in: ZRI-Bericht 6/94, University Kaiserslautern, 1994.

Capturing Abstraction Relationships' Semantics for Concurrency Control 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

November 1994

Abstract

Knowledge Base Management Systems (KBMSs) are a growing research area finding applicability in several different domains. On behalf of this increasing applicability, the demand for ever-larger knowledge bases (KBs) is growing more and more. Inside this context, knowledge sharing turns out to be a crucial point to be supported by KBMSs. In this paper, we propose a way of controlling knowledge sharing, i.e., we present a concurrency control (CC) technique tailored for the KBMS environment. We show how we obtain serializability of transactions providing many different locking granules, which are based on the semantics of the abstraction relationships. We discuss the main challenges to be coped with by a CC technique for KBs and expose our solution to them. The main benefit of our technique is the higher degree of potential concurrency, which may be obtained by means of a logical partitioning of the KB graph and the provision of many lock types to be used on the basis of each one of the partitions. By this way, we capture more of the semantics contained in a KB graph, through an interpretation of its edges grounded in the abstraction relationships, and make feasible a full exploitation of all inherent parallelism in a knowledge representation approach.

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

1. Introduction

KBMSs are a new product generation which is finding ever more applicability in many different areas, like medicine, geology, engineering design, robotics, etc. As expected due to a growing applicability, the use of KBMSs is becoming more and more widespread and, accordingly, the demand for ever-larger KBs higher and higher. The main challenge of the research in the direction of KBMSs nowadays is to try the successful adaptation of such systems to real-life production environments [MB90]. However, the complete success of those systems in the market depends, among other things, of their potential for applicability. For instance, it would be very inefficient to obligate users of such systems to access valuable resources and information in mutual exclusion. Moreover, it would be neither viable (due to economical reasons) nor desirable (due to restricted accesses) to have some KB being accessed by just one user at a time. On the contrary, KBMSs should receive queries and updates in an interleaved fashion and control their concurrent execution against some KB. Consequently, multiple transactions should be able to run at the same time for better performance of such systems. Finally, it is exactly in this point that CC techniques for KBMSs play a crucial role, because they are among the most important means for allowing large, multi-user KBs to be widespread.

The main goals

As a matter of fact, arbitrary concurrent accesses to a resource can lead to many inconsistencies in the stored and retrieved information. All of that is because they can interfere with each other due to the interleaving of operations. This interleaving can cause programs to operate incorrectly, even if they are free from errors and no component of the system fails. The main objective of any CC technique is the coordination of actions that operate in parallel, access shared data, and doing so potentially interfere with each other. In the context of Database Systems (DBSs), CC has been studied extensively by the database (DB) community[1], and there is a vast amount of literature in this area. Unfortunately, CC has not received the attention of the Artificial Intelligence (AI) community.

In this paper, we present our approach for CC in KBMSs. 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. [GLPT76], which states that if an execution produces the same output and has the same effect on the DB as some serial execution of the same set of transactions, it is correct, because serial executions are assumed to be also 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 [HR83]. In other words, our protocol neither treats the semantic knowledge of transactions in order to allow non-serializable executions to be produced, nor copes it with long-duration transactions (in fact, the transactions may span minutes and even hours, but are not in terms of days or months).

Approaches for concurrency control

Among the most important classes of CC algorithms are *locking*, *timestamps*, and *serialization graphs* [BHG87]. There are 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 [CHM92, Ch93]. Consequently, we have chosen to develop our CC technique for KBs based on locking.

With respect to locking, we could consider several alternatives to our development. At a first sight, *predicate locks* [EGLT76] seem to be the best solution and the most desirable one. Among the reasons for that we could mention that predicates offer very appropriate locking granules for each particular transaction. In addition, we could try to use the predicates of knowledge languages in order to build the predicates for

1. [BHG87] presents a well-summarized consolidation of such studies.

locking. At last, with the appliance of predicates we could also lock non-existing objects, in other words, we would elegantly avoid phantoms. Nevertheless, predicate locking has some substantial drawbacks which lead to a lack of applicability. For example, it is not computationally efficient to check overlap between predicates. In addition, to maintain a predicate lock table is also very costly. Mohan [Mo90] advocates that comparing a new predicate against a predicate lock table of some reasonable size (e.g., containing 100 expressions) is just prohibitive. Gray and Reuter [GR93] state that predicate satisfiability is known to be NP-complete (the best algorithms for it run in time proportional to 2^N, where N = number of predicates). Therefore, although being very desirable, it seems to be impossible to determine a reasonable solution for the appliance of predicate locks in KBMSs.

Following another possible alternative, we could consider the most simple solution where every referenced object is explicitly locked on demand. The *two-phase locking* (2PL) *protocol* of Eswaran et al. [EGLT76] represents a well-known realization of such a strategy. This alternative is significant, however, for structures where the objects have the same meaning, like for example records in a file. Unfortunately, this approach does not make any assumption about the structure of the underlying data, and for structures like KB graphs, where nodes have different meanings and edges represent different abstraction relationships, it proffers a too simple solution. It would cause transactions to require too many locks, what would raise a great and unnecessary overhead for the lock manager. In addition, it does not offer the possibility of locking non-existing objects, and therefore phantoms are not at all avoided.

As another alternative, we could consider the KB graph as a structure with different granules of locks, and apply explicit locks on its nodes. In addition, we could state that a lock on a node would implicitly comprise locks on its descendants, what would minimize the number of locks to be acquired and managed. However, to the correct behavior of such implicit locks, we would need to mark the ascendants with some tag indicating that objects are being locked at a lower level. Such a sign would represent an intention to set locks at a finer granularity and should prevent implicit or explicit conflicting locks on the ancestors. In other words, we could consider the *granular locks protocol* (GLP, for short) of Gray et al. [GLPT76] 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. This protocol (and some extensions of it) has been a very popular approach in Object-Oriented Database Management Systems (OODBMSs) (among others, examples are ORION [Ki90] and O_2 [BDK92]). 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 the KBMS environment.

This paper is organized as follows. After providing some particular CC issues in KBMSs (Sect. 2), we discuss some important challenges of a CC technique for KBs, where we criticize related works (Sect. 3). Thereafter, we introduce our protocol for allowing and above all controlling knowledge sharing (Sect. 4). After the exposition of our proposal, we finally conclude the paper (Sect. 5).

2. Particular Concurrency Control Issues in KBMSs

2.1 The Abstraction Concepts

In describing the real world, people organize their knowledge in such a form which normally embodies some abstractions. These abstractions permit the suppression of specific details, while emphasizing those pertinent to the information to be described. Thus, abstractions turned out to be fundamental tools for knowledge organization. More stringently, Mattos [Ma88] advocates that the abstraction concepts are the most important constructs to be supported by any data or knowledge representation model. Such abstractions are expressed as relationships between objects, and have as main purpose the organization of such objects in some form. Particularly, one can find two main kinds of composition principals in the real world.

