Exploiting Abstraction Relationships' Semantics for Transaction Synchronization in KBMSs

Fernando de Ferreira Rezende 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 - Currently, knowledge sharing is turning out to be a crucial point to be supported by Knowledge Base Management Systems (KBMSs). We propose an approach for transaction synchronization in KBMSs - LARS (Locks using Abstraction Relationships' Semantics). We show how we obtain serializability of transactions thereby providing different locking granules. The main benefit of our technique is the high degree of potential concurrency, which is obtained by means of a logical partitioning of the knowledge base (KB) grounded in the abstraction relationships, and the provision of many lock types to be used on the basis of each partition. By this way, we capture the abstraction relationships' semantics contained in a KB graph for transaction synchronization purposes and enable the exploitation of the inherent parallelism in a knowledge representation approach.

Keywords - Transaction synchronization, concurrency control, locking, knowledge base management systems, object-oriented database systems.

1. Introduction

In recent years, the use of KBMSs is becoming more and more widespread and, accordingly, the demand for ever-larger KBs higher and higher. Nowadays, the main challenge of the KBMS research is to try the successful adaptation of such systems to real-life production environments [49]. In this scope, concurrency control (CC) techniques for KBMSs play a crucial role, because they are among the most important means for allowing large, multi-user KBs to become a reality [55, 10].

In this paper, we present our approach for transaction synchronization in KBMSs. The main goal we have in mind is the provision of serializability [24] for ACID [28] transactions. Among the most important classes of CC algorithms are *locking*, *timestamps*, and *serialization graphs* [5]. In particular, the class of locking-based algorithms has shown its practicality and performance. Additionally, locking-based algorithms have special solutions for graph structures, the abstractions for KBs that appear to be the most appealing [11]. Thus, we have chosen to develop our technique based on locking.

With respect to locking, we could consider several approaches - *predicate locks* [17], *the two-phase locking (2PL) protocol* [17], *the multigranularity locking (MGL) protocol* [24], etc. In particular, we are more interested in granular locks, because they provide transactions the possibility of choosing, among different locking granules, the most appropriate one to accomplish their tasks. In addition, the notion of implicit locks significantly minimizes the number of locks to be set by transactions. These are some of the reasons which lead us to use the power and elegance of granular locks also in the KBMS environment.

This paper is organized as follows. After providing short discussions on some particular issues in KBMSs (Sect. 2), we present an overview of MGL and point out the main problems of its pure appliance in the KBMS environment (Sect. 3). Then, we introduce our approach for transaction synchronization in KBMSs (Sect. 4). Thereafter, we discuss related work (Sect. 5) and finally conclude the paper (Sect. 6).

2. Particular KBMS Issues

2.1 Knowledge Bases

Perhaps due to the existence of several journals, conferences, communities, etc. concerned with *knowledge-based systems,* there are so many definitions characterizing the meaning of *knowledge* and *knowledge bases*. Putting aside a discussion on the philosophical meaning of these concepts and without trying to speculate what a KBMS might be and in what way it may differ from conventional Database Management Systems (DBMSs), we refer here to a specific definition given by Levesque and Brachman [37], which is widely accepted in the knowledge representation community:

> *"A knowledge base has explicit structures representing the knowledge of the system which determine the actions of the system. ... It is not the use of a certain programming language or a data-structuring facility that makes a system knowledge-based."*

This definition views a KB as a system with explicit structures representing the knowledge. This is exactly the most important characteristic of such a system for our purposes. Any data model which explicitly represents the knowledge and, therefore, explicitly encodes the knowledge and the semantic structure of an application domain may use the results we present in this paper. Such an explicit representation of knowledge is found not only in knowledge-based systems, but also in several, especially object-oriented, data models. Even many conventional relational database (DB) systems can satisfy this requirement. The essential feature is that they can be visualized as Directed Acyclic Graphs (DAGs), as explained in the following. Therefore, unless otherwise noted, we use in this paper the terms *databases* and *knowledge bases* interchangeably meaning that the results of our work are general and applicable to a broad class of applications.

2.2 The Abstraction Concepts

KBMSs manage complex and structured objects and different types of abstraction relationships. In fact, abstractions turned out to be fundamental tools for knowledge organization. An important aspect of KBMSs is that objects can play different roles at the same time - the object-centered representation [42]. Consequently, the KB features can be visualized as a superposition of the abstraction hierarchies (in fact DAGs) of generalization and classification, association, as well as aggregation, building altogether the so-called *KB graph*. It is beyond the scope of this paper to begin a detailed discussion about the abstraction concepts. The interested reader is referred to [42, 43]. In order to illustrate one such a KB graph, in Fig. 1 we provide an example of a restaurant KB. Notice that the purpose of this scenario is merely to illustrate our solution for knowledge sharing, rather than schema design issues. This scenario will serve as a running example along this paper.

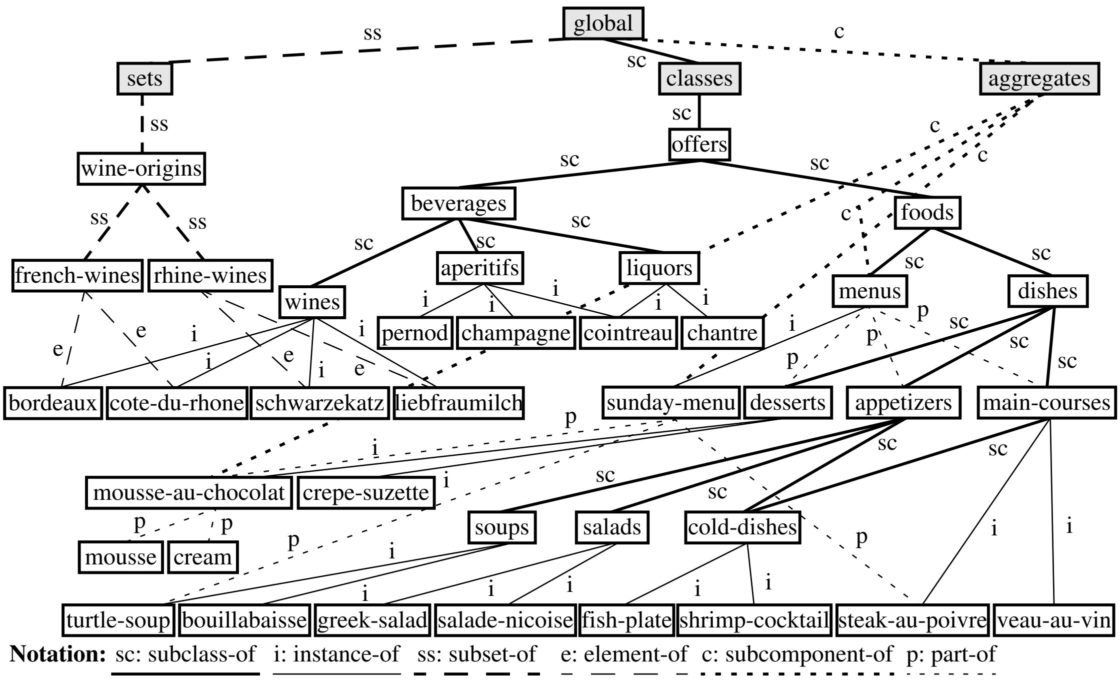


Figure 1: A restaurant knowledge base.

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 our protocol. In addition, we assume that all objects (or schemas) are directly or indirectly related to *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.[1]

2.3 Expressiveness of KB Languages

The expressiveness of the KB languages is generally greater than that usually provided by traditional systems, e.g., relational languages. In knowledge languages, the query formulation can make use of the different abstraction relationships and the complex structure of the objects. Operations, usually set-oriented, on these structures may refer to ancestors as well as subordinate objects to derive or modify KB information. Hence, an influencing factor in the query evaluation mechanisms of KBMSs is the representation of the edges in KB graphs. In general, the edges in KBs may be represented either in a unidirectional way (one link, top-down) or in a bidirectional way (two links, top-down and bottom-up), depending on the implementation characteristics of each system in particular. Representing the edges by unidirectional links has the advantage of less maintenance overhead, since maintaining one link up-to-date is less expensive than two links, of course. However, with a unidirectional representation of edges many significant queries may not be answered (at least at the same costs when bidirectional links are provided). Furthermore, questions involving the inheritance of attributes may be made much more difficult. All in all, the costs for maintaining them up-to-date are paid off when evaluating the queries much more efficiently.

In this paper, we assume that the edges in a KB graph are bidirectionally represented. Like the KBMSs' query evaluation components, we use the power of bidirectional links also in LARS. However, it is convenient to notice here that such bidirectional links are used in LARS just in the representation of the abstraction concepts, and **not** in the access paths (indices like B*-trees). This will become clear in the later sections.

