R and wants to permit write access to tuples of R by its inferiors. Using the same kind of inheritance mechanism as in Sec. 4.1, P has to downgrade its lock on the object to an appropriate mode. To do so, P retains the SIX-lock on R (r:SIX) and holds R in IX-mode (h:IX). Note that r:SIX only prevents incompatible locks on R from being granted to non-descendants, but not to inferiors.

Let us examine whether such a straightforward approach may be applied. In the scenario depicted by the following table, P holds R in SIX-mode and some tuples of R in X-mode. We assume that P downgrades the SIX-lock on R. Requesting a lock by an inferior T implies that T obeys the root-to-leaf and the level-to-level rules. Hence, as soon as T has acquired appropriate locks for the ancestors of R, it can request a compatible lock for R. The presented scenario is meant to serve as a counter-example for arbitrary inheritance of hierarchical locks and aims at clarifying a new issue: *Inheritance of objects in data hierarchies.* It shows that P holds some locks at the tuple level while it has downgraded the corresponding lock at the relation level to NL-mode, i.e. without particular protection.

Object hierarchy	**P**	**P** after downgrade	**T** using downgraded lock
Database DB	h:IX	h:IX	h:IX
Segment S	h:IX	h:IX	h:IX
Relation R	h:SIX	r:SIX/-	h:SIX
Tuples$_i$	h:X on t_1	h:X on t_1	h:X on t_3
	h:X on t_2	h:X on t_2	h:X on t_4

In the illustrated situation, T acquires an SIX-lock on R giving read access to all tuples of R. On the other hand, P has still some tuples locked in X-mode, namely t_1 and t_2. These exclusively locked tuples would be read by T, since read access to tuples of R need not be checked by T anymore. Even worse, write-write interference on the same tuples could occur, if T had locked R in X-mode. Of course, the sketched examples may cause severe consistency problems. These anomalies would not occur if the lock on relation R together with all locks on its tuples had been downgraded. Control given by the hold-mode alone would not guarantee the desired consistency as exemplified by only downgrading R to S-mode.

The key observation in the example above is that downgrading a lock without considering the whole object hierarchy may lead to inconsistencies. The same can be shown for upgrading locks in object hierarchies. For example, if a transaction has locked a database in IS-mode and upgrades an S-lock it holds on a segment of this database to X-mode, similar inconsistencies may occur. Obviously, to prevent violations of the level-to-level rules, upgrading or downgrading of a lock may enforce upgrade or downgrade operations on other locks held in the object hierarchy.

When a transaction T upgrades a lock held on an object O within an object hierarchy, it might be necessary to also upgrade locks of T held on superior objects of O in order to satisfy the level-to-level rules. For example, if T holds an IS-, IS-, and S-lock on a database, a segment of this database and a relation of this segment, respectively, the level-to-level rules enforce the upgrading of both IS-locks to IX-mode before the relation lock can be upgraded to X-mode. Since upgrading the locks on an object and superior objects are not performed in an atomic manner, upgrading should be done in a root-to-leaf direction. Of course, an upgrade operation can only take place if the generalized locking rules GR1 to GR4 are fulfilled. Otherwise, it is blocked which may cause deadlocks to occur (see Sec. 5).

Since upgrading a lock on an object O converts the mode of the locks to a more restrictive one, the level-to-level rules are not violated as far as locks on inferior objects of O are concerned. However, due to the upgrade operation locks held by the upgrading transaction on inferior objects of O may become useless. For example, when a lock on relation R is upgraded from SIX- to X-mode (lock escalation [Bernstein87]), all locks held by the upgrading transaction on individual tuples of R are not needed anymore. A clean approach to handling those useless locks is to release them as part of the upgrade operation. An actual implementation may optimize this cleanup process using pragmatic arguments (e.g. see System R [Astrahan76]).

