In: Proc. of the 6th Int. Conf. on Database and Expert Systems Applications (DEXA'95), London, UK, Sept. 1995. pp. 604-613.

Concurrency Control in Nested Transactions with Enhanced Lock Modes for KBMSs

Fernando de Ferreira Rezende[1] and Theo Härder

Department of Computer Science - University of Kaiserslautern
P.O.Box 3049 - 67653 Kaiserslautern - Germany
Phone: +49 (0631) 205 3274|4031 - Fax: +49 (0631) 205 3558
E-Mail: {rezende|haerder}@informatik.uni-kl.de

Abstract. *Nested transactions (NTs) allow the decomposition of execution units and finer grained control over concurrency and recovery than conventional, flat transactions. Due to their characteristics, they provide adequate control structures for modeling transactions in several different environments. In this paper, we concentrate on the use of NTs in Knowledge Base Management Systems (KBMSs). With respect to those systems, the allowance for knowledge sharing is an important emerging point, which is going to be imperative for the complete success of KBMSs in the market. Nevertheless, to allow for knowledge sharing, special lock modes must be employed to adequately control the concurrency in KBMSs. We present enhanced lock modes tailored to the KBMS environment, which capture the abstraction relationships' semantics. Additionally, we couple these lock modes to an NT model allowing for upward as well as controlled downward inheritance of locks. Finally, on one hand, our NT model allows for an effective exploitation of intra-transaction parallelism, on the other hand, our enhanced lock modes make feasible the exploitation of the inherent parallelism in a knowledge representation approach.*

1 Introduction

When multiple users access a database (DB) 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 (CC) and should provide each concurrent user the illusion of referencing a dedicated DB. The classical transaction concept [7] defines a transaction as the unit of CC, i.e., the DB 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. In a DBMS, the component responsible for achieving this is transaction management which includes CC as a major function.

In the context of DBMSs, CC has been extensively studied by the DB community, and there is a vast amount of literature in this area. Unfortunately, CC has not received much of the attention of the Artificial Intelligence (AI) community, in spite of KBMSs are becoming more and more widespread and, accordingly, the demand for ever-larger knowledge bases (KBs) higher and higher. Due to the ever growing applicability of KBMSs, it is time to allow for knowledge sharing [2, 3, 14]. Consequently, multiple transactions should be able to run at the same time for better performance of such systems [4]. Finally, it is exactly in this point that transaction models and CC mechanisms with appropriate lock modes for KBMSs play a crucial role, because they are among the most important means for allowing large, multi-user KBs to be widespread.

In this paper, we try to fill this one more gap existing between DBMSs and KBMSs. To put it another way, we couple an enhanced CC technique for NTs with enhanced lock modes for KBMSs. The CC method for NTs we use is the one proposed by Härder and

1. Financially supported by the CNPq (National Council for the Scientific and Technological Development) of the Secretary for Science and Technology of Brazil.

Rothermel [10], and the enhanced lock modes for KBMSs are the ones proposed by Rezende and Härder [14, 15]. The main advantage of the former is the introduction of the concept of *controlled downward inheritance of locks*, which makes objects manipulated by a parent transaction available to its children[2]. In turn, the lock modes introduced by the latter capture more of the semantics contained in a KB graph, by means of an interpretation of its edges grounded in the abstraction relationships. This paper is organized as follows. In Sect. 2, we present a general CC method for NTs. In Sect. 3, the enhanced lock modes for KBMSs are introduced. Thereafter, we couple both together, showing how we use the enhanced lock modes for KBMSs in NTs (Sect. 4). Finally, we conclude the paper (Sect. 5).

2 Concurrency Control in Nested Transactions

2.1 An Overview of Nested Transactions' Nice Properties

When executing more complex transactions, it turns out that single-level transactions do not achieve optimal flexibility and performance. As a solution, the concept of NTs was popularized by Moss [13], where single-level transactions are enriched by an inner control structure[3]. 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. These aspects lead to advantages in a computing system [10], like: Intra-transaction parallelism, intra-transaction recovery control, explicit control structure, system modularity, and distribution of implementation.