1. This representation and behavior are very similar to the ones used by KRISYS [43] to represent KBs.

2.4 Behavioral Aspects of Objects

The behavioral aspect of objects can be expressed by the methods, rules, demons[2], etc. commonly provided by KBMSs. In the last few years, there have been considerable efforts in order to approach the limits of concurrency by exploiting the semantics of objects and their operations (methods) when synchronizing transactions [20, 9, 35, 19, 23, 40, 2, 59, 48, 18, 58, 52, 13, 26]. The main idea behind these approaches is to break the serializability of transactions, allowing non-serializable schedules to be produced, as long as they preserve the consistency and are acceptable to the system users.

When we started analyzing the challenging issue of knowledge sharing, we investigated the details of such approaches thoroughly. Nevertheless, we came to the conclusion that we should not exploit the methods' semantics to allow for non-serializable schedules due to several reasons. Essentially, mainly due to schema evolution, complications on recovery, and direct accesses to objects which bypass the object encapsulations. Thus, in our technique, methods are not treated separately; data references in the body of methods are synchronized as ordinary read and write operations.

3. The Multigranularity Locking Protocol

The basic idea of MGL [24] comes from the choice of different lockable units to be locked by the system in order to ensure consistency and to provide isolation. Its main benefit is that it allows lockable units of different granularities to coexist in the same system. Moreover, this protocol created the notion of *implicit locks*, stating that by putting a lock on a granule, all its descendants become implicitly locked, without the necessity of setting further locks. Lastly, this protocol introduced the so-called *intention locks*, in order to prevent locks on the ancestors of a node which might implicitly lock it in an incompatible mode. Basically, the lock modes of MGL are: IS (Intention Share), IX (Intention eXclusive), S (Share), SIX (Share Intention eXclusive), and X (eXclusive). These are then applied to the nodes in a lock graph - a hierarchy or a DAG (Fig. 2) [24].

MGL is designed for a single organization hierarchy, extended to DAGs in case of index structures. Particularly in the case of DAGs, MGL requires that, before requesting an X mode access to a node, all superiors of the node must be covered with IX (or greater) locks [22, 24]. One question arises: How may transactions know which are the superiors of a node? The data model, to which MGL may be applied, provides a strict separation

2. Demons are procedures to be attached to attributes of objects that are automatically activated when these attributes are accessed. In the DB terminology, this notion is similar to *triggers*.

between data and meta-data, a separate DB catalog. Hence, transactions may access, e.g., the DB catalog, and learn that (using Fig. 2) a *record* is always contained in a *file* and pointed to by an *index*, in turn, a *file* and the respective *index* are contained in an *area*, and at last an *area* is contained in a *database*.

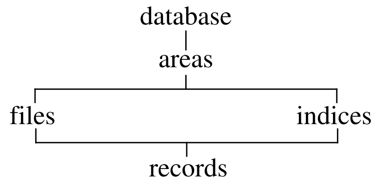


Figure 2: A lock graph for granular locks.

However, in the data (knowledge) models provided by KBMSs, the object concept is completely symmetric, such that this separation data/meta-data (like, for example, in the relational model) no longer exists. The superiors of a node (an object in the KB graph) may be arbitrarily chosen, accordingly to the semantics of the application being modeled. More importantly, this information may be dynamically changed as a KB undergoes changes over time. Exemplifying, by means of the classification DAG of Fig. 3, how could a transaction know which are the superiors of any class, say $class_k$? A transaction, obeying MGL, does need such information in order to lock a class in X mode. This information is not statically available in KBMSs, as in usual DB catalogs. In addition, how could a transaction be sure that by putting an X lock on a class, no one of its subclasses would be accessed in a conflicting mode by another transaction? Using Fig. 3, a transaction putting an X lock on $class_k$ and IX on all its superiors (..., $class_i$, ...) would have no guarantee that another transaction would not access $class_n$ by similarly putting an X lock on $class_j$ and IX on all its superiors (..., $class_i$, ...). Therefore, transactions obeying MGL may get into troubles with implicitly locked objects, as long as they implicitly lock those objects in conflicting modes via different paths of the graph.

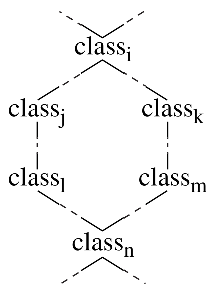


Figure 3: A classification DAG.

Summarizing, if there would be just *strict* abstraction hierarchies in KBMSs, MGL would be adaptable and could work without problems. However, when *overlapping*, *multi*-abstraction hierarchies are possible, MGL fails due to the implicit locks on objects with multiple direct parents, because those objects might be implicitly locked in incompatible modes by different transactions using different paths along a single kind of hierarchy or different paths using different hierarchies. Finally, the richness of the KB structure makes the synchronization substantially more complicated in KBMSs.

Another problem of directly applying MGL in the KBMS environment is the non-use of the semantically rich structure represented by KB graphs. To put it another way, using MGL, when a shared/exclusive lock on a node is granted to a transaction, all descendants of this node are implicitly locked in the same mode, independently of the relationship the descendants have to the ancestor. However, if such an object is, at the same time, a set (component), all its subsets (subcomponents) and elements (parts) become also implicitly locked. This needlessly restricts the access to those objects by other transactions. Therefore, with such a behavior, many objects may be locked unnecessarily, because it is not possible to precisely specify which kind of descendants should be implicitly locked, and thus the overall concurrency may be affected negatively.

4. The LARS Protocol

4.1 The Main Goals

Before presenting the LARS - Locks using Abstraction Relationships' Semantics - protocol, this section lists the main goals we aimed for in our work.

- Different Granules of Lock

  In KBMSs, accesses normally refer to different granules of objects. Hence, we strive for providing different granules of locking based on objects and for making use of implicit locks in order to improve the overall performance, in particular the lock manager's one.

- Multiple Abstraction Relationships to Objects

  The objects in a KB build a complex and dynamic graph structure, where multiple paths to objects at any time are possible. This characteristic imposes an extra task for us, since we have aimed for using implicit locks, because an object with multiple parents may be accessed via a path that does not have any lock on it.

- Semantics of the Relationships between Objects

  The relationships between objects in KBs are based on the abstraction concepts. In turn, each abstraction concept has a particular and special semantics. We target at using the abstraction relationships' semantics in order to improve the concurrency.

4.2 The Basic Idea

The key feature of KBs is the presence of several semantic relationships. The basic idea which originates LARS is based on this feature: The KBs can be partitioned into several graphs, according to the semantics of those several relationships. Thus, we create three different logical partitions from the whole KB graph. These are called the *classification* (which includes also generalization), *association*, and *aggregation* graphs. Finally, we apply granular locks to each graph. By this way, we provide users with the possibility of looking at a KB and abstracting from it just the partition to be worked out. On one hand, we acquire a minimization of the number of locks in comparison with, for example, a conventional approach with shared and exclusive lock modes, where every touched object must be locked. On the other hand, we define more precisely the granule of lock to be accessed by a transaction, allowing it to lock just the objects it really needs to access.

4.3 The Lock Modes

Following these logical partitions, we have created three distinct sets of lock types. Hence, similar to MGL, we have a *basic set* of lock modes, named: IR (Intention Read), IW (Intention Write), R (Read), RIW (Read Intention Write), and W (Write). However, we have this basic set to each logical partition - classification (recognized by a subscript *c* ($_c$) following the lock mode), association ($_s$), and aggregation ($_a$) graphs. We named those locks as pertaining respectively to the sets of *C_type*, *S_type*, and *A_type locks* (in general, we call them *typed locks*). Table 1 presents, in a compact form, their semantics.

Table 1: 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.

4.4 The Lock Compatibilities

Two lock requests for the same object by two different transactions are said to be *compatible* if they can be granted concurrently [22]. With respect to the compatibility of the above mentioned lock types, we have two distinct situations to cope with. These are discussed in the following.

Compatibility of Locks of Identical Types

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 MGL known from the literature [24, 22] (Table 2).

Table 2: Compatibility matrix for locks of identical types.

Granted Mode [c | s | a]

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

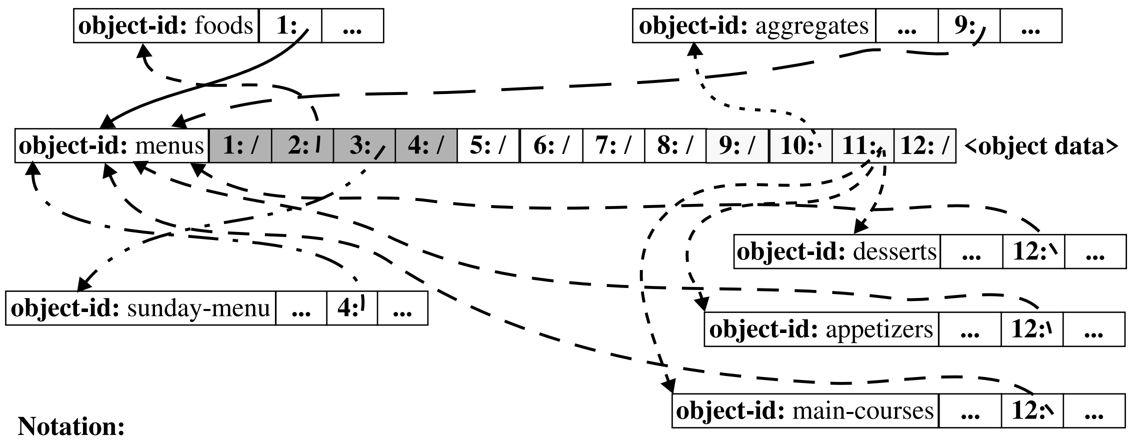
Compatibility of Locks of Distinct Types

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 we are dealing with distinct sets of objects. Let us try to build such a compatibility matrix. To do that, we need to compare pairs of lock modes in order to find out whether conflicts may happen or not when both are granted simultaneously.