Downgrading a lock held by a transaction T on an object O is confined to the subhierarchy having O as the root object. Superiors of O in the object hierarchy are not involved since downgrading cannot violate the level-to-level rules as far as superiors of O are concerned. However, with respect to the objects in its subhierarchy, downgrading the lock on O may cause a violation of the level-to-level rules: if T holds a lock on a subobject O' of O, then after downgrading, the mode of the lock held on O' may violate the level-to-level rules. For example, assume T holds an IX-lock on a relation R and an X-lock on a tuple t of R. If T downgrades its lock on R from IX to IS, then T's X-lock on t is not consistent anymore with the lock mode of its parent object. As a consequence, downgrading the lock on O may require downgrading locks held by T on objects in the subhierarchy of O, such that the level-to-level rules are satisfied. In the example above, this would require downgrading T's lock on tuple t to S- or NL-mode. The following table, derived from the level-to-level rules, lists for each possible mode to which the lock on O can be downgraded, the modes in which T can hold locks on the objects in the subhierarchy of O without violating the level-to-level rules. For example, if the lock is downgraded to IS-mode, T can hold subobjects of O either in NL-, IS- or S-mode. If subobjects are held in more restrictive modes, the locks on these objects must be downgraded to one of the listed modes. Note, since downgrading an entire subhierarchy cannot be done atomically, downgrading should be performed in a leaf-to-root direction.

Object O of transaction T downgraded to mode	**Consistent modes for locks of T on subobjects of O**
NL	NL
IS	NL, IS, S
IX	NL, IS, IX, SIX, S, X
SIX	NL, IX, X
S	NL

By observing these rules, consistency-preserving downward inheritance of locks may be easily achieved by P in the previous example by downgrading the tuples t_1 and t_2 before downgrading relation R. Control of lock usage is then possible by downgrading to the appropriate modes. In the following scenario, the locks in the subhierarchy of relation R have been downgraded to different modes which allows for selective control of access to R's subobjects.

Object hierarchy	**P**	**P** after downgrade	**T** using downgraded lock
Database DB	h:IX	h:IX	h:IX
Segment S	h:IX	h:IX	h:IX
Relation R	h:SIX	r:SIX/h:IX	h:SIX
Tuples_i	h:X on t_1	r:X/h:S on t_1	h:S on t_1
	h:X on t_2	r:X/- on t_2	h:X on t_2

Downgrade of an intention mode (IS, IX, SIX) implies subsequent downgrades of locks on subobjects in order to satisfy the level-to-level rules. This, however, can be avoided by restricting downgrade operations to S- and X-locks. If T holds a lock on object O in S- or X-mode, then the entire subhierarchy of O is locked implicitly for T, that is, T does not hold any locks on subobjects of O. Hence, downgrade of O does not involve downgrading locks on lower levels of O's subhierarchy.

Let us summarize our findings for controlled downward inheritance of locks in a data hierarchy. In general, downgrading of entire subtrees is necessary for hierarchical objects to guarantee consistency of downward inheritance in nested transactions. That is, if an M-lock held by a transaction T on an object O is downgraded, it might be necessary to downgrade locks held by T on inferiors of O in order to satisfy the level-to-level rules. If downgrading is allowed only on X- and S-locks, then the downgrading of a lock never involves locks held on lower levels of the object hierarchy, which simplifies the downgrade mechanism substantially.

5. Deadlock Detection in Nested Transactions

Lock protocols are pessimistic, that is, they are blocking lock requests of data currently granted to another transaction in a conflicting mode, and, therefore, not immune to deadlocks. Deadlocks may occur among transactions belonging to various TL-transactions and even among subtransactions within a single transaction hierarchy. For deadlock detection, we mainly follow the basic approach sketched in [Moss85], which allows to identify existing deadlocks. In addition, we propose the maintenance of further information (waits-for-retained-locks relation) to detect opening-up deadlocks as early as possible.

Deadlocks in nested transactions can be resolved by the concepts known for single-level transactions extended by some mechanisms tailored to the properties of the nested structure [Moss85, Rukov91]. When a transaction acquires a lock for some data which is incompatible with a lock held by another transaction, the requesting transaction is deactivated: a direct wait for the lock holder occurs. All direct waits are maintained in a waits-for-lock relation in order to detect deadlocks. Using this waits-for-lock relation, deadlock detection can be performed immediately when a transaction is blocked or after some elapsed time. A deadlock exists if and only if a cycle is found in the waits-for-lock relation. For single-level transactions, such a cycle is composed of direct waits (or waits for lock) only.