In addition, NTs lead to some more advantages in the particular field of KBMSs. Among others, they provide adequate control structures for:

- *Methods inside methods*: In KBMSs, methods may recursively call other methods, thus producing a natural nesting of methods. The use of NTs is clearly well-suited to control such executions. In addition, it provides to users the possibility of explicitly controlling the executions of methods, allowing them to appropriately react in case of failure of any method, and to take the necessary corrective measures.
- *Complex functions*: Like methods, complex functions in KBMSs may recursively embody other less-complex functions. Similarly, those functions may be well-represented through NTs, e.g., considering each function call as a creation of a new subtransaction. Also, failure handling in functions' execution is made easier with NTs.
- *Virtual rule processing*: In KBMSs, a user may be interested in the results of some rule processing, but at the same time one may want to avoid any modifications in the KB (e.g., *what-if* questions). By means of NTs, a user may start a transaction inside another one, which may be rolled back later.

2.2 A Model of Nested Transactions

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 NTs. 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 also speak of *ancestors* and *descendants*. The ancestor (descendant) relation is the reflexive transitive closure of the parent (child) relation. We use the term *superior* (*inferior*) for the non-reflexive version of the ancestor (descendant). The set of descendants of a transaction

2. The Moss' CC method for NTs [13] allows only for upward inheritance of locks.
3. The ideas underlying the concept of NTs stem from Davies' *spheres of control* [5, 6].

together with their parent/child relationships is called the transaction's *hierarchy*. In the following, unless otherwise noted, we use the term *transaction* to denote both TL-transactions and subtransactions.

The properties defined for flat transactions are *atomicity*, *consistency*, *isolation*, and *durability* (ACID) [9]. In the NT model, these 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. 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. If the CC scheme introduced by Moss is applied, *isolated* execution is guaranteed for subtransactions. However, to increase intra-transaction parallelism, our enhanced scheme allows transactions belonging to the same TL-transaction hierarchy to share data in a controlled manner. The *durability* 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.

2.3 General Locking Rules

In this section, we present general locking rules for the model of NTs introduced previously. Nevertheless, before describing the rules, we shall introduce some terminology. The Moss' NT model [13] is based on the assumption that only leaf transactions acquire and use locks (i.e., are able to access shared objects), and hence no distinction is made between the locks explicitly acquired by a transaction and those acquired by inferiors and then passed on to their parents at commit time. However, such an assumption prohibits parent/child parallelism and therefore may limit the use of inherent parallelism. In our model, we enable maximum parallelism in a transaction hierarchy, allowing for parent/child as well as sibling parallelism. On the one hand, this degree of parallelism requires a sophisticated CC scheme. On the other hand, it permits arbitrary intra-transaction parallelism, i.e., all transactions of a TL-transaction hierarchy may be potentially executed concurrently.

Due to that, we need to distinguish the locks explicitly acquired by a transaction from the ones inherited from the children. Hence, in our model a transaction can *acquire* a lock on an object O in some mode M. Doing that, it *holds* the lock in mode M until its termination (represented by h:M). Besides holding a lock, a transaction can *retain* a lock in mode M (represented by r:M). 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). However, the same is not true for retained locks. A retained lock is only a place holder and indicates that transactions outside the hierarchy of the retainer cannot acquire the lock, but that descendants potentially can. As soon as a transaction becomes a retainer of a lock, it remains a retainer for that lock until it terminates. Finally, the general locking rules, which deal with the various situations of transaction management, are presented in Table 1. In particular, these locking rules are already expanded for allowing upward as well as (controlled) downward inheritance of arbitrary lock modes [10]. Rule TR1 deals with lock requests. The main point of this rule is that a transaction *retaining* a lock blocks other conflicting lock requests from transactions *outside* its hierarchy. Rule TR2 establishes the criteria for the inheritance of locks by the parent of a committing subtransaction. Rule TR3 governs the release of locks by committing TL-transactions, whereas TR4 the release of locks by aborting transactions. At last, rule TR5 allows for downward inheritance of locks, relinquishing the isolation among participating parent transaction and inferior subtransactions.

Table 1: General locking rules (Transaction Rules).

