In: Fong, J., Siu, B. (Eds.), Multimedia, Knowledge-Based & Object-Oriented Databases, Springer Verlag, Singapore, 1996. pp. 126-148.

Multiple Granularity Locks for the KBMS Environment

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. *As ever-larger knowledge bases (KBs) are being built, knowledge sharing becomes an aspect of paramount importance in Knowledge Base Management Systems (KBMSs). In this paper, we propose a way of controlling knowledge sharing by means of the LARS (**L**ocks using **A**bstraction **R**elationships' **S**emantics) approach for concurrency control in KBs. LARS synchronizes transactions through many different granules of locking, which are based on the semantics of the abstraction relationships commonly used in knowledge representation approaches. LARS supports a higher degree of potential concurrency in that it maintains different logical partitions of a KB graph, a means for representing KBs, and offers many lock types to be used on the basis of each one of the partitions. By such a way, LARS captures more of the semantics contained in a KB, through an interpretation of the (abstraction) relationships between objects, profits from such semantics for synchronizing the transactions, and thus makes feasible the exploitation of the inherent parallelism in a knowledge representation approach.*

Keywords: *concurrency control, knowledge bases, object-oriented databases.*

1 Introduction

We begin this paper with the question: What exactly is a knowledge base? There is a lot of discussion on the meaning of *knowledge* or a *knowledge base*. Similarly, a variety of definitions of a knowledge base may be found in the literature. We present here a specific definition, widely accepted in the knowledge representation community, by Levesque and Brachman [LEV85]:

> *"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 knowledge base as a system with explicit structures representing the knowledge, and that 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. We will see in a later section how our protocol visualizes such systems (as single-rooted, directed, and acyclic graphs). In other words, our approach for concurrency control (CC) is general and applicable to a broad class of applications (e.g., object-oriented), and not only to knowledge-based systems.

In this paper, we present our approach for CC in KBs, an important research direction as ever-larger KBs are being built and the applicability of KBMSs grows [MYL90]. The main objective we have in mind is the provision of serializability for ACID transactions. With serializability we mean that our technique is governed by the *Serializability Theory* of Gray et al. [GRA76], which states that if an execution produces the same output and has the same effect on the database (DB) as some serial execution of the same transactions, it is correct, because serial executions are assumed to be correct. With ACID transactions we mean that the transactions running in our system have the properties of conventional ones, the ACID (atomicity, consistency, isolation, and durability) properties pointed out by Härder and Reuter [HÄR83]. In other words, our protocol neither treats the operations' semantics in order to allow non-serializable schedules to be produced, nor copes it with long-duration transactions (in fact, the transactions may span minutes and even hours, but are not anticipated in terms of days or months).

Among the most important classes of CC algorithms are *locking*, *timestamps*, and *serialization graphs* [BER87]. There is also a great body of variations of these classes based on multiple versions, multi-level, optimistic methods, etc. 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 [CHA92, MYL94]. Consequently, we have chosen to develop our CC technique for KBs based on locking. In particular, we consider multiple granules of locking for controlling the concurrent accesses in a KB. Granular locks are known to be meaningful, because they provide transactions the possibility of choosing, among different locking granules, the most appropriate one to accomplish their tasks. In addition, the use of implicit locks significantly minimizes the number of locks to be set by transactions. These are some of the reasons which lead us to hope to be able to use the power and elegance of granular locks also in KBs.

An initial version of the LARS protocol has been published in [REZ94]. In this paper, we present the evolution of our investigations on CC in KBs since [REZ94], which are here condensed in this much more robust version of LARS. Differently from [REZ94], here we have cut two lock modes (the conventional lock modes in the terminology of that paper), we analyze the representation of edges in KB graphs, we make some comments on inference engines, we analyze the problems of considering the semantics of methods for CC purposes, we expose in more details the problems of granular locks in a semantically rich structure like KB graphs, and we discuss in much more details the issues of insert and delete operations. This paper is organized as follows. After providing some particular KBMSs' issues (Sect. 2), we present an overview of granular locks 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 KBMSs' Issues

2.1 Abstraction Relationships

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 [MAT88]. 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 to begin a detailed discussion about the abstraction concepts, the reader is referred to [MAT88, MAT89] for more details on this topic. In order to illustrate one such a KB graph, in Fig. 1 we provide an example of a KB to support architects in the design of houses. Notice that the purpose of this scenario is merely to illustrate our solution for knowledge sharing, rather than schema design issues. This application scenario will serve as a running example for the rest of the paper.