As we have seen in Sec. 3.1, nested transactions have an inner structure which determines along which paths locks are inherited and whether retained locks can be acquired. Assume a subtransaction R waits for a lock held by a subtransaction T. After commit of T, its locks are inherited and retained by its parent transaction P (see Fig. 7). Now, lock requests from transactions in P's (X-or S-) sphere can be served. Transaction R outside P's sphere, however, cannot acquire retained locks; for this reason, it has to wait for the retained locks of P. Such waits for retained locks are indirect waits. They propagate along the ancestor hierarchy of P. In the following, we will introduce two different waiting relationships.

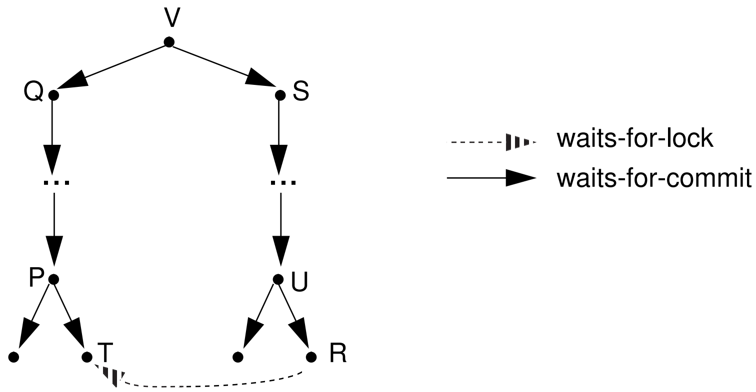
Waits-for- retained-locks: A lock requestor R directly waits for a lock holder T if the mode of the requested lock is in conflict with the lock mode hold by T. Let Q be the highest ancestor of T that is not an ancestor of R. Then, R indirectly waits for all ancestors of T up to Q until they commit. Those wait relationships are called waits-for-retained-lock.

This wait rule implies if requestor R is not in the same TL-transaction as T, then R must wait for the retained locks of T's TL-transaction which are released at its commit.

Waits-for-commit: Since a waiting lock requestor R cannot proceed with its work, all ancestors of R may have to wait. In Fig. 7, U cannot commit until R does and U's parent cannot commit until U does, that is, all ancestors of R cannot commit before R does.We denote this kind of wait relationship by waits-for-commit, which can be represented by the parent-child relationships, as outlined in Fig. 7.

Due to this dependency, an ancestor of R may wait for all transactions for which R directly or indirectly waits. Of course, waiting may be broken up as soon as T or one of its ancestors abort. As illustrated in Fig. 7, R directly waits for T and indirectly waits (for retained locks) for P, ..., Q. Furthermore, since S, ..., U wait for commit of R, they also wait for T, P, ... Q.

Fig. 7: Lock and commit waits

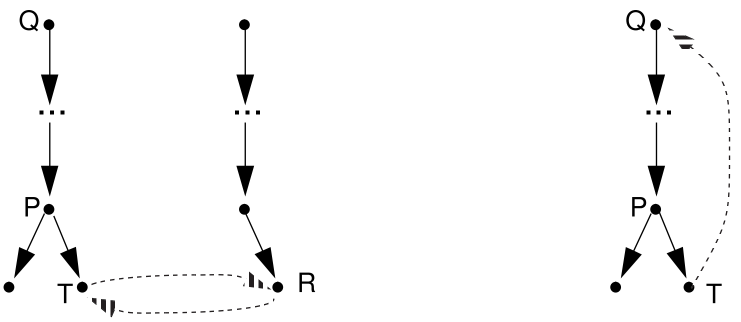


In Sec. 4.4, hierarchical locking was employed to nested transactions. The key observation exhibited that object and transaction hierarchies are orthogonal. As a consequence, further aspects are not added to deadlock detection when hierarchically composed objects are used. As illustrated by Fig. 7, waits-for relations occur among transactions; thus, the rules of the hierarchical locking protocol do not interferewith the waits-for-lock and waits-for-commit relationships as long as the root-to-leaf and the level-to-level rules are observed.