TR1	A transaction T may acquire a lock in mode M or upgrade a lock it holds to mode M if, first, no other transaction holds the lock in a mode that conflicts with M, and second, all transactions that retain the lock in a mode conflicting with M are ancestors of T.
TR2	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.
TR3	When a TL-transaction commits, it releases all locks it holds or retains.
TR4	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.
TR5	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 and holds it in mode M'.

Notice that if rule TR5 were omitted, we would get a generalization of Moss' scheme, which only provides for upward inheritance. The rules stated above require 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 no longer act as firewalls such that application code within a subtransaction had to cope with concurrency and recovery issues. In turn, with downward inheritance of locks, 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 uncommitted 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. Finally, a discussion about the correctness of this protocol may be found in [10].

3 Enhanced Lock Modes for KBMSs

Thus far, we have introduced an NT model with general locking rules and lock modes. In this section, we present the LARS (**L**ocks using **A**bstraction **R**elationships' **S**emantics) protocol for transaction synchronization in KBMSs [14, 15].

3.1 An Example Knowledge Base

KBMSs manage complex and structured objects, and also different types of abstraction relationships. In fact, abstractions turned out to be fundamental tools for knowledge organization, and one of the most important aspects of KBMSs is that objects can play different roles at the same time [11]. Consequently, the KBs features can be visualized as a superposition of the abstraction hierarchies (in fact Directed Acyclic Graphs (DAGs)) of generalization, classification, association, and aggregation, building altogether the so-called KB graph. It is beyond the scope of this paper a detailed discussion about the abstraction concepts, the reader is referred to [11, 12] for more details on this topic. In order to illustrate one such a KB graph, in Fig. 1 we provide an example of a restaurant KB. In order to restrict the KB to a rooted and connected graph, we have added the objects *global*, the only root of the whole graph, *sets*, the root of the association graph, *classes*, the root of the classification/generalization graph, and finally *aggregates*, the root of the aggregation graph. We provide such objects in order to have an adequate environment for the appliance of LARS. In addition, we assume that all objects (or schemas) are directly or indirectly related to the root *global*. When a schema is neither a class/instance, nor a set/element, nor a component/part, it is connected as a direct instance of *global*. In turn, all classes/instances, sets/elements, and components/parts are directly or indirectly related to the predefined schemas *classes, sets*, and *aggregates*, respectively. Moreover, we assume that the KB graph automatically stays in this form (rooted and connected) as changes undergo over time[4].

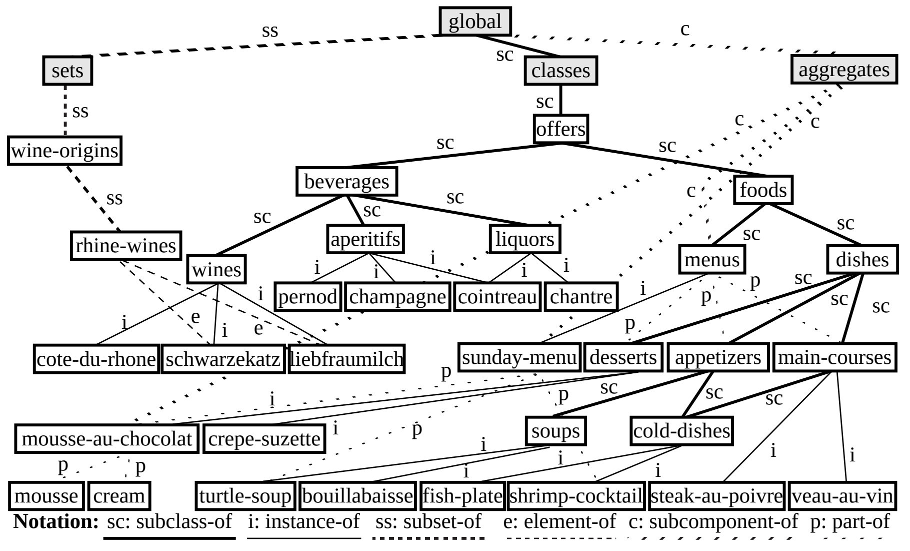

Fig. 1. A restaurant knowledge base.