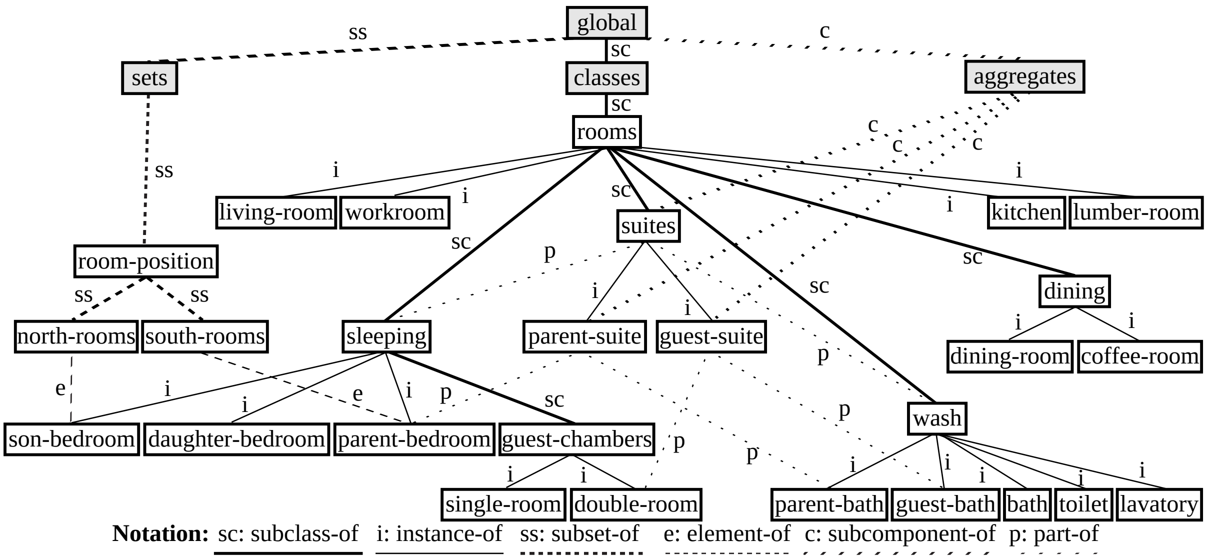

Fig. 1. An architecture knowledge base.

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 LARS. 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].

2.2 The Representation of Edges in KB Graphs

A point deserving a bit discussion is the representation of the edges in KB graphs. 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 a particular system. Representing the edges by unidirectional links has the clear 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. To be more precise, we provide some examples (using the KB of Fig. 1) of such queries using a concrete knowledge base language, namely KOALA (KRISYS Object-Abstraction LAnguage) [DES90, MAT89]. Let us analyze the following KOALA query:

```
(ASK  (?x      inherited-slots
               (supers guest-chambers dining))
      (is-instance   ?x      rooms *))
```

This query retrieves all (*) the instances of the class *rooms* and shows their attributes inherited from the classes *guest-chambers* and *dining*. The first part of this query is a projection clause, which limits the results' presentation just to the attributes inherited (inherited-slots) from the classes *guest-chambers* and *dining*. The second part is a selection clause, which transitively retrieves the instances (?x) of the class *rooms*. In order to evaluate this query, first of all KOALA instantiates the class *rooms* (using the top-down links), retrieving all its instances. Secondly, KOALA goes from the instantiated objects to the classes *guest-chambers* and *dining* (using the bottom-up links), learns their structures, and by this way performs the projection. This is a common query, which may be evaluated very efficiently if bidirectional links are present. In the following, we provide a query now using both the classification and aggregation graphs:

```
(ASK  (?x ?y)
      (and  (is-instance ?x rooms *)
            (is-instance ?y suites *)
            (is-aggregation ?x ?y *)))
```

In simple words, this query retrieves all (*) the instances of *rooms* which take part in any *suites*. To evaluate this query, KOALA retrieves the instances of *rooms* (?x), then the instances of *suites* (?y), and thereafter it goes from the instances of *rooms* to the instances of *suites* via the aggregation relationships (bottom-up). This is also a very common query which may be evaluated efficiently by means of bidirectional links. In fact, whenever one wants to know which instances of a class take part in some aggregate, one uses the bottom-up links for finding the aggregates. Similar arguments may be used to justify bidirectional links in the association graph (for example, to efficiently answer simple queries like: Which instances of a class take part in some set?).

Therefore, bidirectional links may not be considered performance 'bottlenecks' in KBMSs, because the costs for maintaining them up-to-date are paid off when evaluating the queries much more efficiently. Smalltalk represents the edges for the abstraction concepts it provides by means of bidirectional links [GOL89]. C++ standard is being expanded to support bidirectional links, and in the next standard they are going to be available [HAR94, HÜS94]. KRISYS also supports bidirectional links [MAT89]. These are just some examples. Due to that, we assume that the edges in a KB graph are bidirectionally represented. Like the KBMSs' query evaluation components, we use

1. This representation and behavior are very similar to the ones used by KRISYS [MAT89] to represent KBs.

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.

2.3 Inference

In Artificial Intelligence, *inference* refers to the various processes by means of which programs (the so-called *inference engines*) drawn conclusions or generate new knowledge from facts and suppositions. There are several alternatives to knowledge representation aimed at performing inference, such as production rules, (full) first-order logic, Horn clauses, etc. In this section, we briefly discuss this topic and analyze the access and further lock requirements imposed by inference engines.