5.1 Detection of existing deadlocks

In order to handle deadlock detection in nested transactions successfully, we have to combine the various waits-for relations. Considering the waits-for-lock relation, only direct-wait deadlocks can be found, as indicated in Fig. 8a, whereas other kinds of deadlocks cannot be detected. This is true no matter whether a deadlock occurs within a TL-transaction or among subtransactions of various TL-transactions.

Fig. 8: Deadlock situations



a) direct-wait deadlock b) ancestor-descendant deadlock

The cycle in Fig. 8a consists of direct waits only, that is, all transactions in the cycle cannot proceed anymore. In Fig. 8b, however, another kind of cycle is encountered; this situation does not mean that progress has stopped everywhere in the cycle. T waits for a lock of Q; Q, ..., P may proceed for some time, but cannot commit without aborting T. Since T must be rolled back anyway, the best decision is to detect and resolve this ancestor-descendant deadlock immediately. A request of T causing a lock wait on its ancestor Q can be detected only by using the combined waits-for-lock and waits-for-commit relation information.

Situations such as illustrated by Fig. 8b would be frequently caused when descendants refer to exclusively used data in an uncoordinated way. Controlled downgrading of locks, however, provides a mechanism to avoid such cycles, that is, application knowledge is applied to reduce the possibility of a deadlock involving lock and commit waits. Coordinated work requires that a parent P should downgrade the lock on an object O currently granted to P, before it creates a child T to do some work on O . Then, T can acquire the lock for O in a non-conflicting mode without causing a blocking situation. Downgrading enables deadlock-free cooperation, but cannot enforce it; if T requests the lock in a mode more restrictive than the offered one, a deadlock may arise.

Upgrading a lock may lead to wait situations and therefore to deadlocks as they occur in single-level transactions. Assume in Fig.7 that T and R already hold an S-lock on object O. Now, if R upgrades the lock to mode X, R has to wait until a direct ancestor of R retains the lock or, if T and R are not in the same TL-transaction, until T's TL-transaction has committed. Hence, our wait rule applies to lock upgrades, too.

5.2 Detection of opening-up deadlocks

The combined use of waits-for-lock and waits-for-commit relations turned out to be sufficient for nested transactions to detect existing cycles embodying direct-wait or ancestor-descendant deadlocks. Since a waits-for-lock relationship is only represented between the requestor and the holder of a lock (or, after commit of the lock holder, the current retainer), waits-for-retained-lock relationships between the requestor and all ancestors of the holder (retainer) are not explicitly established in the waits-for information. For nested transactions, however, these waits-for-retained-lock relationships should be taken into account to provide for early deadlock detection. This may save a lot of useless work as shown in the scenario of Fig. 9.