Let us use as a general example an extreme case, IW and W lock modes of different types being requested on a same object, say *menus* (see Fig. 1). Suppose we have a physical representation of *menus* like the one sketched in Fig. 4. As can be seen, all relationships are bidirectionally represented. Let us consider them top-down, like the way the transactions are going to request locks. Suppose a transaction, say T1, comes from *aggregates* and wants to write[3] the object *menus* and all its parts, namely *desserts*,

3. In the scope of this paper, we use the term *'to write an object'* as meaning an update operation in an existing object. For insert and delete operations (see Sect. 4.7), we explicitly use the terms *'to insert an object'* and *'to delete an object'*, respectively.

appetizers, and *main-courses*. T1 must require then, in addition to an IW_a on *aggregates*, a W_a on *menus*. Suppose no other transaction is actuating on *menus* in the moment, so that this lock may be immediately granted to T1. Once granted, T1 is able to write the object *menus* and all its parts, accordingly to the semantics of W_a. Considering only the object *menus* in Fig. 4, we can say that T1 is able to write the fields of menus from 9 until end. The fields 1-8 may neither be accessed nor traversed by T1 with its current lock, just because the semantics of a W_a does not comprise the objects pointed by those fields (see Table 1). In other words, T1 may write no descendants of *menus*, but only its parts. Let us go ahead with another transaction, say T2. Suppose T2 comes from the object *foods* to *menus*. In addition, suppose T2 sets an IW_c on *foods* and tries to set an IW_c on *menus*, in order to set, further, a W_c on *sunday-menu*, an instance of *menus*. If we analyze the semantics of IW_c, we notice that this lock represents an intention to write subclasses and instances of an object. In our example, it represents an intention to write the object *sunday-menu*. Using Fig. 4, we may notice that T2 wants to traverse only the fields 1-4 of *menus*, the ones pointing to its subclasses and instances. Therefore, although T1 has a W_a on *menus*, the lock manager may grant this IW_c to T2, because both transactions are accessing different fields of *menus*, and so they may not stay in conflict with one another.[4] Therefore, when applied to distinct sets of objects, IW and W lock modes are compatible.



Notation:

generalization/classification	*association*	*aggregation*
1: has-subclasses	**5:** has-subsets	**9:** has-subcomponents
2: subclass-of	**6:** subset-of	**10:** subcomponent-of
3: has-instances	**7:** has-elements	**11:** has-parts
4: instance-of	**8:** element-of	**12:** part-of

Figure 4: Physical representation of the object *menus*.

4. Notice that if T2 would require a W_c on *menus*, it would conflict with T1, because T2 would be able to write not only the fields 1-4, but also the ones after 12, i.e., the object data of *menus*.

Our discussion has shown that conflicting lock modes applied to requests of the same abstraction hierarchy may become compatible when issued for different hierarchies, e.g., IW_c and W_a. Table 3 shows the compatibility for typed locks of distinct types. 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 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. In Table 3, the boxes marked with darker shadows are where our technique offers more concurrency, all of that due to the consideration given to the semantics of the edges in a KB graph.

Table 3: Compatibility matrix for locks of distinct types.

Granted Mode [c | s | a] (columns); Requested Mode [s or a | c or a | c or s] (rows)

	IR	IW	R	RIW	W
IR	✓	✓	✓	✓	✓
IW	✓	✓	✓	✓	✓
R	✓	✓	✓	✓	
RIW	✓	✓	✓	✓	
W	✓	✓			

4.5 Accessing Implicitly Locked Objects

In Sect. 3 and Sect. 4.1, we have briefly discussed that multiple abstraction relationships to an object may lead to problems with the implicit locks. As a matter of fact, an interference arises whenever an object with multiple parents is implicitly locked via one of them. From now on, we call these objects with multiple parents *bastards*, in contrast to *purebreds*, objects with only one parent.

To illustrate this problem, let us refer to Fig. 5. There, both transactions T1 and T2 required an IW_c lock on *beverages* and were granted because they are compatible. Thereafter, T1 followed the path to *aperitifs* and locked it in W_c mode. Then, it received an exclusive lock on *aperitifs* and implicitly on its instances (*pernod*, *champagne*, and *cointreau*). Following another path, T2 locked *liquors* in W_c mode and implicitly received exclusive locks on its instances too (*cointreau* and *chantre*). T1 and T2 may get into troubles with one another. The problem is that none of them knows a priori which are the instances of those objects due to the dynamism of the KB graph; hence, both requested a lock on a node in the hope that its descendants were locked as a whole implicitly.

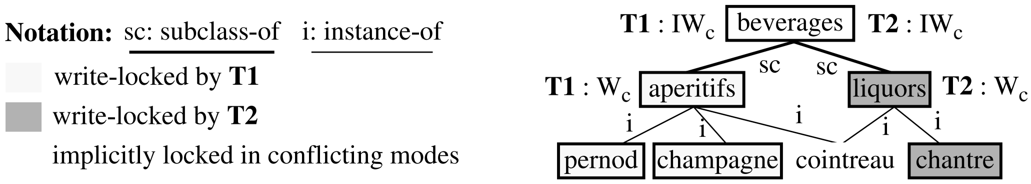


Figure 5: The problem with implicit locks in a graph structure.

In order to find out conflicts with implicitly locked objects, we may access all ancestors or descendants of an object. For this purpose, we use the bidirectional representation of links in KB graphs (c.f. Sect. 2.3). We could follow basically five approaches.

Lock All Referenced Objects

The first and most simple approach is to explicitly lock all referenced objects. In the example of Fig. 5, if either T1 or T2 locks all objects explicitly, the interference in *cointreau* is detected. This practically vanishes the semantics of implicit locks, but it solves the problem.[5] Nevertheless, this method leads to a large overhead, since many locks are required.

Search for Conflicts

The second approach is, before accessing any implicitly locked bastard, to climb up the structure in order to search for possible conflicts. In this case, a conflict is detected if such a bastard is already implicitly locked by any other ancestor in a conflicting mode. In the above example (Fig. 5), T1 needs to upward traverse the other path coming in *cointreau* in order to look for conflicts. In this particular case, it soon realizes a conflict in *liquors*. This alternative requires less locks to be held than the first one, because it does not consider explicit locks on all referenced objects and still makes use of implicit locks, but it also leads to some substantial drawbacks. Of course, it is very expensive if an object has several parents, which in turn have several parents, and so on. In such a case, a transaction needs to traverse very long paths in order to find out possible conflicts. After all, it may happen that there is no conflict at all.

5. This alternative is followed by ORION [30] for its class lattices (see Sect. 5.2).

Analysis of All Descendants

The third approach is, before setting any explicit lock on an object, to analyze all descendants of this object and to explicitly set locks on the bastards.[6] So, any conflict is immediately avoided, because the objects where potential conflicts may happen, are already explicitly locked. In the current example (Fig. 5), as soon as transaction T1 sets a W_c lock on *aperitifs*, it also needs to set the same lock on *cointreau*, the only bastard descendant of *aperitifs*. When following the same proceeding, T2 detects the conflict and must then wait until T1 terminates. In this alternative, the lock manager, always before granting an explicit lock, needs to downward traverse all paths affected by this explicit lock and to set an explicit lock on all bastard descendants.

Lazy Evaluation Strategy

The fourth approach is to add to the previous one a kind of *lazy evaluation* strategy for lock conflict resolution. In this approach, a transaction may request and be granted an explicit lock without further analysis. However, before effectively accessing an implicitly locked bastard, it must verify whether this object is already locked in a conflicting mode. 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 bastard indicating that it has been already accessed via another parent of it.

The main difference of this alternative to the previous one is that a transaction needs to explicitly lock only those bastard descendants which it actually accesses, leaving the others for the concurrent access by other transactions. In the current example (Fig. 5), the W_c lock on *aperitifs* by T1 is immediately granted. T1 can access *pernod* and *champagne* without problems, but if, and only if, it accesses *cointreau*, it then needs to set an explicit lock on this object. On the other side, T2 performs a similar proceeding, and it only needs to set an extra lock if it wants to access *cointreau*. In this case, if the lock by T1 is already released, for example because T1 has already committed, T2 can receive the lock, but if T1 still holds the lock, T2 must wait. In this approach, only the bastard descendants effectively accessed need to be explicitly locked. Those which are not accessed are not locked. Hence, implicitly locked bastards not touched via some parent may be accessed via another one. For these reasons, this is the best alternative to solve the problem with implicit locks in graph structures, and therefore we are going to follow it in LARS.