Production rules represent general knowledge about the problem domain, and can be used to represent a variety of different types of inference. Each rule is a simple program with the format [DAV77]:

```
(IF     <condition>
        THEN        <action>)
```

where the condition is typically a conjunction of predicates, and the action activates other procedures which potentially change the KB state. A very important aspect of the inference process is the kind of control or guidance that is available to direct the process to the desired conclusion [HEU87]. Mechanisms used to identify applicable rules are denoted inference strategies [PUP86]. Among these, data-driven (or forward reasoning) and goal-driven (or backward reasoning) are the most common. In turn, the mechanisms used to determine the rule to be executed are called conflict resolution strategies (usually further divided into search strategies and conflict solving strategies).

Additionally to rules, first-order logic can also be used to represent procedural knowledge for the application domain. However, a major problem with general first-order logic for knowledge representation is the difficulty in expressing control structures that efficiently guide the use of a large KB [VAS85]. In order to reduce such problems, at the costs of also reducing the representation power, practical tools (e.g. Prolog) do not use full first-order logic, but only a subset known as definite (Horn) clauses. Horn clauses are interpreted in a procedural way reminiscent of the backward reasoning of production rules, leading to efficient search processes [FRO86].

All in all, the backward or forward reasoning mechanisms are processed following an object(or record)-oriented approach, and the paths that they use are the commonly available ones (indices, hash tables, abstraction graphs, etc.). Anyway, the search and/or (subsequently) update processes triggered by either production rules or Horn clauses may be mapped to the usual operators (like selection, projection, join, semi-join, etc.) supported by most data manipulation and knowledge languages. Therefore, the process of locking the objects that need to be accessed whenever the inference engine activates such production rules and/or Horn clauses does not generate any new problem to a CC mechanism operating in KBMSs where inference is supported.

2.4 Methods

Methods are used in KBMSs in order to describe the operational aspects of objects' attributes, i.e., they characterize the behavior of the real world entity. 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 when synchronizing transactions. For example, a set object may allow simultaneous inserts, because the semantics of a set implies that the order in which concurrent inserts are performed does not matter, i.e., it makes no difference in the results of these operations, even though inserts would be classified as writes in a read-write scheme and therefore could not occur concurrently. The main idea behind the use of operations' semantics for transaction synchronization 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.

Such an approach may be meaningful, since in some applications users may be satisfied with a schedule that preserves consistency, even though it is not serializable. It is well-known that allowing the transaction processing mechanism to run these schedules may result in higher parallelism and better performance in some cases [GAR83, CAS80, ESW76, KUN79, FIS82, GRA81, LYN83]. However, there is one drawback, among the main important ones, of such an approach [GAR83]: It is difficult for a general transaction processing mechanism to decide what schedules are semantically consistent. Although Badrinath and Ramamrithan [BAD88, BAD87] in their approach hold the opinion that the designer of an object type needs only to specify the semantics of operations, and their compatibilities ought to be (dynamically) determined from these specifications when the operations are requested to be executed on an object, it is more widely accepted that such a transaction processing mechanism must receive some help from the users of the system. The user (or objects designer) must provide a table indicating which operations conflict (cannot be executed concurrently). Hence, every time an object is created, besides describing the operations (or in the KB terminology, the methods) to be available for manipulating the object, the user must define a compatibility matrix for those operations (generally based on the commutativity of the operations). Such a ‘commutativity matrix’ is then used by the scheduler to synchronize the transactions executing those operations on the object.

In LARS, we do not exploit the semantics of methods to allow for non-serializable schedules due to several reasons. In the following, we discuss some of these reasons:

- One important question arising with such a methodology gives respect to schema evolution (or extensibility). That is, KBMSs and object-oriented database management systems (OODBMSs) allow the objects’ definitions to be dynamically modified (concurrently) with the accesses to the objects defined by them. For example, a class can be modified by replacing an old method implementation by a new and perhaps more efficient one. Methods or even attributes may be added or deleted from a class. All of that has some impact on the use of methods’ semantics for CC. Unfortunately, it is not yet so clear in such methodologies in how far schema evolution will be affected. It may become very hard to users, if every time one changes or inserts or deletes a method, one must analyze the semantics of all other methods of the object, in order to adjust the object’s commutativity matrix. This is a drawback which may seriously confuse the extensibility property of such systems. In summary, the cost of serializable schedules may be unacceptably high in some applications, but it is really not a good idea to burden users with the classification and analysis of methods in order to define their commutativities.
- Another point worth of discussion is recovery. If commutative operations are allowed to be simultaneously performed on an object, a conventional (e.g., page-oriented) recovery mechanism does not work correctly. In such cases, logical logging must be performed and compensating transactions (or countersteps) must be provided. In turn, it could be very difficult in some cases to analyze transactions in order to write countersteps for their operations. The proper notion of commutativity depends upon the implementation and representation of the objects and all of that has impact on recovery. In particular, Weihl [WEI89] has shown that when operations on an object are executed by using the *update-in-place* policy [HÄR83] for recovery then the conflict (commutativity) relation must be derived from what he called *right backward commutativity*. On the other hand, if operations are executed with the *deferred-update* approach then the conflict relation is derived from *forward commutativity*. In general, the more semantics one uses in synchronization to improve concurrency, the more complex become the synchronization and, as a consequence, the recovery mechanisms.
- Most of the approaches for commutativity-based CC assume that each object provides operations that can be called by transactions to examine and modify the object’s state, and additionally that these operations constitute the sole means by which transactions can access the state of the objects (among others, examples are [WEI88, BAD88, FAR89, GAR83, SCH84, RAK90, CHR91, HAD91]). However, we believe that in a real-life environment, some transactions may want to bypass the encapsulation of an object and rather access the object directly through a generic knowledge or data manipulation language. Therefore, the scheduler should in fact deal with the coexistence of methods and generic transactions that access objects directly. This may even more complicate the synchronization of transactions in such methodologies. Muth et al. [MUT93] present conceivable reasons for bypassing the encapsulation of an object and propose a protocol addressing this issue.

Finally, the most critical question is whether performance will really improve or not with the use of operations’ semantics for transaction synchronization. Of course, in special applications performance may be definitively improved. However, there are applications where semantically consistent schedules may not be acceptable, because certain *anomalies* or *race conditions* [CAS80] may occur. As stated by Garcia-Molina [GAR83], it may be impossible to obtain the results of a semantically consistent schedule with any serializable schedule, and this may be undesirable to some users (refer to [GAR83] for some examples). In general, it is not yet known how practical these methodologies will be. Since the use of operations' semantics did not convince us so far, we designed the LARS protocol where methods compete for locks like any other transaction request. Hence, LARS copes with methods like ordinary read and write operations.

3 The Multigranularity Locking Protocol

Granular locks were first introduced by Gray et al. in [GRA76] by means of the *Multigranularity Locking Protocol* (MGL, for short). The basic idea of MGL comes from the choice of different lockable units, which are locked by the system to ensure consistency and to provide isolation. When choosing the lockable units for implementing this protocol, one will be always faced with the dichotomy: Concurrency versus overhead. On one hand, concurrency is increased by a fine lockable unit (e.g., a record or a field). Such a unit is appropriate for small transactions which access few units [GRA76]. On the other hand, a fine locking granule is costly for complex transactions which access a large number of granules. Such transactions would have to acquire and maintain a large number of locks [GRA93], which implies a larger overhead. Thus, a coarse locking granule (e.g., a file) would be more convenient for such transactions. However, a coarse granule discriminates against transactions which only want to lock a fine granule of the file [GRA76]. The main benefit of MGL is that it satisfies both of these situations, allowing lockable units of different granularities to coexist in the same system. Moreover, MGL created the notion of *implicit locks*, stating that by putting a lock on a granule, all descendants of it become implicitly locked without the necessity of setting further locks. Lastly, MGL 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. Those locks are used to sign the intention of a transaction to set locks at a finer granularity. Thus, MGL has a basic set of locks composed of the IS (Intention Share), IX (Intention eXclusive), S (Share), SIX (Share Intention eXclusive), and X (eXclusive) modes, which are then applied to the nodes in a lock graph (a hierarchy or a DAG) (Fig. 2) [GRA76].

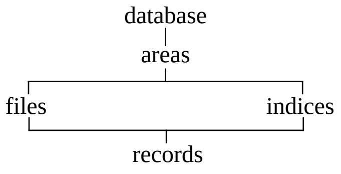

Fig. 2. A lock graph for granular locks.

Notwithstanding, a protocol like 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 parents of the node must be covered with IX (or greater) locks [GRA78, GRA76]. Consequently, all superiors of the node cannot be held by other transactions in incompatible modes. In the DAG of Fig. 2, to lock a particular *record* in X mode, one should lock in advance the corresponding *file*, *index*, *area* and *database* in IX mode. One question arises: How may transactions know which are the superiors of a node, in order to cover them with IX locks before requesting an X lock on it? This is possible because the data model, to which MGL may be applied, provides a strict separation 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*.

However, in 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 nowhere to find statically in KBMSs, as in usual DB catalogs in DB systems (DBSs). 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. As a consequence, the serializability of transactions may become violated.

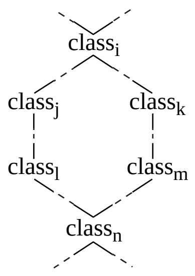

Fig. 3. A classification DAG.