The first involves simple objects in order to build a composite object, creating an one-level hierarchy. The second involves composite objects in order to build a more complex composite object. This, in turn, may be applied recursively (n times), creating so an n-level hierarchy. In the following, we provide a brief description of the abstraction concepts of classification, generalization, association, and aggregation. In order to illustrate these concepts, we use as example a restaurant application[2].

2.1.1 Classification

Classification is the most important, and probably the best understood form of abstraction. It is achieved by grouping simple objects (called instances) that have common properties into a new composite object (called class) for which uniform conditions hold [Ma88]. Classification establishes an *instance-of* relationship between instances and class. Hence, it creates an one-level hierarchy. For example, suppose our restaurant offers four kinds of wines, namely, *bordeaux*, *cote-du-rhone*, *schwarzekatz*, and *liebfraumilch*. In such a case, we can congregate the common properties of all kinds of wines into a composite object called, for example, *wines* (Fig. 1).

2.1.2 Generalization

Generalization is the complementary concept of classification. It allows a more complex composite object (called superclass) to be defined as a collection of less complex composite ones (called subclasses). In other words, it extracts from one or more given classes, the description of a more general class that captures the commonalities but suppresses some of the detailed differences in the description of the given classes [Ma88]. Generalization establishes a *subclass-of* relationship between subclasses and superclass. Since it may be applied recursively, it creates an n-level hierarchy. Exemplifying, suppose our restaurant offers, besides *wines*, also some *aperitifs* (*pernod*, *champagne*, and *cointreau*) and *liquors* (*cointreau* and *chantre*). In this case, we can generalize these objects, creating a superclass named, for example, *beverages* (Fig. 1).

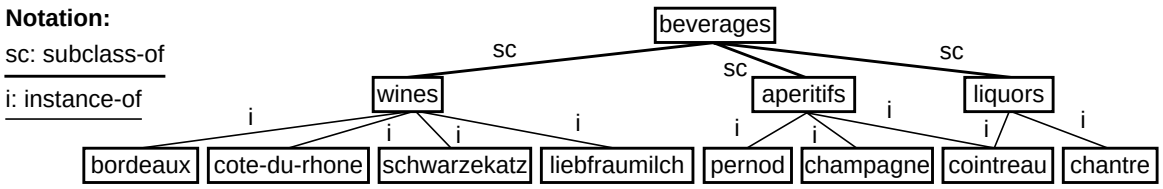

Figure 1: Example of classification and generalization.

Since the properties described in the superclasses are generalized properties of their subclasses, there is no need to describe over again these properties in the subclasses. This observation builds the most important characteristic of generalization, namely inheritance, by means of which the properties of the superclasses are reflected in the subclasses. This also holds for classification, i.e., the instances inherit the properties of their classes, which inherit from their superclasses, and so forth.

2.1.3 Association

There are two types of association, namely element- and set-association [Ma88]. Element-association allows the introduction of an object (called set) to describe some properties of a group of objects (called elements). It suppresses the details of the element objects whereas emphasizing the properties of the group as a whole. Hence, element-association creates an one-level hierarchy, and between elements and set an *element-of* relationship is established. For example, we could group the objects *bordeaux* and *cote-du-rhone* of our restaurant application into a set representing *french-wines*, and *schwarzekatz* and *liebfraumilch* into a set representing *rhine-wines* (Fig. 2).

2. This restaurant example to be used throughout the paper is a simplification of the first application modeled by means of the KBMS prototype KRISYS [Ma89].

On the other hand, set-association builds a more complex set object (called superset) in order to represent properties of a group of set objects (called subsets). Set-association establishes a *subset-of* relationship between subsets and superset. In addition, it may be applied recursively, thereby building an n-level hierarchy. Exemplifying, in our restaurant application, we could group the sets *french-wines* and *rhine-wines* into a superset representing *wine-origins* (Fig. 2).

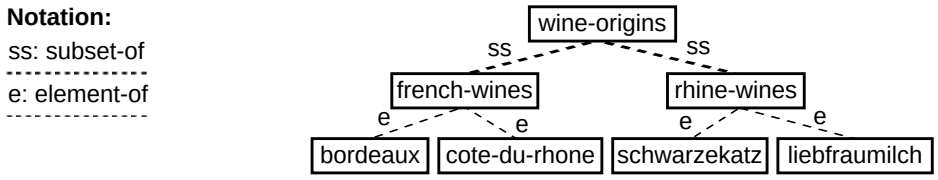

Figure 2: Example of element- and set-association.

2.1.4 Aggregation

Aggregation corresponds to the notion of property in the sense of composition. Like above, it involves two types of objects: Simple and composite [Ma88]. Simple, atomic objects (called elements or parts) are the ones which cannot be further decomposed. When they are aggregated in order to represent parts of a higher-level, composite object (called component), we are applying the element-aggregation concept, and the relationship between the parts and the component object is called *part-element-of* (or *part-of*, for short). Element-aggregation builds an one-level hierarchy. In turn, component objects (called subcomponents) may be used to build a more complex higher-level object (called supercomponent). This characterizes the component-aggregation concept, and between subcomponents and supercomponent, a *subcomponent-of* relationship is established. Since this concept may be applied recursively, it creates an n-level hierarchy.

However, aggregation is more stringently in the sense that it is used to express the idea that an object must have some necessary properties in order to exist consistently. For example, suppose our restaurant offers *mousse-au-chocolat* as a dessert. In turn, we could express that *mousse-au-chocolat* is composed of *mousse* and *cream* (Fig. 3). Clearly, it is hard to imagine a *mousse-au-chocolat* without either the *mousse* or the *cream*. This characteristic makes aggregation quite different from the other concepts.

Figure 3: Example of element-aggregation.

2.1.5 An Example Knowledge Base

The above explanations lead to the conclusion that KBMSs manage complex and structured objects, and also different types of abstraction relationships. In fact, one of the most important aspects of KBMSs is that objects can play different roles at the same time. Consequently, the KBs features can be visualized as a superposition of the generalization, classification, association, and aggregation hierarchies (in fact graphs), building altogether the so-called KB graph. In order to illustrate one such a KB graph, in Fig. 4 we provide a more detailed example, complementing the ones we have seen, of the restaurant application.