---

6. This alternative was pointed out by Garza and Kim [21] for the class lattices in ORION [30], implemented for test purposes, but discarded.

Semantic Optimizations

As a last point for discussion, we briefly mention a fifth approach, which represents an improvement in the previous one, by means of the addition of some semantic optimizations. For example, if we state that when all possible paths to an implicitly locked bastard are already explicitly locked by a transaction, this transaction does not need to set an explicit lock on this bastard when accessing it. In fact, all paths reaching this bastard should be already covered by this transaction with explicit locks on its parents, and therefore the potential conflicts would be already detected. This proceeding may be cheap in some special cases, but in general it is too difficult to be realized and too expensive.

4.6 The Locking Rules

Having presented the general guidelines of LARS, we are finally able to expose its locking rules (Table 4). Before explaining these rules, it is convenient to notice that: First, transactions are allowed to directly set locks in the root object in any mode. Second, LARS always produces *strict executions* [5], i.e., it requires the locks of a transaction to be released only at its termination (commit or abort). Third, as we have assumed in Sect. 2.2, the KB graph is (single) rooted and connected, and it automatically stays is this form as changes undergo over time. Thus, even in a constantly changing graph, there will always be at least one path from the object to the root. In turn, which specific path should be locked does not matter for LARS, it must be one, but anyone (we return to this point in a moment). (Of course, the abstraction relationship being used for locking must be considered.)

Table 4: Locking rules.

1	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.
2	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.
3	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.
4	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.
5	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 first rule states that an IR lock (from the C_type, S_type, or A_type) on a non-root object must be preceded by either IR or IW 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 second rule has a similar meaning, but for the IW locks, requiring that they must be preceded by IW or RIW locks on at least one path from that object to the root object. The third rule states, first of all, that an R lock on a non-root object must be covered by IR or IW locks on at least one path from this object to the root object. Thereafter, it requires that a transaction must explicitly lock the bastard descendants.[7] This is implemented by LARS' lazy evaluation strategy, thereby avoiding conflicts with implicitly locked objects. The fourth and fifth rules have a similar meaning, but for RIW and W locks, respectively.

We now provide a complete example (Fig. 6) using again our restaurant KB (Fig. 1). Suppose T1 wants to read the object *turtle-soup* as a part of the object *sunday-menu*. To do that it must follow rules 1 and 3 for requesting, respectively, IR_a locks on the parents of *turtle-soup*, and an R_a lock on it. On the other side, T2 wants to write the object *appetizers* together with its subclasses and instances. In turn, it must follow rules 2 and 5 for requesting IW_c locks on the ancestors of *appetizers* and a W_c lock on it, respectively. However, when trying to access the object *cold-dishes*, T2 realizes that this object is a bastard and, as stated by the rule 5, it requests a W_c lock on this object and is granted because this object was free. The same may happen for the object *turtle-soup* as long as T2 tries to access it. When trying this, either T2 must wait, if the R_a lock on this object is still held by T1, or it may be granted, if T1 has already terminated.

An important point of explicitly locking bastard descendants, besides guaranteeing serializability, is the slackness of the original requirement of MGL of covering all paths from the node to the root, and as a consequence all ancestors, with intentions before granting an exclusive lock [24]. This is a serious limitation when an object has several ancestors and is likely to be used via many of them. In such situations, it is very inefficient to set intention locks on all the parents [29] and as a consequence on all paths to the root. LARS limits the overhead of the whole process of setting write locks and still provides, to a limited extent, a minimization of the number of locks to be set by transactions, through the use of implicit locks.

7. There may be situations where a descendant may have two edges pointing to the same ancestor. 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.

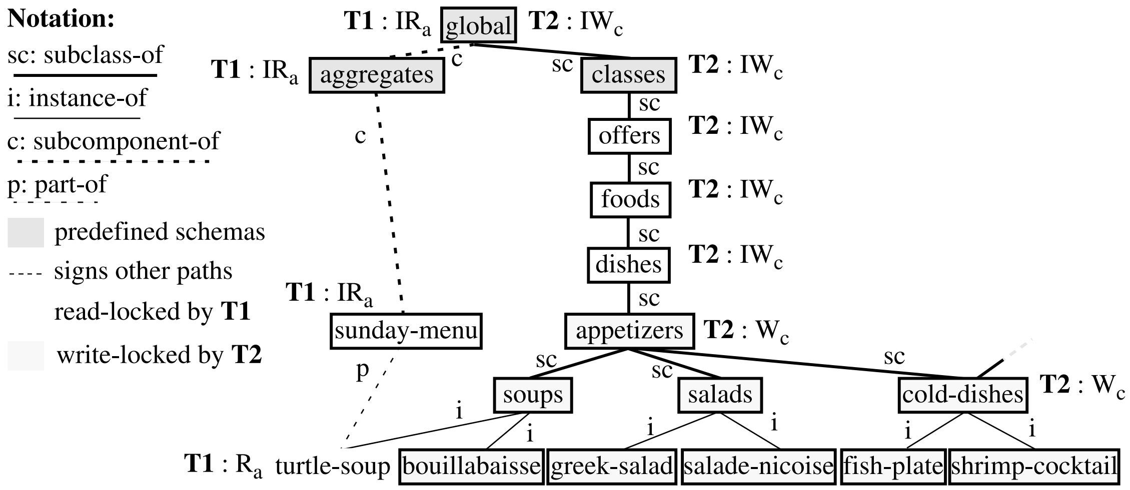


Figure 6: Avoiding conflicts with implicitly locked objects.

4.7 Coping with Insert and Delete Operations

Thus far, we have considered a KB as a fixed set of objects, which can be accessed by reads and writes. Most real KBs can dynamically grow and shrink. Therefore, in addition to reads and writes, we must support operations to insert new relationships and objects as well as to delete existing relationships and objects. Before passing on to the explanation of the rules, we need to make some considerations in the way these operations are performed. We have assumed (Sect. 2.2) that a KB is represented by a rooted and connected graph and, additionally, that when an object does not participate in the defined abstraction relationships, it is treated as being an instance of the predefined root *global*. Further, we have assumed that the abstraction relationships between the objects are represented in a bidirectional way (Sect. 2.3). All of that has some consequences in the way insert and delete operations should be performed. In the following, we discuss inserts and deletes in detail. In particular, these operations may be arbitrarily complex, and we are interested in finding out the primitive operations by means of which any other complex operation may be realized as a composition of those. The essence of our idea is: There are four operations - insert node, insert edge, delete node, and delete edge; node operations are always accompanied by one edge operation; to operate on a node, it must be locked, and to operate on an edge, its end points must be locked.

Inserting an Object

Since the KB graph is connected, the insertion of an object must be handled as an operation composed of two steps: The creation of the object itself and its connection to another existing object.[8] In turn, since two objects are involved in this operation, one could ask: Which is the object being inserted, the superior or the inferior object? The way LARS represents the KB graph (as a single-rooted graph) answers this question. It must be the inferior object, otherwise one would create another root in the graph when inserting an object as a superior. Hence, LARS considers the object being inserted as the inferior. Notice that this is not a restriction, but the establishment of a primitive case. If one states that an object O being inserted must be the superior, LARS can handle it as two operations. First, the insertion of O and its connection to a superior object (at least to the corresponding predefined object, and hence O is handled as an inferior), followed by the connection of O to the inferior object (coped with by the objects' connection rule).

Another important point in the insertion of an object gives respect to the roles of the superior object in the current KB state. We use the restaurant KB (Fig. 1) in order to explain this point. Suppose we are designing our KB and that we have not yet defined the parts of the object *mousse-au-chocolat*. Hence, *mousse-au-chocolat* currently is just an instance of *desserts* and therefore takes no part in the aggregation graph. When inserting any part of *mousse-au-chocolat*, one should acquire a lock of the A_type on it, since the aggregation concept is being applied. However, it is impossible to acquire an A_type lock on *mousse-au-chocolat*, because it is not yet in the aggregation graph, and therefore one cannot navigate from the predefined object *aggregates* to it. Nevertheless, since the aggregation graph is rooted at *aggregates*, this operation must be accompanied by the connection of *mousse-au-chocolat* to *aggregates* anyway. Hence, LARS treats such cases as first of all the connection of the superior object to the corresponding graph, followed by the insertion of the inferior object. In our example, LARS would connect *mousse-au-chocolat* to *aggregates* and thereafter insert any part of it. By this way, we have that the superior object is already connected to the corresponding graph when an inferior of it is being inserted. Particularly, we need this to synchronize the type of the locks to be requested in both objects.