Fig. 9: Opening-up deadlock among nested transactions

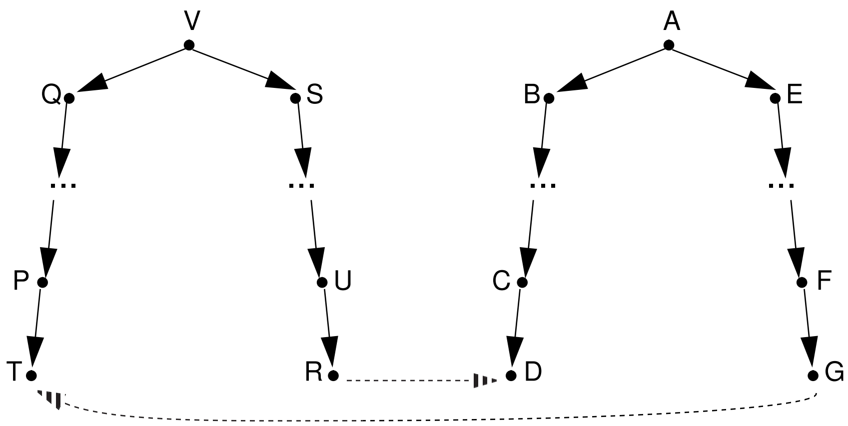


Fig. 9 represents a deadlock-free situation, since transactions T , D, and possibly others can proceed with their work. R waits for D and G for T to obtain the requested locks. All other waits indicated are waits-for-commit. R indirectly waits for the oldest ancestor of D that is not an ancestor of R which is A. On the other hand, G indirectly waits for V. If we evaluate this information ($R \rightarrow A$, $G \rightarrow V$), we are able to immediately detect a cycle to be opening up.

An optimistic attitude would not care about such an opening-up deadlock, since an abort of any transaction involved would eventually avoid the actual occurrence of the deadlock. For example, the abort of any transaction in Fig.9 resolves the opening-up deadlock before all progress ceases within the TL-transactions V and A. However, transaction aborts are regarded as exceptions and should not be taken into account as a remedy to break opening-up deadlock cycles.

In contrast, a pessimistic approach usually saves work. If we use the transitive waits-for-commit and waits-for-retained-lock relationships of all ancestors, e.g., of V on R and A on G as well as R on A and G on V, we can construct a direct (future) cycle between V and A and can roll back either V or A. However, deadlock detection and resolution at the level of the highest non-common ancestors of the transactions which have caused the conflict may not be appropriate. Deadlock resolution is typically based on transaction rollback and should affect minimal data granules or work lost.

For this reason, special measures should be used to determine an opening-up deadlock as early as possible and at the suitable level in the nested transaction hierarchies. In addition to waits-for-lock and waits-for-commit relations, the waits-for-retained-lock relationships have to be included in the waits-for information. For example in Fig.9, the relationships R → C, ..., R → B, R → A, as well as G → P,..., G → Q, G → V have to be represented in order to successfully search for opening-up cycles. Once an opening-up deadlock is detected, all transactions involved have to be considered to determine a low-cost victim for rollback. Since rollback of a parent transaction implies rollback of all its inferiors (committed and uncommitted), rollback of a child transaction is always cheaper than that of the corresponding parent transaction. For this reason, rollback of a lock holder (retainer) or a lock requestor is always cheaper than their ancestors in a potential cycle; hence, the set of transactions to choose a rollback victim from is the set of lock holders (retainers) and lock requestors. In Fig. 9, this set of candidates is D,T and R,G, respectively.

Note in contrast to [Moss85], in our transaction model these transactions must not be leaves in the current transaction tree. However, the same methods and cost measures could be applied for breaking up the cycle, in principle. Since candidate transactions can occur arbitrarily in the transaction hierarchy, resource estimation involving the evaluation of subtrees may become much more complicated.

To summarize, waits-for-retained-lock relationships are evaluated only to detect opening-up deadlocks as early as possible. Since the candidate transactions for breaking up the cycle are lock holders (retainer) and lock requestors, the mechanisms for deadlock resolution can be derived from those provided for single-level transactions.

Early detection of opening-up deadlocks saves transaction work. However as discussed above, the additional representation and management of waits-for-retained-lock relationships require some overhead. If deadlocks are infrequent, a particular system implementation has to take this trade-off into account.

6. Comparison of some System Implementations

In the following, we will compare some systems implementing nested transactions with regard to the degree of parallelism supported, the applied concurrency control schemes and the way deadlocks are treated. In particular, we will consider Argus [Liskov88, Liskov87], Camelot [Spector88, EpingerS91], Clouds [Ahamad87, Dasgupta89], Eden [Almes85, Pu85] and LOCUS [Müller83, Weinstein85]. The table below summarizes the results.