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 schemas are directly or indirectly related to the root *global*. When a schema is neither a class/instance, nor a set/element, nor a component/part, it is connected as a direct instance of *global*. In turn, all classes/instances, sets/elements, and components/parts are directly or indirectly related to the predefined schemas *classes, sets*, and *aggregates*, respectively. Moreover, we assume that the KB graph automatically stays in this form

(rooted and connected) as changes undergo over time[3]. At last, we use this KB throughout the paper.

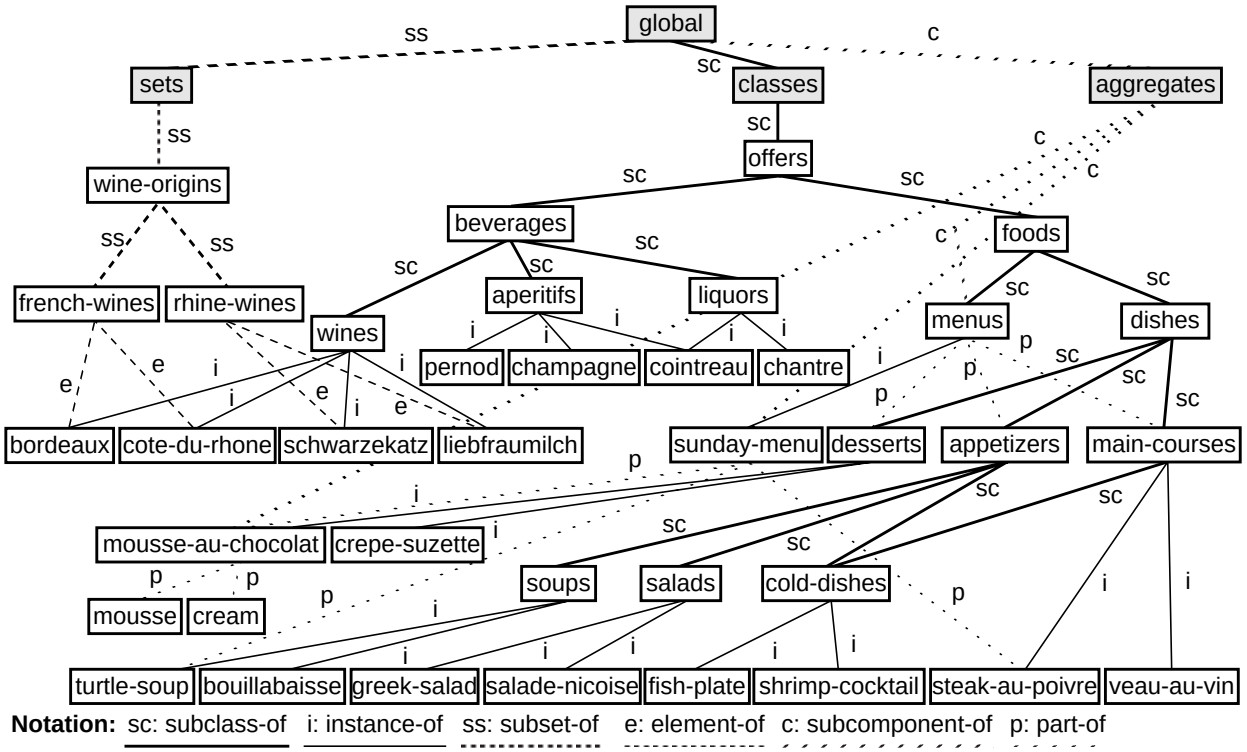

Figure 4: The restaurant knowledge base.

2.2 Methods

In the last years, there have been considerable efforts in order to increase concurrency by means of the semantic knowledge of transactions [Ga83, Ly83, FÖ89]. The main idea behind this use of transactions' semantics is to allow non-serializable schedules, which preserve consistency and are acceptable to the system users. With respect to KBMSs, the methods could be a starting point to the applicability of such an approach. 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. Hence, the semantics of user- or system-defined methods could be considered in order to allow more general, non-serializable schedules of transactions to be produced. Such a semantic knowledge use could decrease the transaction response time, and could then be useful when the cost of producing only serializable interleavings is unacceptably high. At the actual point of our work, we did not make any considerations about the semantics of methods yet. In particular due to the high cost and difficulty of determining the commit order of methods' operations. This research direction will be considered by us, in our future work, as soon as we get the basis of our protocol well-established and robust. Therefore, methods currently compete for locks like any other transaction request in our protocol (the same holds for the use of demons and rules).

3. Related Work

3.1 Some Important Challenges

Unlike conventional Database Management Systems (DBMSs), KBMSs impose some new and different

3. This representation and behavior are very similar to the ones used by KRISYS [Ma89] to represent KBs.

challenges to be coped with by CC techniques. The most of them is more or less related to the structure built by the objects in a KB (KB graph). In the following, we discuss four main points to be thought of when designing a CC technique for KBs.

Different granules of lock

In KBMSs, accesses normally refer to different and variable granules. Sometimes a transaction accesses just an object, be it for the manipulation of its slots, methods, aspects or whatever. However, it is also common to have accesses involving a set of objects. Nevertheless, such sets are not necessarily constant. On the contrary, they may span an object and its instances, or its elements, or parts, or even all descendants. Thus, a CC technique for KBMSs should provide the possibility of locking different granules of objects.

Large number of objects

Real world applications require the use of large amounts of knowledge for solving problems [Ma90]. In turn, the larger the amount of knowledge, the larger the number of objects in a KB. This large number of objects may cause problems to a CC protocol, due to the also large number of locks to be managed. Hence, it would be useful to have some mechanism to minimize the number of locks to be set by transactions, and consequently the number of locks to be managed by the lock manager. Such a mechanism might be realized by means of the well-known hierarchically organized lock structure of implicit locks [GLPT76]. Therefore, a CC technique for KBMSs should make use of implicit locks in order to improve the overall performance, in particular the one of the lock manager.

Multiple abstraction relationships to objects

As already discussed and exemplified (Fig. 4), the objects in a KB form a complex and dynamic graph structure. The complexity of such a structure is embodied by the many objects and relationships with different semantics composing it, whereas its dynamics is characterized by its flexibility in incorporating and dissimilating at any time objects and relationships. Therefore, it is possible to have multiple paths to objects at any time. This, in turn, imposes an extra overhead for protocols which for example want to make use of implicit locks, because an object with multiple parents may be accessed via a path that does not have any locks on it, thus violating the implicit locks and leading to inconsistencies. A CC technique for KBMSs should pay attention to such particularities.