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 unprofitable use of the semantically rich structure represented by KB graphs. If we would directly apply MGL to KB graphs, we would not at all be able to interpret their edges, which represent the abstraction relationships between objects (nodes). 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. A transaction changing an object's class properties puts an X lock on it and IX on its superiors, changes the object itself and all its subclasses and instances (due to the inheritance mechanism). 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 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.

4.2 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 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. 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$_{c\vert s\vert a}$	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$_{c\vert s\vert a}$ or IR$_{c\vert s\vert a}$ mode, and direct **instances | elements | parts** in R$_{c\vert s\vert a}$ mode.
IW$_{c\vert s\vert a}$	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$_{c\vert s\vert a}$, RIW$_{c\vert s\vert a}$, R$_{c\vert s\vert a}$, IW$_{c\vert s\vert a}$ or IR$_{c\vert s\vert a}$ mode, and direct **instances | elements | parts** in W$_{c\vert s\vert a}$ or R$_{c\vert s\vert a}$ mode.
R$_{c\vert s\vert a}$	gives shared access to the requested object and implicitly to all direct and indirect **sub-classes | subsets | subcomponents** and **instances | elements | parts** of this object.
RIW$_{c\vert s\vert a}$	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$_{c\vert s\vert a}$, RIW$_{c\vert s\vert a}$, R$_{c\vert s\vert a}$ or IW$_{c\vert s\vert a}$ mode, and direct **instances | elements | parts** in W$_{c\vert s\vert a}$ or R$_{c\vert s\vert a}$ mode).
W$_{c\vert s\vert a}$	gives exclusive access to the requested object and implicitly to all direct and indirect **sub-classes | subsets | subcomponents** and **instances | elements | parts** of this object.

4.3 The Lock Compatibilities

With respect to the compatibility of the above mentioned lock types, 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 MGL known from the literature [GRA76, GRA78] (Table 2).

Table 2. 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 [REZ94], a detailed discussion on this topic may be found. Here we limit to presenting the compatibility matrix (Table 3) 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. In Table 3, the boxes marked with darker shadows are where LARS 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 typed locks of distinct types.

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

4.4 Accessing Implicitly Locked Objects

Before passing on to the LARS' locking rules, there is a last important point to be coped with by LARS which deserves a little bit discussion. As a matter of fact, multiple abstraction relations involving an object in a KB may lead to problems with the implicit locks, so that the isolation property of transactions [HÄR83] may be seriously 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 [REZ94]. 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 [REZ94], we have discussed 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. In 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 idea there 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. This strategy may be better comprehended in the next section's example (Fig. 4).

4.5 The Locking Rules

Having presented the general guidelines of LARS, we are finally able to expose its complete rules to be followed by transactions when requesting locks on objects in a KB (see Table 4). 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* [BER87], i.e., it requires the locks of a transaction to be released only at its termination (either commit or abort).

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[2]. 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.

Table 4. Locking rules.

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

We now provide an example (Fig. 4) using the architecture KB (Fig. 1). Suppose a transaction, say T1, wants to read the object *son-bedroom* as an element of the object *north-rooms*. To do that it must follow rules 1 and 3 for requesting, respectively, IR_s locks on the parents of *son-bedroom*, and an R_s lock on it. On the other side, another transaction, say T2, wants to write the object *sleeping* together with its subclasses and instances. In turn, it must follow rules 2 and 5 for requesting IW_c locks on the ancestors of *sleeping* and a W_c lock on it, respectively. However, when trying to access the objects *parent-bedroom* and *double-room*, T2 realizes that these objects are bastards, and, as stated by rule 5, it requests W_c locks on them, and is granted because they were free. The same may happen for the object *son-bedroom* as long as T2 tries to access it. When trying this, either T2 must wait, if the R_s lock on this object is still held by T1, or it may be granted, if T1 has already terminated.

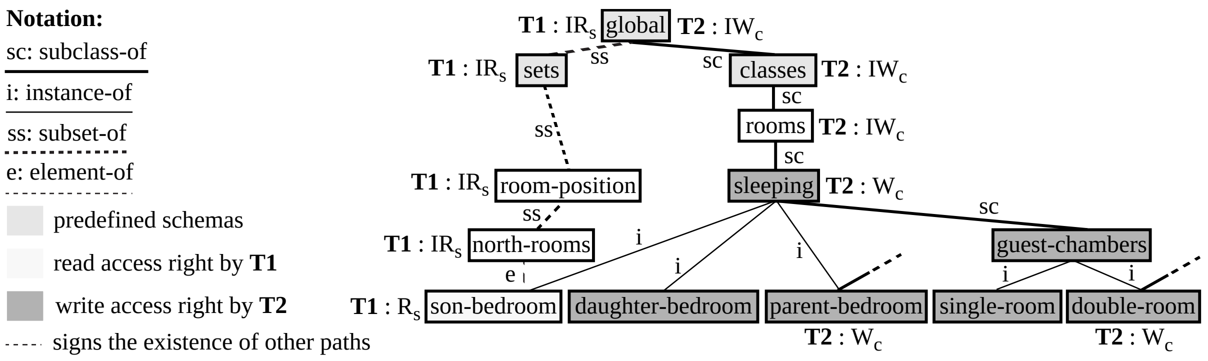

Fig. 4. Avoiding conflicts with implicitly locked objects.