	Parent/Child Parallelism	Sibling Parallelism	Downward Inheritance	Controlled Downward Inheritance	Object Hierarchy Support	Deadlock Avoidance and Detection
Argus	no	yes	no	no	no	timout based resolution
Camelot	yes	yes	no	no	no	timeout based resolution
Clouds	yes	yes	yes	no	no	timeout based resolution
Eden	yes	yes	no	no	no	wound-wait avoidance scheme
LOCUS	yes	yes	yes	no	no	neither resolution nor avoidance

While Camelot, Clouds, Eden and LOCUS allow for parent/child as well as sibling parallelism, ARGUS does not permit a parent transaction to run in parallel with its children, resulting in simpler locking rules [Liskov87]. All the systems considered are based on two-phase locking, however only Clouds and LOCUS support downward inheritance. While the downward inheritance scheme in LOCUS requires a lock holder to explicitly state when downward inheritance may potentially take place, transaction in CLOUDS are allowed to share the locks of their ancestors in a totally uncontrolled manner [Allchin83] When a transaction closes a file in LOCUS, the lock held on this file becomes a retained lock. Although it supports downward inheritance by means of an explicit offering mechnism, the LOCUS scheme is uncontrolled in the sense that a lock holder offering a lock cannot control in which mode its descendants may acquire this lock.. None of the five systems supports controlled downward inheritance, nor do they support object hierarchies. Argus, Camelot and Clouds implement deadlock resolution based on a timeout mechanism, whereas Eden applies a wound-wait deadlock avoidance scheme [Rosenkrantz78]. LOCUS neither performs deadlock detection, nor implements an avoidance scheme. However, it provides an interface to operating system data permitting a system process to detect deadlock by constructing a wait-for-graph. In this manner, different deadlock resolution strategies may be implemented [Weinstein85].

7. Conclusions

We have presented an investigation of concurrency control in nested transactions. The focus of our paper has primarily been on achieving a high degree of intra-transaction parallelism within nested transactions by using locking protocols.

Our initial concurrency control mechanism for nested transactions was based on S-X locking protocols on flat objects which seriously limited parent/child parallelism. Therefore, the concept of downward inheritance was introduced and refined to *controlled downward inheritance* in order to enable a transaction to restrict the access mode of its inferiors for an object. Controlled downward inheritance turned out to be a useful concept for achieving safe parent/child cooperation on data structures to be read or written in a shared manner.

Practical applications sometimes have a need for specialized lock modes as well as multi-level object hierarchies offering efficient ways to lock granules of varying sizes. Therefore, we have generalized the locking rules for nested transactions to be applied for richer access modes on flat objects. Most importantly, this kind of generalization was a prerequisite for the integration of transaction and object hierarchies, since the appropriate use of object hierarchies implied suitable access modes beyond S- and X-locks. As a result, we could combine both types of hierarchies in a general concurrency control model and then could enhance the model again using the concept of controlled downward inheritance, for the even richer set of access modes.

Finally, we studied the principles of deadlock detection in nested transactions. In contrast to single-level transactions where the waits-for-lock relation is sufficient to search for waiting cycles among transactions, detection of all deadlocks in nested transactions further requires the maintenance of the waits-for-commit relation and its combined use with the waits-for-lock relation. If deadlocks are frequently anticipated, opening-up deadlocks, which may span transaction trees, should be detected as early as possible to save transaction work. For this purpose, we have additionally introduced the waits-for-retained-lock relation.

Acknowledgements

C. Mohan shared his great knowledge and experience on concurrency control with us. We would like to thank him for his contributions which led to essential simplifications and clarifications of the concepts proposed in the paper. We would also like to thank J. Palmer and P. Schwarz and the referees for their helpful comments on this paper.

References

[Almes85] Almes, G.T., Black, A.P., Lazowska, E.D., Noe, J.D. The Eden system: A technical review. *IEEE Trans. Softw. Eng.* SE-11, pp. 43-58, January 1985

[Ahamad87] Ahamad, M., Dasgupta, P., Le Blanc, R.J., Wilkes, C.T. : Fault tolerant computing in object based distributed systems. *IEEE 6th Symposium on Reliability in Distributed Software and Database Systems*, pp. 115-125, March 1987.

[Allchin83] Allchin, J. E.: An Architecture for Decentralized Systems. Technical Report GIT-ICS-83-23, School of Information and Computer Science, Georgia Institute of Technology, 1983.