Semantics of the relationships between objects

The relationships between objects in a KB are based on the abstraction concepts. In turn, each abstraction concept has a particular and special semantics. This semantics should be used by a CC technique for KBMSs, simply because it can increase the concurrency. For example, a user that wants to modify an object and its instances does not necessarily stay in conflict with another one manipulating the elements or parts of this same object. Such a conflict would exist if and only if some instance of this object is at the same time an element or a part of it. However, as long as such a situation does not happen, the instances, elements, and parts of an object build distinct and disjoint sets of objects. Thus, there is no point to prohibit parallel access to such sets. Therefore, a CC technique for KBMSs should interpret the semantics of the many relationships, and not at all generalize all of them as being merely structural connections between objects.

3.2 Existing Techniques and Respective Solutions

DDG policy

To the best of our knowledge, there is only one CC technique specially designed for KBMSs already published, namely the Dynamic Directed Graph (DDG) policy [CHM92, Ch94]. It is an extension of the locking protocol for hierarchical DBSs of Silberschatz and Kedem [SK80]. Whereas the former is able to

cope with cycles and updates in the underlying structure, this is not considered by the latter. Due to space limitations, we will not provide an exhaustive discussion of this protocol here[4]. Nevertheless, among the main drawbacks of this protocol, we can cite [Re94]: 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 at all exploited to improve the concurrency, and by this way a full exploitation of the possible parallelism is not reached. Second, no kind of implicit locks is defined. Thus, using the DDG protocol, to lock a class with thousands of instances, thousands of locks will be necessary. 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. As long as phantoms may happen, inconsistent results may be obtained by transactions.

Summarizing, with respect to the main challenges pointed out in the last subsection, the DDG policy neither provides different granules of lock, nor does it cope well with large number of objects, nor does it interpret the semantics of the relationships between objects. Finally, multiple paths to objects are handled by it, wherein a transaction following the DDG policy must start locking the *dominator*[5] of the set of nodes to be ever accessed, and it, before locking any node, must have already locked all ascendants of this node in the past and currently holds a lock on a least one of them. Although not being an elegant solution, because it may require a very large number of unnecessary locks whenever an object is likely to have many ascendants, it avoids the problem of multiple paths leading to an object.

Now, due to the lack of publications in this area, let us analyze some CC protocols of a related area, namely OODBMSs. Again, we will not provide extensive discussion of these protocols here[6], yet just some comments. Let us start with ORION.

ORION

The CC mechanism of the OODBMS ORION [BCGKWB87, KBCGW89, KGBW90] 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 hierarchy. Thus, ORION extended the GLP of Gray et al. [GLPT76] and by this way, it provides implicit locks [GK88, KBG89, KBCGW87]. Nevertheless, the restricted number of lock types used by ORION does not provide a teeming utilization of the parallelism. In addition, ORION does not allow, for example, a subclass of an object and an element of the same object to be written simultaneously, not even a read on a class to be performed in parallel with a write on an instance of it.

With respect to the main challenges, ORION partially supports different granules of locks in the sense we discussed. It applies the lock modes and locking protocol for a granularity hierarchy to a class lattice. But it does not provide granules of locking based on an object and its elements, for example. A large number of objects is also only partially well handled by ORION. It provides implicit locks for the instances of a class, but not, for example, for the subclasses of a class, or elements of a set. Multiple paths to objects are coped with by ORION by means of basically two proceedings. The first one is a restriction on its model, which prohibits an object of being instance of many classes at the same time. This eliminates the problem for the instances in the class lattices, but would be hardly applicable to KBMSs. In turn, classes may have several direct superclasses, and hence the problem appears for classes, and is solved with the requirement that for a query involving a class and its descendants, and 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. In summary, all subclasses of a class must be explicitly locked in such cases. Lastly, the semantics of the relationships between objects is also not completely considered by ORION, which interprets the semantics of classification and aggregation, but not

4. The reader is asked to see [Re94] for a detailed discussion and critical analysis of this protocol.
5. A dominator of a set of nodes in the KB graph is the one such that all paths from the root node to each node of this set of nodes pass through it. For more details, the reader is referred to [CHM92].
6. The reader is asked to see [Re94a] for a more detailed discussion of OODBMS CC techniques and their behavior in the KBMS environment.

the ones of association and generalization.

O_2

The CC technique used in the OODBMS O_2 [BDK92, CF92, Dx90, Dx91] is based on a classification of methods. 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 to an object. There are the so-called *real* and *virtual accesses* [CF92]. 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 also provided. Nevertheless, O_2 lacks of some concepts. Aggregates and sets are not taken into consideration. Thus, writes on a component or element of an object must be made in mutual exclusion, although not necessarily conflicting. In addition, no kind of implicit locks is provided for subclasses of a class.

O_2 supports only partially the mentioned challenges. It provides different granules of locks for instances, but not for classes, sets, and aggregates. Implicit locks are also considered by O_2 for instances, but not for classes, sets, and aggregates. Multiple paths to objects are 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. Finally, O_2 takes into consideration the semantics of classification, but not of generalization, aggregation and association.

GemStone

The OODBMS GemStone [MSOP86, PS87, BMOPS+89, BOS91] 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. First of all, 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 [Hä84, PR83, Mo92]. In turn, the pessimistic method of GemStone does not provide implicit locks, and so transactions may need to acquire a great number of locks, what can cause problems to the lock manager. Moreover, its limited number of lock types restricts the parallelism, and it is unaware about the semantics of the relationships between objects. Therefore, GemStone pays no attention to any of the challenges we pointed out.

ObjectStore

The CC mechanism of the OODBMS ObjectStore 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 [LLOW91]. Therefore, ObjectStore does not show any improvement with respect to CC. Finally, the many drawbacks of a page-based CC applied to OODBMSs, and even many more if they would be applied to KBMSs, will not be discussed in the scope of this paper.

4. The LARS[7] Protocol

4.1 Generalization of Granular Locks

Granular locks were first introduced by Gray et al. in [GLPT76]. The basic idea of the GLP comes from the choice of different lockable units, which are locked by the system to ensure consistency and to provide

7. ***L**ocks using **A**bstraction **R**elationships' **S**emantics.*

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 [GLPT76]. 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 [GR93], 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 [GLPT76]. The main benefit of the GLP is that it satisfies both of these situations, allowing 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 descendants of it 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. Those locks are used to sign the intention of a transaction to set locks at a finer granularity. Thus, the GLP 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 Directed Acyclic Graph (DAG)) (Fig. 5) [GLPT76].

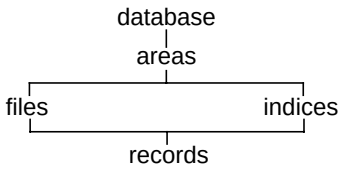

Figure 5: A lock graph for granular locks.