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 [GRA76]. 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 [HER89], and as a consequence on all paths to the root, because too many locks are required, and a transaction may end up locking a large portion of the KB for a possibly simple operation, decreasing seriously the concurrency and increasing the overhead. Therefore, LARS significantly 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.

2. 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.

4.6 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. 2.1) 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. 2.2). All of that has some consequences in the way each one of these 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[3]. 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 architecture 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 *parent-suite*. Hence, *parent-suite* currently is just an instance of *suites*, and therefore takes no part in the aggregation graph. When inserting any part of *parent-suite*, 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 *parent-suite*, 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 *parent-suite* 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 *parent-suite* 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 *son-bedroom* in our architecture 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.

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

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

reached). The type of such an IW mode is dictated by the abstraction relationship being inserted. Fig. 5 provides an example of the appliance of this rule. Suppose transaction T1 wants to insert the object *maid-bedroom* as an instance of *sleeping*. To accomplish this task, T1 must request an IW_c lock on *sleeping*, the parent of *maid-bedroom*. In turn, this IW_c lock must be covered by IW_c locks on the parents of *sleeping* until the root *global*. Just after holding those locks, T1 is then able to insert the object *maid-bedroom*. As soon as *maid-bedroom* is inserted, T1 is granted a W_c lock on this object, and holds it until it terminates.

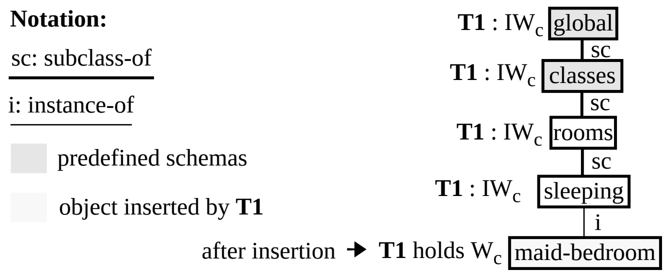

Fig. 5. Locks for the insertion of an object.

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. 6, which complements the last example (Fig. 5), it becomes clear why any exclusive typed lock may be requested in this case, independently of the type. Suppose that T1 wants to connect the recently created object *maid-bedroom* as an element of *north-rooms*. Following rule 8, T1 must request a W lock on this object, normally a W_s lock, since it is applying the association concept. However, this object is not an element of any other object yet, what makes impossible the acquirement of a W_s lock on it (before the connection, there is no path from *sets* to it). Since *maid-bedroom* is an instance of *sleeping*, T1 requires a W_c lock 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 locks). Thereafter, T1 must require an IW_s lock on *north-rooms*, the new parent of it. In turn, this intention lock requires intention on the parents, recursively. Finally, after holding all the required locks, T1 connects the 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 (W_c), because normally an object first receives its structure (type) 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.

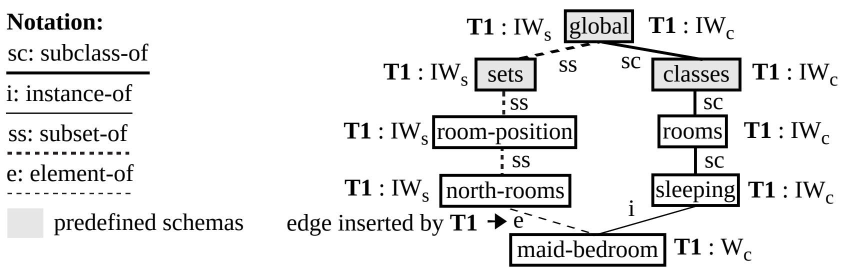

Fig. 6. Locks for the connection of objects.

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 purebred objects, 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 object's 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 thus 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.

4.7 The Phantom Problem

As stated by Bernstein et al. [BER87], the *phantom problem* is the CC problem for dynamic DBs. Granular locks provide physical locks, and being so we have problems with the so-called phantoms in LARS. Phantoms are characterized by inserted or deleted objects which may seem to appear or disappear to some concurrent transactions like a ghost. The phantom problem was first introduced by Eswaran et al. in [ESW76], which also proposed the *predicate locks* for elegantly coping with such situations. Since predicates do not seem applicable in an efficient way in the given KB structures, we must deal with phantoms in some other manner.

The most reasonable solution we found 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). Exemplifying, if a transaction wants to read all instances of some class, it must request an R_c lock on the class (and **not** an IR_c lock on it and start locking its instances in R_c mode). Thereafter, any other transaction may neither create a new instance of this class, nor delete an existing one, due to the R_c lock hold by the reading transaction on it, and hence no phantom appears.