3.2 LARS' Lock Modes

As a matter of fact, a KB graph is built through the superposition of the classification/generalization, association, and aggregation hierarchies (in fact DAGs). However, many accesses in a KB are directed to a particular hierarchy, and not to the KB graph as a whole. These observations build the main idea of LARS. In LARS, the KB graph is partitioned into those three main hierarchies, and hierarchical lock schemes [8] are applied on each one of them. As a result, a minimization of the locks is obtained. In addition, the granule of lock to be accessed by a transaction is more precisely defined, allowing it to lock just the objects it really needs to access. Following these logical partitions, LARS provides three distinct sets of lock types. Firstly, it has a *basic set* of lock modes, named: IR (Intention Read), IW (Intention Write), R (Read), RIW (Read Intention Write), and W (Write). However, it offers this basic set of lock modes to each one of the logical partitions, i.e., to the classification (recognized by a subscript c ($_c$) following the lock mode), association ($_s$), and aggregation ($_a$) graphs. These locks pertain respectively to the sets of *C_type*, *S_type*, and *A_type locks* (generally called *typed locks*). In Table 2, their semantics is presented in a compact form [15].

4. This representation and behavior are similar to the ones used by KRISYS [12] to represent KBs.

Table 2: Typed locks' semantics.

IR$_{c\|s\|a}$	gives intention shared access to the requested object and allows the requester to explicitly lock both direct **subclasses | subsets | subcomponents** of this object in R$_{c\|s\|a}$ or IR$_{c\|s\|a}$ mode, and direct **instances | elements | parts** in R$_{c\|s\|a}$ mode.
IW$_{c\|s\|a}$	gives intention exclusive access to the requested object and allows the requester to explicitly lock both direct **subclasses | subsets | subcomponents** of this object in W$_{c\|s\|a}$, RIW$_{c\|s\|a}$, R$_{c\|s\|a}$, IW$_{c\|s\|a}$ or IR$_{c\|s\|a}$ mode, and direct **instances | elements | parts** in W$_{c\|s\|a}$ or R$_{c\|s\|a}$ mode.
R$_{c\|s\|a}$	gives shared access to the requested object and implicitly to all direct and indirect **subclasses | subsets | subcomponents** and **instances | elements | parts** of this object.
RIW$_{c\|s\|a}$	gives shared and intention exclusive access to the requested object (i.e., implicitly locks all direct and indirect **subclasses | subsets | subcomponents** and **instances | elements | parts** of this object in shared mode and allows the requester to explicitly lock both direct **subclasses | subsets | subcomponents** in W$_{c\|s\|a}$, RIW$_{c\|s\|a}$, R$_{c\|s\|a}$ or IW$_{c\|s\|a}$ mode, and direct **instances | elements | parts** in W$_{c\|s\|a}$ or R$_{c\|s\|a}$ mode).
W$_{c\|s\|a}$	gives exclusive access to the requested object and implicitly to all direct and indirect **subclasses | subsets | subcomponents** and **instances | elements | parts** of this object.

3.3 The Lock Compatibilities

With respect to the compatibility of these lock modes, there are two distinct situations to be coped with by LARS. First, if the locks requested and granted give respect to the same set of objects (either C_type vs. C_type, or S_type vs. S_type, or A_type vs. A_type), then the compatibility matrix to be followed is the same of the Multigranularity Locks protocol known from the literature [8] (Table 3).

Table 3: Compatibility matrix for typed locks of the same type.