Notwithstanding, a protocol like the GLP is designed for a single organization hierarchy, extended to DAGs in case of index structures. If we would directly apply the GLP to a structure rich in semantics like KB graphs, we would not at all be able to interpret their edges. To put it another way, using the GLP, 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 ascendant. 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. This is the main reason why we refute to *directly* apply the GLP to KBs without modifications.

4.2 The Lock Modes

As we have seen, a KB graph is built through the superposition of the classification/generalization, association, and aggregation hierarchies (or in fact DAGs). However, many accesses in a KB are directed to a particular hierarchy, and not to the KB graph as a whole. Due to that, we are going to logically partition the KB graph into those three main hierarchies. As a result, we obtain a combination of different abstraction hierarchies, and we plan to apply hierarchical lock schemes on each one of them. By such a way, on one hand we acquire a minimization of the 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.

Thus, we create three different logical views from the whole KB graph. These are called the *classification* (which includes also generalization), *association*, and *aggregation* graphs. Obviously, these logical views are based on the abstraction relationships that are (or at least should be) provided by KBMSs. By this way, we provide users with the possibility of looking at a KB, and abstracting from it just the viewpoint to be worked out. In addition, this logical division is mirrored in each object of the KB. In order to exemplify it, let us recall our restaurant KB (Fig. 4). Suppose there are three transactions running on it. Transaction T1 wants to access all sets and elements of the KB, whereas T2 the object *menus* as a class and all its

instances, and finally, T3 the object *menus* as a component and all its parts. In such a case, the three transactions are provided with respectively the viewpoints a), b), and c) illustrated in Fig. 6.

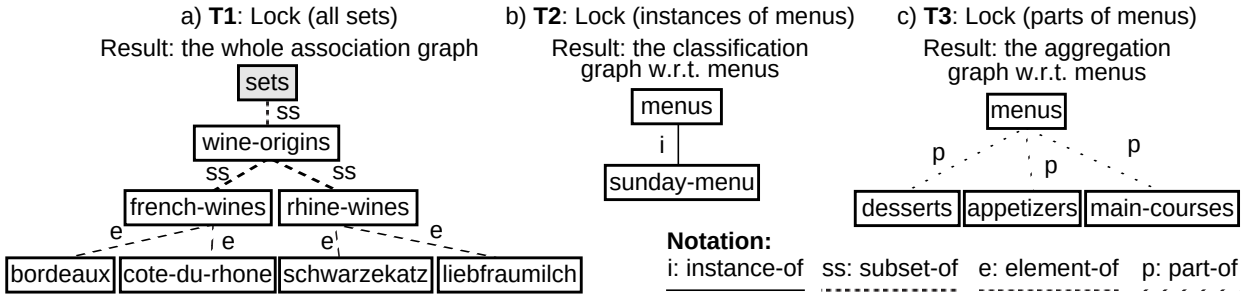

Figure 6: Different transactions' viewpoints.

Following these logical partitions, we have created three distinct sets of lock types. Hence, similar to the GLP, 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 our logical partitions, i.e., to the classification (recognized by a subscript c ($_c$) following the lock mode), association ($_s$), and aggregation ($_a$) graphs, and not to the whole structure as in the GLP. 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*). In Table 1, we present the semantics of each one of them.

4.3 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 [Gr78]. With respect to the compatibility of the above mentioned lock types, we have two distinct situations to cope with. These are discussed in the next subsections.

4.3.1 Compatibility of locks on the same sets of objects (equal 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 the GLP known from the literature [GLPT76, Gr78] (Fig. 7).

Granted Mode [c | s | a]

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

Figure 7: Compatibility matrix for typed locks of the same type.

In order to illustrate the meaning of the compatibility matrix of Fig. 7, let us think of two concurrent transactions operating in our restaurant KB (Fig. 4). Transaction T1 wants to read the object *appetizers* and all its subclasses and instances. Eventually, it may also need to update some of those descendants. In this case, T1 requests an RIW_c lock on *appetizers*, and then it automatically receives read access rights to this object and to all its subclasses and instances, being also allowed to request W_c locks on those, sometime. In turn, T2 wants to read the object *cold-dishes* and its instances. In order to accomplish that T2 needs to request an R_c lock on *cold-dishes*, but before doing that it must sign in the object *appetizers* its intention to read some descendants of it. Thus, it must request to the lock manager an IR_c lock on *appetizers*. Receiving this request, the lock manager compares the lock already hold by T1 on *appetizers* against the one requested by T2 with respect to the compatibility of both. As long as both locks, requested and granted, are compatible,

they may be conceded simultaneously, what is true in this case. This example is illustrated in Fig. 8.

Table 1: Typed locks' semantics.

IR_c	gives intention shared access to the requested object and allows the requester to explicitly lock both direct **subclasses** of this object in R_c or IR_c mode, and direct **instances** in R_c mode.
IW_c	gives intention exclusive access to the requested object and allows the requester to explicitly lock both direct **subclasses** of this object in W_c, RIW_c, R_c, IW_c or IR_c mode, and direct **instances** in W_c or R_c mode.
R_c	gives shared access to the requested object and implicitly to all direct and indirect **subclasses** and **instances** of this object.
RIW_c	gives shared and intention exclusive access to the requested object (i.e., implicitly locks all direct and indirect **subclasses** and **instances** of this object in shared mode and allows the requester to explicitly lock both direct **subclasses** in W_c, RIW_c, R_c or IW_c mode, and direct **instances** in W_c or R_c mode).
W_c	gives exclusive access to the requested object and implicitly to all direct and indirect **subclasses** and **instances** of this object.
IR_s	gives intention shared access to the requested object and allows the requester to explicitly lock both direct **subsets** of this object in R_s or IR_s mode, and direct **elements** in R_s mode.
IW_s	gives intention exclusive access to the requested object and allows the requester to explicitly lock both direct **subsets** of this object in W_s, RIW_s, R_s, IW_s or IR_s mode, and direct **elements** in W_s or R_s mode.
R_s	gives shared access to the requested object and implicitly to all direct and indirect **subsets** and **elements** of this object.
RIW_s	gives shared and intention exclusive access to the requested object (i.e., implicitly locks all direct and indirect **subsets** and **elements** of this object in shared mode and allows the requester to explicitly lock both direct **subsets** in W_s, RIW_s, R_s or IW_s mode, and direct **elements** in W_s or R_s mode).
W_s	gives exclusive access to the requested object and implicitly to all direct and indirect **subsets** and **elements** of this object.
IR_a	gives intention shared access to the requested object and allows the requester to explicitly lock both direct **subcomponents** of this object in R_a or IR_a mode, and direct **parts** in R_a mode.
IW_a	gives intention exclusive access to the requested object and allows the requester to explicitly lock both direct **subcomponents** of this object in W_a, RIW_a, R_a, IW_a or IR_a mode, and direct **parts** in W_a or R_a mode.
R_a	gives shared access to the requested object and implicitly to all direct and indirect **subcomponents** and **parts** of this object.
RIW_a	gives shared and intention exclusive access to the requested object (i.e., implicitly locks all direct and indirect **subcomponents** and **parts** of this object in shared mode and allows the requester to explicitly lock both direct **subcomponents** in W_a, RIW_a, R_a or IW_a mode, and direct **parts** in W_a or R_a mode).
W_a	gives exclusive access to the requested object and implicitly to all direct and indirect **subcomponents** and **parts** of this object.