5 Related Work

As far as we know, the only other work addressing transaction synchronization in KBMSs is Chaudhri's Dynamic Directed Graph (DDG) policy presented in [CHA92, CHA94, CHA94a]. It is an extension of the locking protocol for hierarchical DBSs of Silberschatz and Kedem [SIL80]. 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 problem. Nevertheless, among the main drawbacks of this protocol, we can cite: First, no difference is made 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. Second, no kind of implicit locks is defined. This may jeopardize the overall performance of this protocol, and, in addition, lead the lock system to run out of storage. Third, phantoms are not taken into consideration. A more detailed critical analysis of this protocol may be found in [REZ95].

Due to the lack of work on transaction synchronization in KBMSs, we have analyzed the suitability of some CC protocols from OODBMSs to the KBMS environment. The results of our analysis are reported in details in [REZ95a]. Due to space limitations, we are not going to discuss them here.

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. In the near future, KBMSs will be applied more and more in real world applications. As a matter of fact, the research for CC techniques tailored to the KBMS environment plays a crucial role to this applicability. 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 CC in KBs. The most important characteristic of LARS is the partition of the KB graph into many logical ones, allowing by this way transactions to concurrently access such partitions through different points of view. Thereafter, to each one of these partitions, we have applied granular locks, 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 a 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 of generalization, classification, association, and aggregation. This is the most important point of LARS, by means of which it can obtain a high degree of concurrency, exploiting the inherent parallelism in a knowledge representation approach.

In summary, the most important characteristics of LARS are: First, LARS offers different granules of locks and considers implicit locks, alleviating the task of managing too many locks due to the high number of objects in real world applications. Second, LARS copes well with multiple abstraction relationships to objects, by means of the requirement of explicitly locking bastards, which, in turn, relaxes the necessity of covering all paths to the root with intentions, reducing it to only one path. Third, LARS interprets the relationships between objects with respect to their semantics, providing typed locks for all abstraction concepts.

We have designed LARS to work with the three abstraction concepts that should be provided by KBMSs. However, 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 bastards. Representing the edges bidirectionally in KB graphs exclusively for the right functioning of LARS would probably not pay off. Finally, due to space limitations, we have not discussed issues like deadlocks, lock conversion, lock escalation, correctness concerns, and implementation aspects, although we have already addressed all of them. In particular, we have implemented LARS using as a running example the KBMS KRISYS.

References

[BAD87] Badrinath, B.R., and Ramamrithan, K. Semantics-based concurrency control: Beyond commutativity. In: *Proc. of the 3rd Int. Conf. on Data Engineering*, Los Angeles, USA, 304-311 (1987).

[BAD88] Badrinath, B.R., and Ramamrithan, K. Synchronizing transactions on objects. *IEEE Transactions on Computers* **37** (5), 541-547 (1988).

[BER87] Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency control and recovery in database systems*. Addison-Wesley, USA (1987).

[CAS80] Casanova, M.A., and Bernstein, P.A. General purpose schedulers for database systems. *Acta Informatica* **4** (1980).

[CHA92] Chaudhri, V.K., Hadzilacos, V., and Mylopoulos, J. Concurrency control for knowledge bases. In: *Proc. of the 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, Cambridge, USA (1992).

[CHA94] Chaudhri, V.K. *Transaction synchronization in knowledge bases: Concepts, realization and quantitative evaluation*. Ph.D. Thesis, University of Toronto, Canada (1994).

[CHA94a] Chaudhri, V.K., Hadzilacos, V., Mylopoulos, J., and Sevcik, K.C. Quantitative evaluation of a transaction facility for a KBMS. In: *Proc. of the 3rd Int. Conf. on Information and Knowledge Management*, Gaithersburg, USA (1994).

[CHR91] Chrysanthis, P.K., Raghuram, S., and Ramamritham, K. Extracting concurrency from objects: A methodology. In: *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Denver, USA, 108-117 (1991).

[DAV77] Davis, R., Buchanan, B.G., and Shortliffe, E.H. Production rules as a representation for a knowledge-based consultation program. *Artificial Intelligence* **8** (1), 15-45 (1977).

[DES90] Dessloch, S., Leick, F.-J., and Mattos, N.M. *A state-oriented approach to the specification of rules and queries in KBMS*. ZRI Report 4/90, University of Kaiserslautern, Germany (1990).

[ESW76] Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. The notions of consistency and predicate locks in a database system. *Communications of the ACM* **19** (11), 624-633 (1976).

[FAR89] Farrag, A.A., and Özsu, M.T. Using semantic knowledge of transactions to increase concurrency. *ACM TODS* **14** (4), 503-525 (1989).

[FIS82] Fischer, J.M., Griffeth, N.D., and Lynch, N.A. Global states of a distributed system. *IEEE Transactions on Software Engineering* **SE-8** (3), 198-202 (1982).