		Granted Mode [c | s | a]				
		IR	IW	R	RIW	W
Requested Mode [c | s | a]	IR	✓	✓	✓	✓	
	IW	✓	✓			
	R	✓		✓		
	RIW	✓				
	W					

The second situation with respect to the compatibility of the typed locks is the one where both are of different types (either C_type vs. {S_type or A_type}, or S_type vs. {C_type or A_type}, or A_type vs. {C_type or S_type}). In this case, the compatibility of the lock modes is not the same as above, because distinct sets of objects are being dealt with. In [15], a detailed discussion on this topic may be found. Here we limit to presenting the compatibility matrix (Table 4) and making some comments. The main point of this compatibility matrix is that conflicting lock modes applied to requests of the same abstraction hierarchy may become compatible when issued for different abstraction hierarchies, e.g., IW$_c$ and W$_a$. In general, there are no conflicts between locks in different hierarchies if one of them is an intention lock. Only non-intention locks of different hierarchies conflict like ordinary R and W locks. The reason is simply that an intention lock in hierarchy *h* only 'protects' paths along hierarchy *h*. An R or W lock in another hierarchy *g* only implicitly locks objects reachable by hierarchy *g*. In the absence of multiple abstraction relationships to objects, one talks about disjoint sets of objects. Objects belonging to different hierarchies are implemented such that distinct parts of an object implement different hierarchies. Other object data can be accessed independently of the hierarchy that has been used to locate the object. This is the only chance for conflicts, and is covered by R/W and W/W conflicts. Multiple abstraction relationships to objects are discussed in the next section.

3.4 Accessing Implicitly Locked Objects

As a matter of fact, multiple abstraction relationships involving an object may lead to problems with the implicit locks, so that the isolation property of transactions may be corrupted. Actually, an interference arises whenever an object with two or more parents (from now on called a *bastard*, in order to be differentiated from an object with only one parent, a *purebred*) is implicitly locked by one of them [14]. The implicit lock on a child object is only visible if it is accessed through a specific path of the graph. In order to find out possible conflicts with implicitly locked bastards, all superiors or inferiors of an object may be accessed. For this purpose, all relationships have to be represented in a bidirectional way. In [14, 15], we discuss many possible alternatives for avoiding conflicts in such situations. We have chosen for LARS a kind of *lazy evaluation strategy* for lock conflict resolution with implicitly locked bastards [15]. Following this approach, a transaction may request and be granted an explicit lock without further proceedings. However, just before effectively accessing an implicitly locked bastard, it must verify whether this object is already locked in a conflicting mode by another transaction or not. If so, it must wait until this lock is released. If not, it sets an explicit lock on this object, signalling that it has accessed it. This lock acts like a tag in the object indicating that it has been already accessed via another parent of it. The key observation in this approach is that a transaction needs to explicitly lock only those bastards which it actually accesses, leaving the others for the concurrent access by other transactions.

Table 4: Compatibility matrix for typed locks of distinct types.

		Granted Mode [c | s | a]				
		IR	IW	R	RIW	W
Requested Mode [s or a | c or a | c or s]	IR	✓	✓	✓	✓	✓
	IW	✓	✓	✓	✓	✓
	R	✓	✓	✓	✓	
	RIW	✓	✓	✓	✓	
	W	✓	✓			

3.5 The Locking Rules

The LARS' locking rules to be followed by transactions when requesting locks on objects in a KB are presented in Table 5. A proof of the correctness of LARS may be found in [15]. Notice that these rules are somewhat independent from the rules TR1-TR5 presented in Table 1. We deliberate about this topic in the next section, when coupling the LARS' lock modes to the NT model. Before explaining these rules, it is convenient to notice that transactions are allowed to directly set locks in the root object in any mode, and that LARS always produces *strict executions* [1], i.e., it requires the locks of a transaction to be released only at its termination (either commit or abort).

Table 5: Locking rules (Object Rules).

OR1	Before requesting an $IR_{c\|s\|a}$ lock on an object, the requester must cover a path from the object to the root with $IR_{c\|s\|a}$ or $IW_{c\|s\|a}$ locks.
OR2	Before requesting an $IW_{c\|s\|a}$ lock on an object, the requester must cover a path from the object to the root with $IW_{c\|s\|a}$ or $RIW_{c\|s\|a}$ locks.
OR3	Before requesting an $R_{c\|s\|a}$ lock on an object, the requester must cover a path from the object to the root with $IR_{c\|s\|a}$ or $IW_{c\|s\|a}$ locks. In addition, before accessing any implicitly locked bastard descendant, the requester must set an $R_{c\|s\|a}$ lock on it.
OR4	Before requesting an $RIW_{c\|s\|a}$ lock on an object, the requester must cover a path from the object to the root with $IW_{c\|s\|a}$ or $RIW_{c\|s\|a}$ locks. In addition, before accessing any implicitly locked bastard descendant, the requester must set either a) an $R_{c\|s\|a}$ lock on it, if it is a leaf object, or b) an $RIW_{c\|s\|a}$ lock on it, if it is a non-leaf object.
OR5	Before requesting a $W_{c\|s\|a}$ lock on an object, the requester must cover a path from the object to the root with $IW_{c\|s\|a}$ or $RIW_{c\|s\|a}$ locks. In addition, before accessing any implicitly locked bastard descendant, the requester must set a $W_{c\|s\|a}$ lock on it.