At last, another important aspect is how many relationships (connections) are specified in the insertion of an object. For example, one can state that the object being inserted is an instance of a class and an element of a set (like *bordeaux* in our restaurant KB). In such a case, LARS decomposes such an operation and handles it as an insertion followed by as many connections as necessary (and so handled by the objects' connection rule). Hence, by the insertion of an object we are connecting it to a single superior object.

---

8. At least the predefined objects (*global*, *classes*, *sets*, and *aggregates*) will be present in the KB graph.

Finally, rule 6 in Table 5 presents the lock requests necessary to insert an object. It states that before inserting an object, its parent (the superior object) must be held in at least IW mode (and so recursively until the root object is reached). The type of such an IW is dictated by the abstraction relationship being inserted. Fig. 7 provides an example of the appliance of this rule. Suppose transaction T1 wants to insert the object *cote-de-provence* as an instance of *wines*. To accomplish this task, T1 must request an IW_c on *wines*, the parent of *cote-de-provence*. In turn, this IW_c must be covered by IW_c on the parents of *wines* until the root *global*. Just after holding those locks, T1 is then able to insert the object *cote-de-provence*. As soon as *cote-de-provence* is inserted, T1 is granted a W_c on this object and holds it until it terminates.

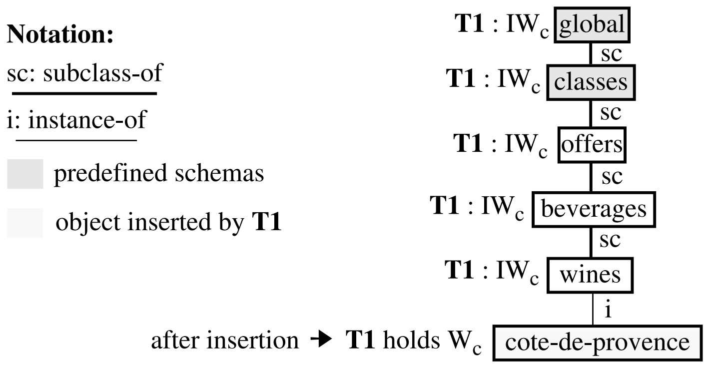


Figure 7: Locks for the insertion of an object.

Deleting an Object

We will profit from the above discussions about the insertion of an object and summarize our considerations about deleting an object. There may be several steps involved in this operation (the deletion of the object itself and several disconnections, depending on the current KB state). The primitive case comprehends the deletion of an inferior object and its disconnection from a superior object. Like above, the other more complex cases may be built upon this simple case, so that they may be composed of this primitive case and as many disconnections as necessary (thus handled by the objects' disconnection rule). Rule 7 in Table 5 deals with deletion of objects, similarly to insertions, with the extra requirement that the object itself (the inferior) must be held in W mode. Notice that such a W lock implies IW locks on a parent, on a parent of the parent, and so forth until the root is reached. Finally, the type of such W and IW locks is dictated by the abstraction relationship in question.

Connecting Objects

Like before, also here two objects are affected by this operation, namely a superior and an inferior object, and the current state of both objects with respect to other objects in the KB may be arbitrary. The main difference here is that the inferior object may be either a bastard or a purebred. Rule 8 in Table 5 copes with the connection of objects. It states that in order to connect objects, the inferior object must be held in *any* W mode and the superior object in at least IW mode (this one according to the abstraction relationship being applied). In Fig. 8, which complements the last example (Fig. 7), it becomes clear why any exclusive typed lock may be requested in this case. Suppose that T1 wants to connect the recently created object *cote-de-provence* as an element of *french-wines*. Following rule 8, T1 must request a W lock on this object, normally a W_s, since it is applying the association concept. However, this object takes no part in the association graph yet, what makes impossible the acquirement of a W_s on it (before the connection, there is no path from *sets* to it). Since *cote-de-provence* is an instance of *wines*, T1 requires a W_c on this object and is granted because it in fact already holds such a lock due to the proceedings of the last example (if this were not the case, it should cover a path to the root with IW_c locks). Thereafter, T1 must require an IW_s on *french-wines*, the new parent of it, and recursively on the ancestors. Finally, after holding all the required locks, T1 connects both objects. Therefore, in the particular case of connecting objects, a transaction is allowed to acquire a W lock of any type in the inferior object. In general, such a W lock will in fact be of the C_type, because normally an object first receives its structure by means of the inheritance mechanism of the classification concept, and thereafter it is connected to other objects using the association or aggregation concepts. As can be seen, the connection of objects is a bit more complicated operation, because the transaction does not know a priori which are the roles of both objects in the current KB state.

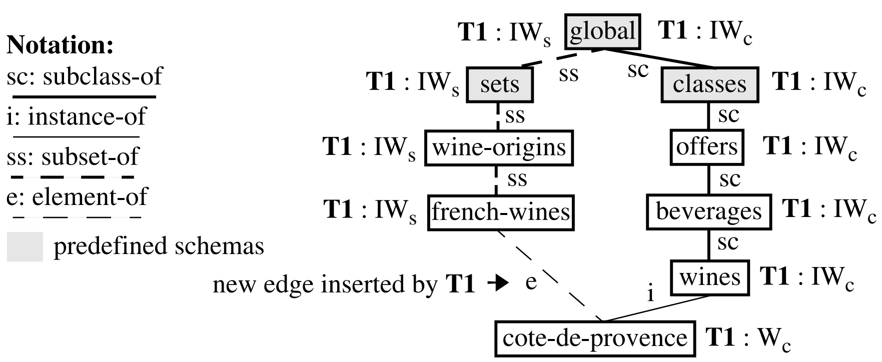


Figure 8: Locks for the insertion of an edge.

Disconnecting Objects

Profiting from all discussions so far, we shortly present the disconnection of objects. We shall only mention that we do not allow the disconnection of purebreds, because if we disconnect a purebred (deleting its only edge, then), we are either disconnecting the KB graph or creating a new root of it. Hence, in the disconnection of a purebred, the transaction must choose between either deleting the object (and thus handled by the objects' deletion rule), or connecting it firstly to another superior object (and hence handled by the objects' connection rule). Therefore, when disconnecting objects, the inferior object must always be a bastard object. Rule 9 in Table 5 presents the objects' disconnection rule. It is a simple case because the transaction does know the current roles of both objects, and by this way the path it must traverse for requesting locks. It must request a W lock on the inferior object, an IW lock on the superior, accordingly to the abstraction concept in question, and finally recursively cover a path to the root with IW locks.

Table 5: Locking rules for insert and delete operations.

6	Before inserting an object in the classification | association | aggregation graph, the requester must acquire an $IW_{c\|s\|a}$, $RIW_{c\|s\|a}$ or $W_{c\|s\|a}$ lock on the superior object. After the insertion, the requester is granted a $W_{c\|s\|a}$ lock on the object.
7	Before deleting an object from the classification | association | aggregation graph, the requester must acquire a $W_{c\|s\|a}$ lock on it and an $IW_{c\|s\|a}$, $RIW_{c\|s\|a}$ or $W_{c\|s\|a}$ lock on the superior object.
8	Before connecting objects using the classification | association | aggregation concept, the requester must acquire either a W_c or a W_s or a W_a lock on the inferior object and an $IW_{c\|s\|a}$, $RIW_{c\|s\|a}$ or $W_{c\|s\|a}$ lock on the superior object.
9	Before disconnecting objects using the classification | association | aggregation concept, the requester must acquire a $W_{c\|s\|a}$ lock on the inferior object and an $IW_{c\|s\|a}$, $RIW_{c\|s\|a}$ or $W_{c\|s\|a}$ lock on the superior object.

4.8 The Phantom Problem

Granular locks provide physical locks and being so we have problems with the so-called phantoms in LARS. The most reasonable solution we found to this problem is to delegate to the transactions the decision about tolerating or not phantoms. If a transaction decides to avoid phantoms at all, it must then request exclusive typed locks (i.e., either W_c or W_s or W_a) on the object in the next higher level of the graph it is currently working on (what is foreseen by the locking rules). Taking this measure accordingly, no phantoms may happen because other transactions are unable to access any inferior of such an object, or to create a new inferior, or to delete an existing inferior (all of that with respect to the working graph, of course). Hence, no phantom appears.

4.9 Correctness Concerns

Definition 1: A ***directed acyclic graph*** (DAG), *G*, is a finite set of nodes *N* and a set of arcs *A* (a subset of N × N). N represents all objects in the KB, whereas A all abstraction relationships between these objects.

Definition 2: The ***classification graph***, G_c, is a subgraph of G, containing the set of nodes N_c and the set of arcs A_c. N_c contains the nodes representing the (super-) classes and instances of N. In turn, A_c contains the arcs representing the generalization and classification abstraction relationships of A.

Definition 3: The ***association graph***, G_s, is a subgraph of G, containing the set of nodes N_s and the set of arcs A_s. N_s contains the nodes representing the (super-) sets and elements of N. In turn, A_c contains the arcs representing the set- and element-association abstraction relationships of A.