[Anon85] Anon et al.: A Measure of Transaction Processing Power. *Datamation*, April issue, 1985.

[Astrahan76] Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D, Eswaran, K.P., Gray, J.N., Griffith, P.P., King, W.F., Lorie, R.A., McJones, P.R., Mehl, J.W., Putzolu, G.R., Traiger, I.L., Wade, B., Watson, V.: System R: Relational Approach to Database Management. *ACM TODS* 1, 2, 1976, 97-137.

[Been89] Beeri, C., Bernstein, P.A., Goodman, N.: A Model for Concurrency in Nested Transaction Systems. *Journal of the ACM,* Vol. 36, No. 1, 1989.

[Bernstein81] Bernstein, P. A., Goodman, N.: Concurrency Control in Distributed Database Systems. *ACM Computing Surveys* 13, 2, 1981, 185-221.

[Bernstein87] Bernstein, P. A., Hadzilacos, N., Goodman, N.: *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bancilhon85] Bancilhon, F., Kim, W., Korth, H.F.: A Model of CAD Transactions. *Proc. 11th Int. Conf. on VLDB*, Stockholm, Aug. 1985, 25-33.

[Dasgupta89] Dasgupta, P., Le Blanc, R., Appelbe, W.: The Clouds distributed operating system. Functional description, implementation details and related work. *IEEE 8th International Conference on Distributed Computing Systems*, San Jose, 1989.

[Eswaran76] Eswaran, K. P., Gray, J. N., Lorie, R. A., Traiger, I. L.: The Notions of Consistency and Predicate Locks in a Database System. *Commun. of ACM 19*, 11, 1976, 624-633.

[EppingerS91] Eppinger, J.L., Mummert, L.B., Spector, A.Z. (eds.): *Camelot and Avalon - A Distributed Transaction Facility.* Morgan Kaufmann, 1991.

[EppingerS91] Eppinger, J.L., Mummert, L.B., Spector, A.Z.: *Camelot And Avalon, A Distributed Transaction Facility*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1991.

[Fekete87] Fekete, A., Lynch, N., Merrit, M., Weihl, W.: Commutativity-Based Locking for Nested Transactions. Technical Report MIT/LCS/TM-370, MIT, Cambridge, 1988.

[Gawlick85] Gawlick, D.: Processing “Hot Spots” in High Performance Systems. *Proc. IEEE Spring CompCon*, San Francisco, Feb. 1985, 249-251.

[Gray78] Gray, J.N.: Notes on Database Operating Systems. Operating Systems - An Advanced Course, *Lecture Notes in Computer Science 60*, Bayer, R., Graham, R.M., Seegmueller, G. (eds.), Springer-Verlag, 1978, 393-481.

[Gray81] Gray, J.N.: The Transaction Concept: Virtues and Limitations. *Proc. 7th Int. Conf. on VLDB, Cannes*, Nov. 1981, 144-154.

[Gray76] Gray, J.N., Lorie, R.A., Putzolu, G.R., Traiger, I.: Granularity of Locks and Degrees of Consistency in a Shared Data Base. *Proc. IFIP Working Conf. on Modelling in Data Base Management Systems*, North-Holland, 1976, 365-394.

[Härder83] Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys 15*, 4, 1983, 287-318.

[Härder87] Härder, T., Rothermel, K.: Concepts for Transaction Recovery in Nested Transactions. *Proc. ACM SIGMOD*, San Francisco, May 87, 239-248.

[Jessop82] Jessop, W.H.: The EDEN Transaction Based File System. *Proc. 2nd Symp. on Reliability in Distributed Software and Databases*, 1982.

[Kim84] Kim, W., Lorie, R., McNabb, D., Plouffe, W.: Nested Transactions for Engineering Design Databases. *Proc. 10th Int. Conf. on VLDB,* Singapore, Aug. 1984, 355-362.

[Liskov87] Liskov, B., Curtis, D., Johnson, P., Scheifler, R.: Implementation of ARGUS, *Proc. 11th Symp. on Operating System Principles*, ACM Operating Systems Review, Vol. 21, No. 5, 1987.