The rule OR1 states that an intention read lock (from the C_type, S_type, or

A_type) on a non-root object must be preceded by either intention read or intention write locks (from respectively the C_type, S_type, or A_type) on at least one parent of this object, and so recursively until the root object is reached. The rule OR2 has a similar meaning, but for the intention write locks, requiring that they must be preceded by intention read or read intention write locks on at least one path from that object to the root object. The rule OR3 states, first of all, that a read lock on a non-root object must be covered by intention read or intention write locks on at least one path from this object to the root object. Thereafter, it requires that a transaction must explicitly lock the accessed bastard descendants[5]. This is basically required for avoiding conflicts with implicitly locked bastards and thus putting in practice the lazy evaluation strategy followed by LARS. The rules OR4 and OR5 have a similar meaning, but for read intention write and write locks respectively.

4 Using Enhanced Lock Modes for KBMSs in Nested Transactions

In the last sections, we have introduced the essential concepts of both generalized locking rules for NTs and appropriate lock modes for objects in a KB. Hence, we now combine both together. For the transaction hierarchy, our generalized transaction rules TR1-TR4 (Table 1) apply. Furthermore, when acquiring a lock on an object, we have to consider the additional object rules OR1-OR5 (Table 5) resulting from the object hierarchies. As far as acquiring locks is concerned, the rules obtained for the transaction hierarchy and the object hierarchy must be satisfied independently. However, as discussed in [10], arbitrary inheritance of hierarchical locks may cause severe consistency problems. The key observation pointed out in [10] about this is that downgrading a lock without considering the whole object hierarchy may lead to inconsistencies. Additionally, similar observation is valid for upgrading locks in object hierarchies. In this paper, we are not going to deliberate about this topic anymore. Instead, we directly adopt the solutions proposed in [10]. The reader is referred to this paper for a detailed discussion about such inconsistencies when arbitrary inheritance of locks on object hierarchies is allowed.

4.1 Upgrading hierarchical locks

The key solution to avoid anomalies is that, since upgrading the locks on an object and superior objects are not performed atomically, *upgrading should be done in a root-to-leaf direction* [10]. Of course, an upgrade operation can only take place if the generalized transaction rules TR1-TR4 (Table 1) are fulfilled. However, due to the upgrade operation, locks held by the upgrading transaction on inferior objects may become useless. For example, when a lock on a class C is upgraded from RIW_c- to W_c-mode (lock escalation [8, 1]), all locks held by the upgrading transaction on instances of C are no longer needed. Our approach to handling those useless locks is to release them as part of the upgrade operation. Of course, those locks on bastards are still held by the upgrading transaction, since LARS' lazy evaluation strategy for avoiding conflicts with implicitly locked bastards requires locks on them anyway.

Fig. 2 illustrates an upgrade operation executed by a transaction A running in our KB (Fig. 1). There, A has acquired W_c locks on the objects *steak-au-poivre* and *veau-au-vin* (A:h:W_c) and according to LARS' locking rules for the object hierarchies, intentions (A:h:IW_c) on their superiors. Thereafter, A upgraded the IW_c lock held by it on

5. There may be situations where a descendant may have two edges pointing to the same parent. For example, when an object is at the same time instance and element of the same object. In such situations, the object is considered to be a bastard, no matter whether the parents are the same object.

main-courses to the W_c-mode. Due to this operation, the lock on *veau-au-vin* became useless, and was therefore discarded. On the other hand, the lock on *steak-au-poivre* was still held by A, because this object is a bastard and therefore must stay locked. Notice that the objects *cold-dishes*, *fish-plate* and *shrimp-cocktail* became implicitly locked by A's upgrade operation. However, *cold-dishes* as a bastard was not automatically explicitly locked. It will be explicitly locked if and only if A tries to access it, due to LARS' lazy evaluation strategy. Also notice that the objects *fish-plate* and *shrimp-cocktail* may be not accessed by A before A explicitly locks the bastard *cold-dishes*.