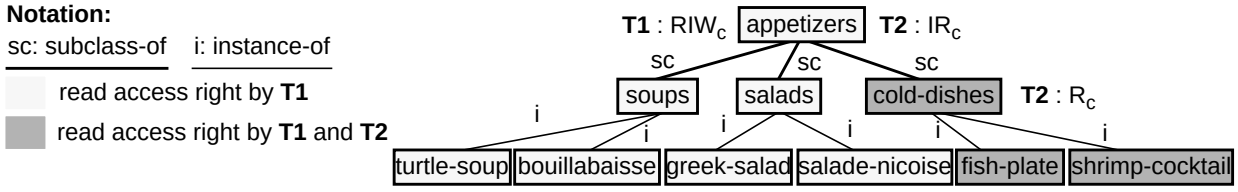

Figure 8: Concurrent transactions operating on the same set of objects.

4.3.2 Compatibility of locks on distinct sets of objects (different 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 example an extreme case, IW and W lock modes. In the GLP, these lock modes are incompatible, because if, for example, we set a W lock on a file, we automatically receive (implicit) W locks on all records of such a file. Therefore, there is no reason to allow another transaction to set an IW lock on this file because all records of it are already implicitly locked in a conflicting mode, and hence the transaction may not access any records anyway. The same observation holds in the LARS protocol for IW and W locks of the same type, of course due to the same reasons.

However, let us analyze a situation where two transactions require IW and W locks of different types on a same object, say *menus*. Suppose we have a physical representation of *menus* like the one sketched in Fig. 9. As we can see, all relationships of *menus* are represented in a bidirectional way. Let us consider these relationships top-down, like the way the transactions are going to request locks. Suppose T1 comes from *aggregates* and wants to write[8] the object *menus* and all its parts, namely *desserts*, *appetizers*, and *main-courses*. T1 must require then, in addition to an IW_a on *aggregates*, a W_a lock 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. 9, 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 lock 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, 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 lock 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. 9, 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 an W_a lock 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[9]. Therefore, when applied to distinct sets of objects, intention write and write lock modes are compatible.

Our discussion has shown 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 the same manner, the remaining lock modes of Fig. 10 may be shown to be compatible according to the given table. We leave this task for the reader. In Fig. 10, we can clearly see that our technique allows a much higher parallelism than the original GLP. 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.

---

8. 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, we explicitly use the terms *'to insert an object'* and *'to delete an object'*, respectively.
9. 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 *menus* itself.

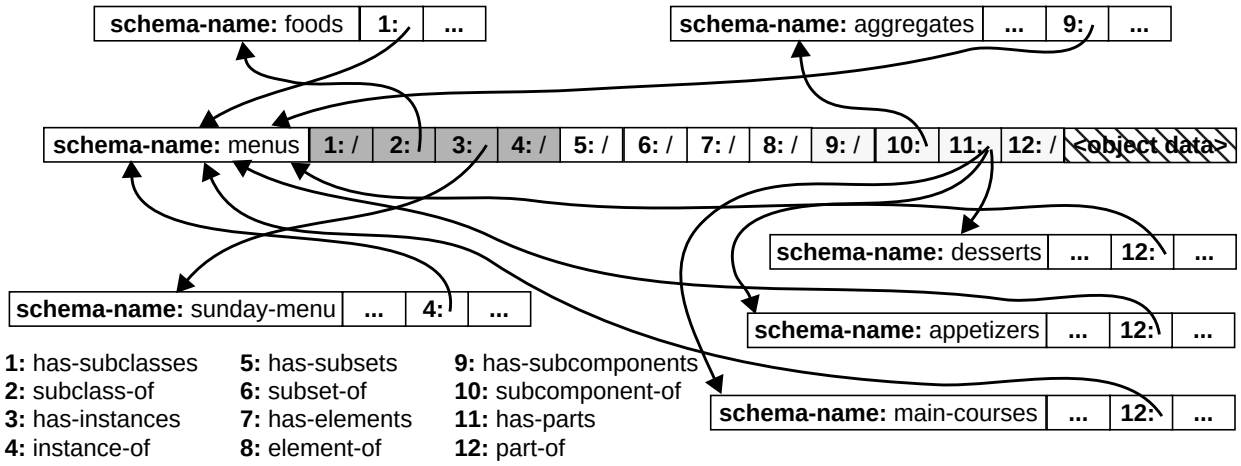

Figure 9: Physical representation of the object *menus*.

Granted Mode [c | s | a]

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

Figure 10: Compatibility matrix for typed locks of distinct types.

4.4 Accessing Implicitly Locked Objects

In Sect. 3.1, we have briefly discussed that multiple abstraction relationships to an object in a KB may lead to problems with the implicit locks, so that the isolation property of transactions [HR83] may be seriously corrupted. 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. 11. There, both transactions T1 and T2 required an IW_c lock on *beverages*. Both were granted because they are compatible. Thereafter, T1 followed the path to *aperitifs* and locked it in W_c mode. Then, it received automatically write access right not only to *aperitifs*, but also to its instances (*pernod*, *champagne*, and *cointreau*). Following another path, T2 locked *liquors* in W_c mode, and received also write access rights to its instances too (*cointreau* and *chantre*). T1 and T2 may get into troubles with one another, and for example a *lost update* may happen in the object *cointreau*. The problem here is that none of the transactions knows a priori which are the instances of those objects due to the dynamism of the KB graph; hence, both requested an explicit lock on a node in the hope that its descendants were locked as a whole implicitly.

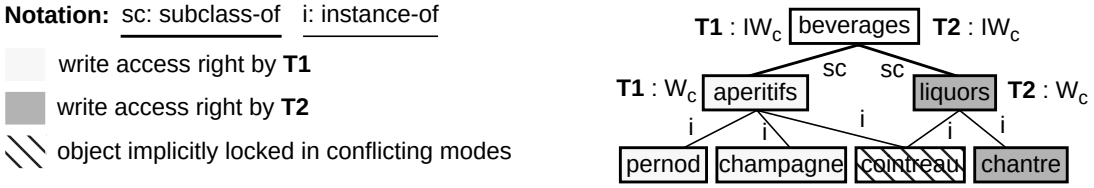

Figure 11: The problem with implicit locks in a graph structure.