[FRO86] Frost, R.A. *Introduction to knowledge base systems*. Collins, U.K. (1986).

[GAR83] Garcia-Molina, H. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS* **8** (2), 186-213 (1983).

[GOL89] Goldberg, A., and Robson, D. *Smalltalk-80 - The language*. Addison-Wesley, USA (1989).

[GRA76] Gray, J.N., Lorie, R.A., Putzolu, G.R., and Traiger, I.L. 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).

[GRA78] Gray, J.N. Notes on database operating systems. In: *Operating systems: An advanced course*, Springer (1978).

[GRA81] Gray, J.N. The transaction concept: Virtues and limitations. In: *Proc. of the 7th VLDB*, France, 144-154 (1981).

[GRA93] Gray, J.N., and Reuter, A. *Transaction processing: Concepts and techniques*. Morgan Kaufmann, USA (1993).

[HAD91] Hadzilacos, T., and Hadzilacos, V. Transaction synchronization in object bases. *Journal of Computer and Systems Sciences* **43** (1), 2-24 (1991).

[HÄR83] Härder, T., and Reuter, A. Principles of transaction-oriented DB recovery. *ACM Computing Surveys* **15** (4), (1983).

[HAR94] Hartinger, R. Syntax puzzle: The newest C++ extensions (in German). *c't* **9**, 184-190 (1994).

[HER89] Herrmann, U., Dadam, P., Küspert, K.M., Roman, E.A., and Schlageter, G. *A lock technique for disjoint and non-disjoint objects*. Technical Report TR.89.01.003, IBM Heidelberg Research Center, Heidelberg, Germany (1989).

[HEU87] Heuschen, L. Reasoning. In: Shapiro, S.C. (Ed.), *Encyclopedia of Artificial Intelligence*, John Wiley & Sons, New York, USA, 822-827 (1987).

[HÜS94] Hüskes, R. Patience play: The forthcoming C++ standard (in German). *c't* **9**, 180-182 (1994).

[KUN79] Kung, H.T., and Papadimitriou, C.H. 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).

[LEV85] Levesque, H.J., and Brachman, R.J. 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).

[LYN83] Lynch, N. Multilevel atomicity: A new correctness criterion for DB concurrency control. *ACM TODS* **8** (4), 484-502 (1983).

[MAT88] 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, 331-350 (1988).

[MAT89] Mattos, N.M. *An approach to knowledge base management - Requirements, knowledge representation, and design issues*. Doctor Thesis, University of Kaiserslautern, Germany (1989).

[MUT93] Muth, P., Rakow, T.C., Weikum, G., Brössler, P., and Hasse, C. Semantic concurrency control in object-oriented database systems. In: *Proc. of the 9th Int. Conf. on Data Engineering*, Vienna, Austria, 233-242 (1993).

[MYL90] Mylopoulos, J., and Brodie, M. Knowledge bases and databases: Current trends and future directions. In: *Proc. of the Workshop on Artificial Intelligence and Databases*, Ulm, Germany (1990).

[MYL94] Mylopoulos, J., Chaudhri, V.K., Plexousakis, D., Shrufi, A., and Topaloglou, T. *Building knowledge base management systems: A progress report*. Technical Report DKBS-TR-94-4, University of Toronto, Canada (1994).

[PUP86] Puppe, F. Expert systems (in German). *Informatik Spektrum* **9** (1), 1-13 (1986).

[RAK90] Rakow, T.C., Gu, J., and Neuhold, E.J. Serializability in object-oriented database systems. In: *Proc. of the 6th Int. Conf. on Data Engineering*, Los Angeles, USA, 112-120 (1990).

[REZ94] Rezende, F.F., and 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, 112-121 (1994).

[REZ95] Rezende, F.F. Concurrency control techniques and the KBMS environment: A critical analysis. *RITA - Journal for Theoretical and Applied Computer Science* **2** (1), Brazil, 37-76 (1995).

[REZ95a] Rezende, F.F. *Evaluating the suitability of OODBMS concurrency control techniques to the KBMS environment*. Internal Report, University of Kaiserslautern, Germany (1995). (submitted for publication).

[SCH84] Schwarz, P.M., and Spector, A.Z. Synchronizing shared abstract types. *ACM TOCS* **2** (3), 223-250 (1984).

[SIL80] Silberschatz, A., and Kedem, Z. Consistency in hierarchical database systems. *Journal of the ACM* **27** (1), (1980).

[VAS85] Vassiliou, Y., Clifford, J., and Jarke, M. Database access requirements of knowledge-based systems. In: Kim, W., Reiner, D.S., and Batory, D.S. (Eds.), *Query Processing in Database Systems*, Springer-Verlag, 156-170 (1985).

[WEI88] Weihl, W.E. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers* **37** (12), 1488-1505 (1988).

[WEI89] Weihl, W.E. The impact of recovery on concurrency control. In: *Proc. of the 8th ACM PODS*, 259-269 (1989).