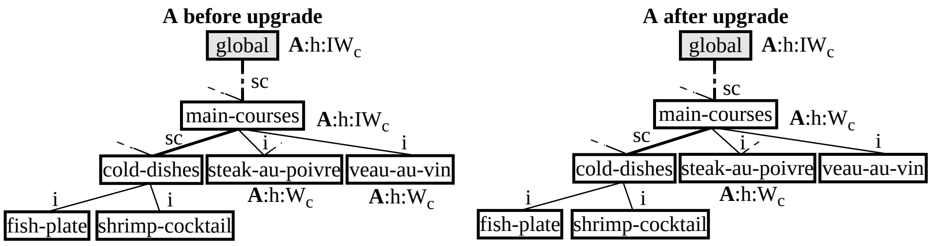

Fig. 2. Upgrading locks on objects in a classification hierarchy.

4.2 Downgrading hierarchical locks

Downgrading a lock held by a transaction A 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. However, downgrading the lock on O may require downgrading locks held by A on inferior objects of O. Table 6 lists for each possible mode to which the lock on O can be downgraded, the modes in which A can hold locks on inferior objects of O without leading to inconsistencies. As can be seen, the downgrade of a lock (as for upgrade) is adjusted in accordance to the type of lock. For example, if the lock is downgraded to IR_c-mode, A can hold inferiors of O in null-, IR_c-, or R_c-mode. If inferiors 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* [10].

Table 6: Possible modes to be held on inferior objects after downgrading a lock.

Lock of A on object O downgraded to mode	Consistent modes for locks of A on inferiors of O
- (null)	-
$IR_{c\|s\|a}$	-, $IR_{c\|s\|a}$, $R_{c\|s\|a}$
$IW_{c\|s\|a}$	-, $IR_{c\|s\|a}$, $IW_{c\|s\|a}$, $R_{c\|s\|a}$, $RIW_{c\|s\|a}$, $W_{c\|s\|a}$
$RIW_{c\|s\|a}$	-, $IW_{c\|s\|a}$, $R_{c\|s\|a}$, $RIW_{c\|s\|a}$, $W_{c\|s\|a}$
$R_{c\|s\|a}$	$R_{c\|s\|a}$

By observing these rules, consistency-preserving downward inheritance of locks may be achieved. Control of lock usage is then possible by downgrading to the appropriate modes. In the scenario of Fig. 3, the W_c lock held by a parent transaction A on *main-courses* (A:h:W_c) has been downgraded to IW_c mode (A:r:W_c:h:IW_c), whereas the W_c lock on *steak-au-poivre* to R_c (A:r:W_c:h:R_c) and the W_c lock on *cold-dishes* to null-mode (A:r:W_c:h:-). This observation allows for selective control of access to inferiors of an object. Hence, the transaction B, child of A, may write any inferior (with respect to the classification hierarchy) of *main-courses*, but *steak-au-poivre*.

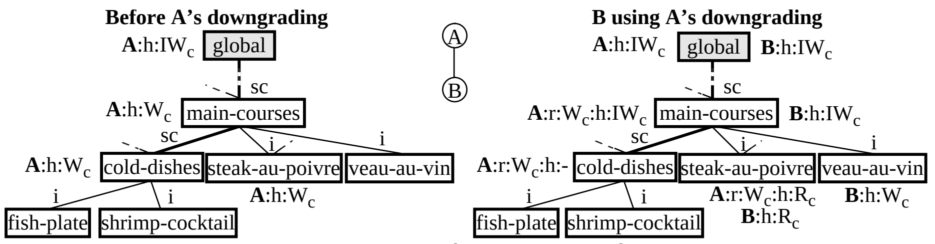

Fig. 3. Selective downgrading of locks in a classification hierarchy.