Definition 4: The ***aggregation graph***, G_a, is a subgraph of G, containing the set of nodes N_a and the set of arcs A_a. N_a contains the nodes representing the (super-) components and parts of N. In turn, A_a contains the arcs representing the component- and element-aggregation abstraction relationships of A.

Observation 1: $(G_c \cup G_s \cup G_a) = G$

Observation 2: $(N_c \cup N_s \cup N_a) = N$

Observation 3: $(A_c \cap A_s) = (A_c \cap A_a) = (A_s \cap A_a) = \emptyset$

Definition 5: A node *p* is a ***parent*** of node *c* and *c* is a ***child*** of node *p*, if $<p, c> \in$ A.

Definition 6: A node with no parents is a ***root***. There is always only one root in G, namely, the predefined node ***global***.

Definition 7: A node with no children is a ***leaf***.

Definition 8: A ***path*** is a set of arcs of A, $a_1 \dots a_n$, where $a_i = <b_i, b_{i+1}>$ and $b_i \in$ N.

Definition 9: Node *b* is an ***ancestor*** of node *c* if *b* = *c* or *b* lies in some path from the root to node *c*.

Definition 10: Node *b* is a ***superior*** of node *c* if *b* is an ancestor of *c* other than *c* itself.

Definition 11: Node *c* is a ***descendant*** of node *b* if *b* is not a leaf and either *c* = *b* or *c* lies in some path from *b* to some leaf.

Definition 12: Node *c* is an ***inferior*** of node *b* if *c* is a descendant of *b* other than *b* itself.

Definition 13: A ***bastard*** is any node which has more than one parent.

Definition 14: A ***purebred*** is any node which has only one parent.

Observation 4: The LARS protocol is a strict two-phase locking (strict 2PL) protocol. All transactions are well-formed and strict two-phase. Hence, as a strict 2PL protocol, for any arbitrary schedule of transactions running under the LARS protocol, the serializability graph is acyclic and thus it is serializable. (The reader is referred to [5] for correctness concerns about the serializability of 2PL protocols.)

Suppose all transactions obey the LARS protocol with respect to a given lock graph, G, that is a DAG.

Theorem 1: If a transaction owns an explicit or implicit lock on a node of G, then no other transaction owns a conflicting explicit or implicit lock on that node.

Proof: *We prove by induction on the length of the shortest path from the root to a node that the claim of the Theorem 1 is true.*

Base case: *The base case is trivial. The length **l** of the shortest path from the root **r** to a node **n** is equal zero (**l** = 0). In this case, **n** is the root **r** itself. As the root, **n** then has only outgoing vertices, and therefore no implicit locks on **n** are possible. Conflicting explicit locks on **n** are handled by the compatibility matrices. Hence, the claim holds for the base case.*

Induction step: *Suppose the Theorem 1 is true for all nodes **n** such that the length of the shortest path **p** from **r** to **n** is less than **l**.*

*Consider a particular node **n** such that the length of the shortest path from **r** to **n** is **l**. There are two different cases.*

Case 1: *If **n** is a bastard, then the claim follows easily, it must have been explicitly locked by using any of the locking rules 3-5, and conflicting explicit locks are coped with by the compatibility matrices.*

Case 2: *If **n** is a purebred, it may have been either explicitly or implicitly locked. Thus, there are two subcases.*

Case 2.1: *If **n** is an explicitly locked purebred, then the claim holds due to the same arguments above, i.e., any of the locking rules 3-5 was applied, and conflicting explicit locks are handled by the compatibility matrices.*

Case 2.2: *If **n** is an implicitly locked purebred, then there must be a superior of it in **p** explicitly locked. Considering each of them in turn, in conjunction with the induction hypothesis, one falls back on Case 1 or Case 2.1 or latest on the base case, and therefore the claim holds.* ❑

Theorem 2: If a transaction inserts a node in G, then no other transaction may lock that node until the inserting transaction commits.

Proof: *We prove by using rule 6 that the claim of the Theorem 2 is true. As stated by rule 6, a transaction is granted an exclusive lock on the object being inserted. In addition, since an object is inserted in only one of G_c, G_s, or G_a, such an exclusive lock is incompatible with any other lock of the same type, and hence any other lock request on that node will be refused, obeying the compatibility matrix for locks of the same type (Table 2). Since LARS is a strict two-phase locking protocol, this lock will be released only at transaction's termination. Hence, the claim holds.* ❑

Theorem 3: If a transaction owns an explicit or implicit lock on a node of G, then no other transaction may delete that node.

Proof: *We prove by using rule 7 that the claim of the Theorem 3 is true. As stated by rule 7, a transaction must acquire an exclusive lock on the object being deleted. In turn, such an exclusive lock must be covered by intention exclusive (or higher) locks on the superior objects. Since an object is deleted from one of G_c, G_s, or G_a, we have that: First, such an exclusive lock on the object itself is incompatible with any other explicit lock of the same type on the object, and second, such intention exclusive locks on the superiors of the object are incompatible with any other lock which could lock the object implicitly, accordingly to the compatibility matrix for locks of the same type (Table 2). Hence, a transaction may not delete an object which is explicitly of implicitly locked by another transaction, and therefore the claim holds.* ❑

Theorem 4: If a transaction owns an explicit or implicit C_type | S_type | A_type lock on a node of $G_{c|s|a}$, then no other transaction may connect that node to another superior node in $G_{c|s|a}$.

Proof: *We prove by using rule 8 that the claim of the Theorem 4 is true. As stated by rule 8, a transaction must acquire an exclusive lock on the object before connecting it to another superior object. In turn, such an exclusive lock must be covered by intention exclusive (or higher) locks on the superior objects. Since an object is connected to one of G_c, G_s, or G_a, we have that: First, such an exclusive lock on the object itself is incompatible with any other explicit lock of the same type on the object, and second, such intention exclusive locks on the superiors of the object are incompatible with any other lock which could lock the object implicitly, accordingly to the compatibility matrix for locks of the same type (Table 2). Hence, a transaction may not connect an object which is explicitly of implicitly locked by another transaction, and therefore the claim holds.* ❑

Theorem 5: If a transaction owns an explicit or implicit C_type | S_type | A_type lock on a node of $G_{c|s|a}$, then no other transaction may disconnect that node from another superior node in $G_{c|s|a}$.

Proof: *We prove by using rule 9 that the claim of the Theorem 5 is true. As stated by rule 9, a transaction must acquire an exclusive lock on the object before discon-*

necting it from another superior object. In turn, such an exclusive lock must be covered by intention exclusive (or higher) locks on the superior objects. Since an object is disconnected from one of G_c*,* G_s*, or* G_a*, we have that: First, such an exclusive lock on the object itself is incompatible with any other explicit lock of the same type on the object, and second, such intention exclusive locks on the superiors of the object are incompatible with any other lock which could lock the object implicitly, accordingly to the compatibility matrix for locks of the same type (Table 2). Hence, a transaction may not disconnect an object which is explicitly of implicitly locked by another transaction, and therefore the claim holds.* ❑

4.10 Final Considerations

Including Indices in the KB Graph

Thus far, we have pretended that all accesses against a KB take place through the abstraction relationships. However, a more detailed examination of KBMSs suggests otherwise. Hash tables, trees, sorted and unsorted lists, arrays, access sequences, etc., are normally used to speed up the access to objects in KBs. There are several special purpose CC algorithms for indices (e.g., [14, 46, 47, 44]), most of which uses some form of non-two-phase locking. In general, any of them may be used for controlling the accesses via indices in KBs. LARS, which is a strict two-phase locking protocol, would probably not be useful to index locking as compared to existing techniques.

Deadlocks

LARS is subject to deadlocks. In addition, the protocol for handling bastards introduces deadlocks that did not occur in conventional hierarchical locking. The difference is that LARS' rules 3-5 require also explicit locks on the implicitly locked bastards, and when acquiring those locks, transactions may get deadlocked.

Lock Conversion

Lock conversions are normally used to increase (upgrade) the access mode a transaction has to an object [24]. In LARS, lock conversions are handled accordingly to the type of lock. In [56] we discuss how upgrade operations shall be done in our transaction model, which allows also for controlled downgrading of locks.

Lock Escalation

Lock escalation [25, 5] is also taken into consideration by LARS. Nevertheless, a lock escalation in LARS alleviates the transaction from requesting locks just on purebreds, but not on bastards - they must still be explicitly locked on access.

In: *Data & Knowledge Engineering* **22** (1997) 233-259.

Implementation and Performance Considerations

The LARS protocol is not directly comparable to MGL. As a matter of fact, LARS takes advantage of the hierarchical structuring of different locking granules like MGL does, but the accessibility of objects via multiple relationships (paths) alters its behavior substantially. LARS was fully implemented in the KBMS KRISYS, following a model of nested transactions [56]. The details of its implementation can be found in [38].