In order to find out possible conflicts with implicitly locked objects, we may access all ascendants or descendants of an object. For this purpose, all relationships have to be represented in a bidirectional way. We could follow basically five possible approaches. Let us discuss all of them separately.

4.4.1 Lock all referenced objects

The first and most simple approach is to explicitly lock all referenced objects. In the example of Fig. 11, 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[10]. Nevertheless, this method leads to a great overhead, since many locks are required. In addition, it extinguishes the implicit locks, but as we have discussed we do want to use them in the KBMS environment due to their nice properties and potential advantages.

4.4.2 Search for conflicts

The second approach is, before accessing any implicitly locked bastard, to climb up the structure in order to find out possible conflicts. In this case, a conflict is detected if such a bastard is already implicitly locked by any other ascendant in a conflicting mode. In the above example (Fig. 11), 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 requires testing locks. In turn, 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. In addition, too many deadlocks may happen with this approach. In the current example (Fig. 11), a deadlock easily happens if T1 and T2 climb up the structure at nearly the same time. Lastly, this approach is too pessimistic in the sense that the object, although being implicitly locked, may be not updated yet, or even not be updated at all, what slackens the conflict and frees the transaction from any obligation of detecting it.

4.4.3 Analysis of all descendants

The third approach is, before setting any explicit lock on an object, to analyze all descendants of this object and explicitly lock the bastards[11]. As long as one proceeds so, any possible conflict is immediately avoided, because the objects where potential conflicts may happen, are already explicitly locked. In the current example (Fig. 11), this means that when transaction T1 sets a W_c lock on *aperitifs*, it needs also 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. This alternative is better than the previous one, but it still is too expensive. In this case, 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. In addition, this alternative is also somewhat pessimistic, because the transaction may not need to access all those descendants. At last, it may also lead to many deadlocks, like in the previous approach.

4.4.4 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 proceedings. However, 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

10. This alternative is followed by ORION for its class lattices (see Sect. 3.2).
11. This alternative was pointed out by Garza and Kim [GK88] for the class lattices in ORION, implemented for test purposes, but discarded.

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. 11), 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 needs then 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 on this bastard by T1 is already released, for example because T1 has already modified it and committed, T2 can receive the lock, but if T1 still holds the lock, T2 must wait. This proceeding is certainly more precise than the others. In addition, it involves less overhead because only the bastard descendants effectively accessed need to be explicitly locked. Those which are not accessed are not locked, what minimizes the overhead of the lock manager and increases the concurrency because they may be accessed by other transactions in the meanwhile. Hence, implicitly locked bastards not touched via some parent may be accessed via another one. Apparently, this approach also leads to deadlocks (to be discussed in Sect. 4.8.3). Let us summarize: This approach requires, in a set of already implicitly locked objects, explicit locks only for those objects that are actually accessed and that belong to more than one parent. 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 the LARS protocol.

4.4.5 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 already be 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 special cases, but in general it is too difficult to be realized and too expensive.

4.5 The Locking Rules

Having presented the general lines of the LARS protocol, we are finally able to expose its complete rules to be followed by transactions when requesting locks on objects in a KB (see Table 2).

4.5.1 The starting rule (1)

The first rule is clear when it states that transactions are allowed to directly set locks in the root object in any mode, without further requirements.

4.5.2 The rules for requesting typed locks (2-6)

The second rule, in turn, states that an intention read lock (from the C_type, S_type, or A_type) on a non-root object must be preceded by either intention read or intention write locks (from respectively the C_type, S_type, or A_type) on at least one parent of this object, and so recursively until the root object is reached. The third rule has a similar meaning, but for the intention write locks, requiring that they must be preceded by intention read or read intention write locks on at least one path from that object to the root object. The fourth rule states, first of all, that a read lock (from the C_type, S_type, or A_type) on a non-root object must be covered by intention read or intention write locks (from respectively the C_type, S_type, or A_type) on at least one path from this object to the root object. Thereafter, it requires that a transaction must explicitly lock the bastard descendants[12]. This is basically required for avoiding conflicts with implicitly locked objects. Thus, as already discussed, the LARS protocol makes a kind of lazy evaluation to detect conflicts in such

objects. If such an object is already locked in a conflicting mode via another parent of it, this transaction must wait until this lock is released. If not, the transaction is granted the required explicit lock on this object, and no other transaction will be able in the meanwhile to access it in a conflicting mode. The fifth and sixth rules have a similar meaning, but for read intention write and write locks, respectively.

Table 2: Locking rules.

1	The root object can be locked directly in any mode.
2	Before requesting an $IR_{c\|s\|a}$ mode lock on a non-root object, the requester must hold a path to the root in $IR_{c\|s\|a}$ or $IW_{c\|s\|a}$.
3	Before requesting an $IW_{c\|s\|a}$ mode lock on a non-root object, the requester must hold a path to the root in $IW_{c\|s\|a}$ or $RIW_{c\|s\|a}$ mode.
4	Before requesting an $R_{c\|s\|a}$ mode lock on a non-root object, the requester must hold a path to the root in $IR_{c\|s\|a}$ or $IW_{c\|s\|a}$ mode. In addition, before accessing any implicitly locked bastard descendant, the requester must set an $R_{c\|s\|a}$ lock on it.
5	Before requesting an $RIW_{c\|s\|a}$ mode lock on a non-root object, the requester must hold a path to the root in $IW_{c\|s\|a}$ or $RIW_{c\|s\|a}$ mode. 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.
6	Before requesting a $W_{c\|s\|a}$ mode lock on a non-root object, the requester must hold a path to the root in $IW_{c\|s\|a}$ or $RIW_{c\|s\|a}$ mode. In addition, before accessing any implicitly locked bastard descendant, the requester must set a $W_{c\|s\|a}$ lock on it.
7	Release all locks as soon as the transaction terminates (either commits or aborts).

We now provide an example (Fig. 12) using again our restaurant KB (Fig. 4). Suppose T1 wants to read the object *turtle-soup* as a part of the object *sunday-menu*. To do that it must follow rules 2 and 4 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 3 and 6 for requesting IW_c locks on the ascendants 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 6, 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.