[Liskov85] Liskov, B.: The ARGUS Language and System. Distributed Systems - Methods and Tools for Specification: An Advanced Course, *Lecture Notes in Computer Science 190*, Paul, M. and Siegert, H.J. (eds.), Springer-Verlag, 1985, 343-430.

[Liskov88] Liskov, B.: Distributed programming in ARGUS. *Comm. ACM 31*, pp. 300-312, March 1988.

[Mohan86] Mohan, C., Lindsay, B., Obermark, R.: Transaction Management in R* Distributed Data Base Management System. IBM Research Report RJ5037, San Jose, California, 1986.

[Moss85] Moss J.E.B.: *Nested Transactions: An Approach to Reliable Distributed Computing.* M.I.T. Press, Cambridge, 1985.

[Moss86] Moss J.E.B., Griffeth, N.D., Graham, M.H.: Abstraction in Recovery Management. *Proc. Int. Conf. on Management of Data (SIGMOD)*, Washington, 1986.

[Moss87] Moss, J.E.B.: Log-Based Recovery for Nested Transactions. *Proc. 13th VLDB Conference,* 1987.

[Müller83] Müller, E.T., Moore, J.D., Popek, G. A: Nested Transaction Mechanism for LOCUS. *Proc. 9th Symposium on Operating Systems Principles*, ACM/SIGOPS, October 1983, pp. 71-89.

[Pu85] Pu, C., Noe, J.D.: Nested Transactions for General Objects: The Eden Implementation. University of Washington, Department of Computer Science, TR-85-12-03, December 1988.

[Rahm92] Rahm, E.: A Framework for Workload Allocation in Distributed Transaction Processing Systems. *Journal of Systems and Software*, 1992.

[Reuter82] Reuter, A.: Concurrency on High-Traffic Data Elements. *Proc. Conf. on Principles of Database Systems*, Los Angeles, March 1982, 83-93.

[Rothermel89] Rothermel, K., Mohan, C.: ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions, *Proc. 15th Intern. Conf. on Very Large Data Bases,* Amsterdam, Aug. 1989, 337-346.

[Rosenkrantz78] Rosenkrantz, D.J., Stearns, R.E., Lewis II, P.M.: System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems 3(2),* June 1978.

[Rukoz91] Rukoz, M. : *Hierarchical Deadlock Detection for Nested Transactions.Distributed Computing* (1991) 4: 123-129.

[Schwarz84] Schwarz, P.M., Spector, A.Z.: Synchronizing Shared Abstract Types. *ACM TOCS 2*, 3, 1984, 223-250.

[Spector83] Spector, A.Z., Schwarz, P.M.: Transactions: A Construct for Reliable Distributed Computing. *Operating Systems Review 17,* 2, 1983, 18-35.

[Spector88] Spector, A.Z., Pausch, R.F., Bruell, G. Camelot: A Flexible, Distributed Transaction Processing System. *Proc. Spring COMPCON '88*, IEEE, 1988.

[Walter84] Walter, B.: Nested Transactions with Multiple Commit Points: An Approach to the Structure of Advanced Database Applications. *Proc. 10th Int. Conf. on VLDB,* 161-171, Singapore, Aug. 1984, 161-171.

[Weikum86] Weikum, G. A Theoretical Foundation of Multi-Level Concurrency Control. *Proc. ACM SIGACT-SIGMOD*: Symposium on Principles of Database Systems, Cambridge, March 1986, 31-42.

[Weikum91] Weikum, G.: Principles and Realization Strategies of Multilevel Transaction Management, *ACM TODS*, Vol. 16, No. 1, 1991.

[Weikum84] Weikum, G., Schek, H.-J. Architectural Issues of Transaction Management in Layered Systems. *Proc. 10th Int. Conf. on VLDB*, Singapore, Aug. 1984, 454-465.

[Weinstein85] Weinstein, M., Page, T., Livezey, B., Popek, G.: Transactions and Synchronization in a Distributed Operating System. *Proc. 10th Symposium on Operating Systems Principles*, pp. 115-126, ACM/SIGOPS, December 1985.