An important point to the performance of LARS is the frequency of bastards in the KB graph. If bastards are very, very common (e.g., over 90% of the objects), then LARS' performance would probably be comparable to 2PL, where every touched object must be locked (10% of the objects would not be explicitly locked, but LARS has the extra costs of setting intention locks). Whether or not bastards are common in KBs will certainly depend on the richness of the modeling and the requirements (features) of the applications. We have applied single-user KRISYS to run quite a number of prototype applications during the last few years [15]. For the evaluation of the bastard problem, we have analyzed five different KBs which were developed in the last years in our university (namely, an architecture KB, a restaurant KB, a mechanical engineering KB, a medical KB, and a real estate appraiser KB). Since they were developed for academic purposes and for single-user environments, the number of objects was about 500 in each KB. Not specially the absolute number, but importantly the degree of bastard occurrences may be indicative for larger KBs. In three of them, the percentage of bastards was in average 2% (more specifically, 1.5%, 1.8%, and 2.6%). In one of them, it reached 30% and in the other 50%. Of course, the fewer the bastards, the better is the performance of LARS. At the moment, LARS is undergoing more detailed investigations on its performance [39].

5. Related Work

5.1 Concurrency Control for KBMSs

As far as we know, the only other work addressing transaction synchronization in KBMSs is Chaudhri's Dynamic Directed Graph (DDG) policy presented in [11, 10, 12]. It is an extension of the locking protocol for hierarchical DB systems of Silberschatz and Kedem [57]. Whereas the former is able to cope with cycles and updates in the underlying structure, this is not considered by the latter. The main distinction between LARS and DDG is that they address different problems. When transactions access a large number of objects there are two potential problems. The first problem is that the large number of locks held by a transaction can mean high locking overhead which can be potentially reduced by locking several objects at once (i.e., by using coarse granules of locking). The second problem, which is a consequence of using two-phase locking, is that the locks may be held for a long period of time, thus limiting the concurrency. DDG attempts to address the second problem and does not say anything about the first. In turn, LARS addresses the first problem and does not deal with the second.

Nevertheless, the DDG policy makes no difference between different abstraction relationships, i.e., it does not treat, for example, neither a class and its instances, nor an aggregate and its components, etc., as a single lockable unit. Hence, the semantics of the KB graph is not exploited to improve the concurrency. Further, no kind of implicit locks is defined. This may jeopardize the overall performance of DDG and, in addition, lead the lock system to run out of storage. Finally, phantoms are not taken into consideration. A more detailed critical analysis of this protocol may be found in [54].

5.2 Concurrency Control for OODBMSs

Due to the lack of work on transaction synchronization in KBMSs, we have analyzed some CC protocols of a related area, namely OODBMSs [53]. There are some CC methods for OODBMSs that have been designed independent of any specific system [1, 26]. Due to space limitations, we provide here just a brief analysis of CC protocols in some specific OODBMSs.

ORION

ORION [4, 31, 34] supports locks on three different types of hierarchy, namely the so-called granularity hierarchy for logical entities, the class lattice hierarchy, and the composite objects (aggregates) hierarchy. ORION extended MGL and partially provides implicit locks [21, 33, 32]. The main problem of ORION is that it does not allow even a read on a class to be performed in parallel with a write on an instance of it. ORION provides implicit locks for the instances of a class, but not for the subclasses of a class. ORION prohibits an object of being instance of many classes at the same time. This eliminates the bastard problem at the instance level, but would be hardly applicable to KBMSs. In turn, classes may have several direct superclasses, and hence the bastard problem appears for classes. It is solved with the requirement that for a query involving a class and its descendants as well as for a schema change operation on a class, a lock must be set not only on the class, but also on each of its subclasses. Consequently, all subclasses of a class must be explicitly locked in such cases.

O_2

The CC technique actually implemented in the O_2 [3, 16] is a conventional one used in DBMSs, but there is an interesting approach for CC in O_2 presented by Cart and Ferrié [8] based on a classification of methods. Here we discuss the proposal of [8]. According to [8], in O_2 methods are classified according to whether they are performed on a class or on an instance, and as a reading or a writing method. In addition to this classification of methods, O_2 also distinguishes the type of access a transaction requires for an object. There are the so-called *real* and *virtual accesses* [8]. The main benefit of this classification is that reading (but not writing) a class is compatible with either reading or writing any of its instances. Implicit locks on instances of a class are provided. However, no kind of implicit locks is available for subclasses of a class. The bastard problem is handled by O_2 in a similar manner as by ORION, with the inclusion of the distinction between real and virtual accesses which may be also used for detecting conflicts.

GemStone

GemStone [41, 51, 6, 7] protects its concurrent transactions using a combination of optimistic and pessimistic CC techniques. Particularly, the choice of whether to use the optimistic or pessimistic technique depends on the degree of contention of an object. Using an optimistic access, the objects do not need to be locked, being controlled by a shadowing mechanism. Instead, at commit time, existing conflicts are detected. Finally, locks are used to control the pessimistic accesses. Optimistic methods may show very poor performance due to, among other things, the possibly high percentage of transactions that must be aborted when, at commit time, conflicts are detected [27, 50, 45]. In turn, the pessimistic method of GemStone does not provide implicit locks. GemStone's limited number of lock types restricts the parallelism, and it is unaware about the semantics of the relationships between objects.

ObjectStore

The CC mechanism of ObjectStore [36] is similar to those used in conventional DBMSs. It provides 2PL with a read/write lock for each page, i.e., the locking granularity is on a per-page basis. Every time a user needs to access an object, the corresponding page is transferred to the workstation and locked in the server in either exclusive or shared mode [36]. Thus, ObjectStore does not show any improvement with respect to CC.

6. Conclusions

KBMSs are a growing research area finding applicability in many different domains. The higher its demand, the greater the necessity for knowledge sharing. As a matter of fact, the research for CC techniques tailored to the KBMS environment plays a crucial role in this context. Moreover, it assumes a paramount importance as the demand for ever-larger KBs grows.

Following this research direction, we have presented the LARS approach for transaction synchronization in KBMSs. The most important feature of LARS is the partition of the KB graph into several logical ones, hence allowing transactions to concurrently access such partitions through different points of view. Thereafter, LARS applies granular locks to each partition, providing thus many different lock types and taking the necessary precautions with respect to the dynamism of the KB graph. In this manner, LARS captures more of the semantics contained in the KB graph, in the sense that it does not consider descendants of an object as being simply descendants of it, but, on the contrary, descendants with special characteristics and significance, which are based on the abstraction relationships. By such a means, LARS can exploit the inherent parallelism in a knowledge representation approach. Further, LARS offers different locking granules and considers implicit locks, alleviating the task of managing too many locks due to the high number of objects in real world applications. Finally, LARS copes well with multiple abstraction relationships to objects, by requiring explicit locks on bastards. This, in turn, relaxes the necessity of covering all paths to the root with intentions, reducing it to only one path.

Abstraction concepts are defined for KBMSs' use in general. We have designed LARS to work with the three abstraction concepts that should be provided by KBMSs, and we have referred to an existing KBMS, namely KRISYS [43], as a powerful example to make our discussions specific. Nevertheless, LARS is flexible enough to be used by other object-oriented data models. In the case of a specific data model not supporting all the three abstractions, one should only cut off the corresponding lock modes of LARS and handle bastards in the same way. Of course, the data model must be powerful enough to support bidirectional links in the representation of edges in KB graphs, because LARS uses them to easily find bastard objects. Representing the edges bidirectionally in KB graphs exclusively for the right functioning of LARS would probably not pay off.

Acknowledgments - We would like to thank J. Reinert for the helpful discussions on the correctness concerns of LARS, J. Thomas for supporting us understanding the upper layers of KBMSs, and the anonymous referees for suggestions to improve the presentation.

In: *Data & Knowledge Engineering* **22** (1997) 233-259.

References

[1] D. Agrawal and A. El Abbadi. A non-restrictive concurrency control protocol for object-oriented databases. In: *Proc. of the 3rd Int. Conf. on Extending Database Technology*, Vienna, Austria, 469-482 (1992).

[2] B.R. Badrinath and K. Ramamrithan. Semantics-based concurrency control: Beyond commutativity. In: *Proc. of the 3rd Int. Conf. on Data Engineering*, Los Angeles, USA, 304-311 (1987).

[3] F. Bancilhon, C. Delobel and P. Kanellakis (eds.). *Building an object-oriented database system: The story of O_2*. Morgan Kaufmann, USA (1992).

[4] J. Banerjee, H.-T. Chou, J.F. Garza, W. Kim, D. Woelk and N. Ballou. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems* **5** (1), 3-26 (1987).

[5] P.A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, USA (1987).

[6] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Willians and M. Willians. The GemStone data management system. In: W. Kim and F.H. Lochovsky (eds.), *Object-oriented concepts, databases, and applications*, ACM Press, USA, 283-308 (1989).

[7] P. Butterworth, A. Otis and J. Stein. The GemStone object database management system. *Communications of the ACM* **34** (10), 64-77 (1991).

[8] M. Cart and J. Ferrié. Integrating concurrency control into an object-oriented database system. In: [3], 463-485.