In summary, downgrading of entire subtrees is necessary for hierarchical objects to guarantee consistency of downward inheritance in NTs. That is, if a lock held by a transaction A on an object O is downgraded, it might be necessary to downgrade locks held by A on inferiors of O. Due to that, downgrading must be performed in leaf-to-root direction. In a contrary way, upgrading must be performed in a root-to-leaf direction.

5 Conclusions

We have presented an investigation of CC in NTs. The focus of our paper has primarily been on achieving a high degree of intra-transaction parallelism within NTs. In addition, our model allows for *controlled downward inheritance* of locks, in order to enable a transaction to restrict the access mode of its inferiors for an object. We have also focused the particular field of KBMSs, which has a need for specialized lock modes as well as multi-level object hierarchies offering efficient ways to lock granules of varying sizes. Inside this context, we have presented the LARS protocol, a CC technique with enhanced lock modes tailored for KBMSs. The most important point of LARS is the partition of the KB graph into many logical ones, allowing transactions to concurrently access such partitions through different points of view. LARS provides many different lock types and takes the necessary precautions with respect to the dynamism of the KB graph. By this means, a high degree of potential concurrency is obtained, exploiting the inherent parallelism in a knowledge representation approach.

References

[1] Bernstein, P.A., Hadzilacos, N., Goodman, N.: *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, USA, 1987.
[2] Chaudhri, V.K.: *Transaction Synchronization in Knowledge Bases: Concepts, Realization and Quantitative Evaluation*. Ph.D. Thesis, University of Toronto, Toronto, Canada, 1994.
[3] Chaudhri, V.K., Hadzilacos, V., Mylopoulos, J.: Concurrency Control for Knowledge Bases. In: *Proc. of the 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, Cambridge, USA, 1992.
[4] Chaudhri, V.K., Hadzilacos, V., Mylopoulos, J., Sevcik, K.C.: Quantitative Evaluation of a Transaction Facility for a KBMS. In: *Proc. of the 3rd CIKM*, Gaithersburg, USA, Nov. 1994.
[5] Davies, C.T.: Recovery Semantics for a DB/DC System. In: *Proc. of the ACM Nat. Conf.*, USA, 1973.
[6] Davies, C.T.: Data Processing Spheres of Control. *IBM Systems Journal*, 17 (2), 1978.
[7] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19 (11), Nov. 1976.
[8] Gray, J.N., Lorie, R.A., Putzolu, G.R., Traiger, I.: Granularity of Locks and Degrees of Consistency in a Shared Data Base. *Proc. of the IFIP Working Conf. on Modelling in DBMSs*, North-Holland, 1976.
[9] Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15 (4), 1983.
[10] Härder, T., Rothermel, K.: Concurrency Control Issues in Nested Transactions. *Journal of the VLDB*, 2 (1), 1993.
[11] Mattos, N.M.: Abstraction Concepts: The Basis for Data and Knowledge Modeling. In: *Proc. of the 7th Int. Conf. on Entity-Relationship Approach*, Rom, Italy, Nov. 1988.
[12] Mattos, N.M.: *An Approach to Knowledge Base Management - Requirements, Knowledge Representation, and Design Issues*. Doctor Thesis, University of Kaiserslautern, Kaiserslautern, Germany, 1989.
[13] Moss J.E.B.: *Nested Transactions: An Approach to Reliable Distributed Computing.* M.I.T. Press, 1985.
[14] Rezende, F.F., Härder, T.: A Lock Method for KBMSs Using Abstraction Relationships' Semantics. In: *Proc. of the 3rd Int. Conf. on Information and Knowledge Management*, Gaithersburg, USA, Nov. 1994.
[15] Rezende, F.F., Härder, T.: *Capturing Abstraction Relationships' Semantics for Concurrency Control in KBMSs*. ZRI Report No. 6/94, University of Kaiserslautern, Kaiserslautern, Germany, Nov. 1994.