[9] M.A. Casanova and P.A. Bernstein. General purpose schedulers for database systems. *Acta Informatica* **4** (1980).

[10] V.K. Chaudhri. *Transaction synchronization in knowledge bases: Concepts, realization and quantitative evaluation*. Ph.D. Thesis, University of Toronto, Toronto, Canada (1994).

[11] V.K. Chaudhri, V. Hadzilacos and J. Mylopoulos. Concurrency control for knowledge bases. In: *Proc. of the 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, Cambridge, USA (1992).

[12] V.K. Chaudhri, V. Hadzilacos, J. Mylopoulos and K.C. Sevcik. Quantitative evaluation of a transaction facility for a KBMS. In: *Proc. of the 3rd Int. Conf. on Information and Knowledge Management*, Gaithersburg, USA (1994).

[13] P.K. Chrysanthis, S. Raghuram and K. Ramamritham. Extracting concurrency from objects: A methodology. In: *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Denver, USA, 108-117 (1991).

[14] P. Dadam, V. Lum, U. Prädel and G. Schlageter. Selective deferred index maintenance and concurrency control in integrated information systems. In: *Proc. of the 11th Int. Conf. on Very Large Data Bases*, Sweden (1985).

[15] S. Dessloch, F.-J. Leick, N.M. Mattos and J. Thomas. The KRISYS project: a summary of what we have learned so far. In: *Proc. of the BTW'93*, Braunschweig, Germany, March 1993.

[16] O. Deux et al. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering* **2** (1), 91-108 (1990).

[17] K.P. Eswaran, J.N. Gray, R.A. Lorie and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM* **19** (11), 624-633 (1976).

[18] A.A. Farrag and M.T. Ozsu. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems* **14** (4), 503-525 (1989).

[19] J.M. Fischer, N.D. Griffeth and N.A. Lynch. Global states of a distributed system. *IEEE Transactions on Software Engineering* **SE-8** (3), 198-202 (1982).

[20] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems* **8** (2), 186-213 (1983).

[21] J.F. Garza and W. Kim. Transaction management in an object-oriented database system. In: *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Chicago, USA, 37-45 (1988).

[22] J.N. Gray. Notes on database operating systems. In: *Operating systems: An advanced course*, Springer, Berlin (1978).

[23] J.N. Gray. The transaction concept: Virtues and limitations. In: *Proc. of the 7th Int. Conf. on Very Large Data Bases*, Cannes, France, 144-154 (1981).

[24] J.N. Gray, R.A. Lorie, G.R. Putzolu and I.L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In: *Proc. of the IFIP Working Conf. on Modeling in DBMSs*, Freudenstadt, Germany, 365-394 (1976).

[25] J.N. Gray and A. Reuter. *Transaction processing: Concepts and techniques*. Morgan Kaufmann, USA (1993).

[26] T. Hadzilacos and V. Hadzilacos. Transaction synchronization in object bases. *Journal of Computer and Systems Sciences* **43** (1), 2-24 (1991).

[27] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems* **9** (2), 111-120 (1984).

[28] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys* **15** (4), 287-317 (1983).

[29] U. Herrmann, P. Dadam, K.M. Küspert, E.A. Roman and G. Schlageter. *A lock technique for disjoint and non-disjoint objects*. Technical Report TR.89.01.003, IBM Heidelberg Research Center, Heidelberg, Germany (1989).

[30] W. Kim. *Introduction to object-oriented databases*. MIT Press, USA (1990).

[31] W. Kim, N. Ballou, H.-T. Chou, J.F. Garza and D. Woelk. Features of the ORION object-oriented database system. In: W. Kim and F. Lochovsky (eds.), *Object-oriented concepts, databases, and applications*, ACM Press, USA, 251-282 (1989).

[32] W. Kim, J. Banerjee, H.-T. Chou, J.F. Garza and D. Woelk. Composite objects support in an object-oriented database system. In: *Proc. of the 2nd OOPSLA*, Orlando, USA (1987).

[33] W. Kim, E. Bertino and J.F. Garza. Composite objects revisited. In: *Proc. of the ACM SIGMOD Int. Conf. on the Management of Data*, Portland, USA, 337-347 (1989).

[34] W. Kim, J.F. Garza, N. Ballou and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering* **2** (1), 109-124 (1990).

[35] H.T. Kung and C.H. Papadimitriou. An optimality theory of concurrency control for databases. In: *Proc. of the ACM SIGMOD Int. Conf. on the Management of Data*, Boston, USA, 116-126 (1979).

[36] C. Lamb, G. Landis, J. Orenstein and D. Weinreb. The ObjectStore database system. *Communications of the ACM* **34** (10), 50-63 (1991).

[37] H.J. Levesque and R.J. Brachman. A fundamental tradeoff in knowledge representation and reasoning. In: R.J. Brachman and H.J. Levesque (eds.), *Readings in knowledge representation*, Morgan Kaufmann, USA (1985).

[38] J. Lutze. *Lock Management in the KBMS KRISYS - An Implementation of the LARS Protocol for Nested Transactions* (in German). Undergraduation Project Work, Univ. of Kaiserslautern, Kaiserslautern, Germany (1996).

[39] J. Lutze: *Benchmarking the Architectural Components of a Multi-User Knowledge Base Management Systems* (in German). Undergraduation Diploma Work, Univ. of Kaiserslautern, Kaiserslautern, Germany (in preparation).

[40] N. Lynch. Multilevel atomicity: A new correctness criterion for database concurrency control. *ACM Transaction on Database Systems* **8** (4), 484-502 (1983).

[41] D. Maier, J. Stein, A. Otis and A. Purdy. Development of an object-oriented DBMS. In: *Proc. of the OOPSLA*, Portland, USA, 472-482 (1986).

[42] N.M. Mattos. Abstraction concepts: The basis for data and knowledge modeling. In: *Proc. of the 7th Int. Conf. on Entity-Relationship Approach*, Rom, Italy, 331-350 (1988).

[43] N.M. Mattos. *An approach to knowledge base management*. Lecture Notes in Artificial Intelligence 513, Springer, Germany (1991).

[44] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indices. In: *Proc. of the 16th Int. Conf. on Very Large Data Bases*, Australia, 392-405 (1990).

[45] C. Mohan. Less optimism about optimistic concurrency control. In: *Proc. of the 2nd Int. Workshop on RIDE: Transaction and Query Processing*, Tempe (1992).

[46] C. Mohan, D. Haderle, Y. Wang and J. Cheng. *Single table access using multiple indices: Optimization, execution and concurrency control techniques*. IBM ARC Research Report RJ7341 68822, Almaden, USA (1989).

[47] C. Mohan and F. Levine. *ARIES/IM: An efficient and high concurrency index management method using write-ahead logging*. IBM ARC Research Report RJ6846 65380, Almaden, USA (1989).

[48] P. Muth, T.C. Rakow, G. Weikum, P. Brössler and C. Hasse. Semantic concurrency control in object-oriented database systems. In: *Proc. of the 9th Int. Conf. on Data Engineering*, Vienna, Austria, 233-242 (1993).

[49] J. Mylopoulos and M. Brodie. Knowledge bases and databases: Current trends and future directions. In: *Proc. of the Workshop on Artificial Intelligence and Databases*, Ulm, Germany (1990).

[50] P. Peinl and A. Reuter. Empirical comparison of database concurrency control schemes. In: *Proc. of the 9th Int. Conf. on Very Large Data Bases*, Florence, Italy, 97-108 (1983).

[51] D.J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In: *Proc. of the OOPSLA*, Orlando, USA, 111-117 (1987).

[52] T.C. Rakow, J. Gu and E.J. Neuhold. Serializability in object-oriented database systems. In: *Proc. of the 6th Int. Conf. on Data Engineering*, Los Angeles, USA, 112-120 (1990).

[53] F.F. Rezende. *Evaluating the suitability of OODBMS concurrency control techniques to the KBMS environment*. Internal Report, Univ. of Kaiserslautern, Kaiserslautern, Germany (1995). (submitted for publication).

[54] F.F. Rezende. Concurrency control techniques and the KBMS environment: A critical analysis. *RITA - Journal for Theoretical and Applied Computer Science* **2** (1), Brazil, 37-76 (1995).

[55] F.F. Rezende and T. Härder. A lock method for KBMSs using abstraction relationships' semantics. In: *Proc. of the 3rd Int. Conf. on Information and Knowledge Management*, Gaithersburg, USA, 112-121 (1994).

[56] F.F. Rezende and T. Härder. Concurrency control in nested transactions with enhanced lock modes for KBMSs. In: *Proc. of the 6th Int. Conf. on Database and Expert Systems Applications*, London, UK (1995).

[57] A. Silberschatz and Z. Kedem. Consistency in hierarchical database systems. *Journal of the ACM* **27** (1), 72-80 (1980).

[58] P.M. Schwarz and A.Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems* **2** (3), 223-250 (1984).

[59] W.E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers* **37** (12), 1488-1505 (1988).