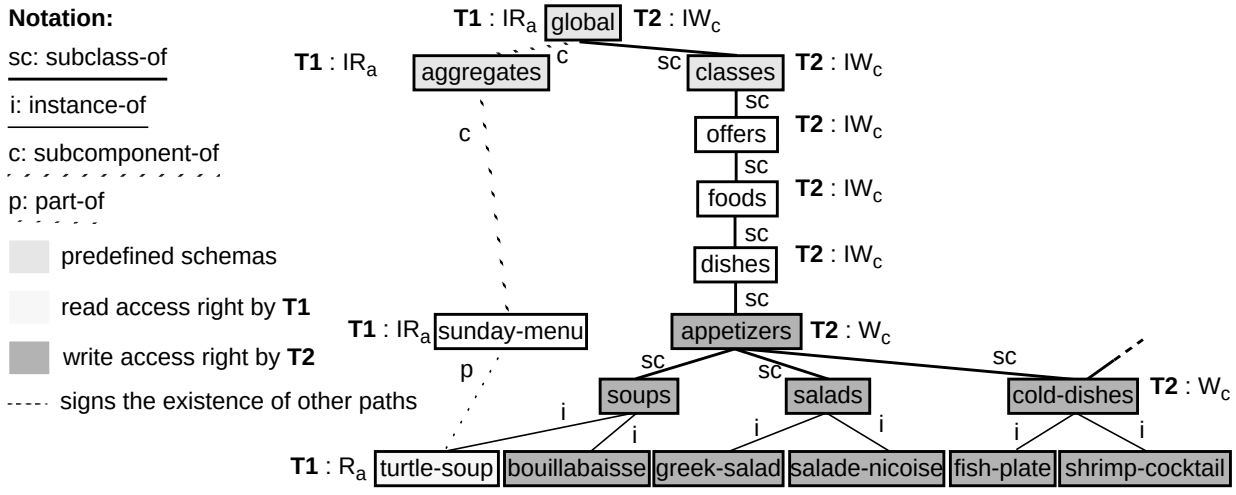

Figure 12: Avoiding conflicts with implicitly locked objects.

12. There may be situations where a descendant may have two edges pointing to the same ascendant. 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.

The most important benefit of explicitly locking bastard descendants, besides guaranteeing serializability, is the slackness of the original requirement of the GLP of covering all paths until the root, and as a consequence all ascendants, with intentions before granting an exclusive lock [GLPT76]. This is a serious limitation when an object has several ascendants and is likely to be used via many of them. For example, consider an aggregation relationship in our restaurant KB. An object, say *rice*, could be used in a large amount of the *menus*, all of which would be its parents. In such situations, it is very inefficient to set intention locks on all the parents [HDKRS89], 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, our approach significantly limits the overhead of the whole process of setting 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.

4.5.3 The commit rule (7)

The seventh rule is responsible for always producing *strict executions* [BHG87], when it requires the locks of a transaction to be released only at its end.

4.6 Correctness Concerns

The goal of the LARS protocol is to ensure that distinct transactions never hold conflicting (explicit or implicit) locks on the same object. In this section, we deliberate on aspects concerning correctness. Before passing on to the proof of the LARS protocol, we need some definitions.

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 x N). The set of arcs A is divided into three disjoint subsets A_c, A_s, and A_a ($A_c \cap A_s = A_c \cap A_a = A_s \cap A_a = \varnothing$). A_c contains the set of arcs of the classification and generalization abstraction concepts, A_s the set of arcs of the element- and set-association concepts, and, at last, A_a the set of arcs of the element- and component-aggregation concepts.

Definition 2: A node *p* is a ***parent*** of node *c*, and *c* is a ***child*** of node *p*, if $<p, c> \in A$.

Definition 3: A node with no parents is a ***root***. There is always only one root in G, namely, the predefined node ***global***.

Definition 4: A node with no children is a ***leaf***. To consider a node a leaf, one must pay attention to the set of arcs A_c, A_s, and A_a separately. For example, a node may be a leaf with respect to A_c, but not necessarily with respect to A_s or A_a.

Definition 5: A ***path*** is a set of arcs of A, $a_1 \ldots a_n$, where $a_i = <b_i, b_{i+1}>$, and $b_i \in N$.

Definition 6: Node *b* is an ***ancestor*** of node *c* if *b* = *c* or *b* lies in some path from the root to node *c*.

Definition 7: Node *b* is a ***proper ancestor*** of node *c* if *b* is an ancestor other than *c* itself.

Definition 8: 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 9: Node *c* is a ***proper descendant*** of node *b* if *c* is a descendant other than *b* itself.

Definition 10: A ***bastard*** is any node which has more than one parent.

Definition 11: A ***purebred*** is any node which has only one parent.

Theorem 1: Suppose all transactions obey the LARS protocol with respect to a given lock graph, G, that is a DAG. If a transaction owns an explicit lock on a bastard node of G or an explicit or implicit lock on a purebred node of G, then no other transaction owns a conflicting explicit lock on that bastard node or a conflicting explicit or implicit lock on that purebred node.

Proof: *In theorem (1), we may notice that implicitly locked bastards are excluded, i.e., only explicitly locked bastards are referenced. This observation comes from the fact that rules (4), (5), and (6) require, for the eventually implicitly locked bastards, explicit locks on those before the access may take place. Hence, any eventual implicit lock conflicts on bastards may be immediately discarded by the provided compatibility matrices, when those are turned on to explicit locks. Therefore, conflicts on bastards are obviously impossible.*

*It is sufficient to prove the theorem for leaf nodes, because if two transactions held conflicting (implicit or explicit) locks on a non-leaf node, they would be holding conflicting implicit locks on all purebred descendants as well as explicit locks on all touched bastard descendants and, in particular, all leaf descendants of this non-leaf node. Suppose that transactions Ti and Tj own conflicting locks on an object, say O. There are fifty-four different possibilities of conflict on O, which can be grouped, per similarity, in five cases (see Table 3, where **i** stands for implicit and **e** for explicit locks).*

Table 3: Possible conflicts on an object O.

Transaction Ti \ Transaction Tj	iR_c	eR_c	iW_c	eW_c	iR_s	eR_s	iW_s	eW_s	iR_a	eR_a	iW_a	eW_a
iR_c			1	2			3	4			3	4
eR_c			2	5			4	5			4	5
iW_c	1	2	1	2	3	4	3	4	3	4	3	4
eW_c	2	5	2	5	4	5	4	5	4	5	4	5
iR_s			3	4			1	2			3	4
eR_s			4	5			2	5			4	5
iW_s	3	4	3	4	1	2	1	2	3	4	3	4
eW_s	4	5	4	5	2	5	2	5	4	5	4	5
iR_a			3	4			3	4			1	2
eR_a			4	5			4	5			2	5
iW_a	3	4	3	4	3	4	3	4	1	2	1	2
eW_a	4	5	4	5	4	5	4	5	2	5	2	5

Case 1*. Under case (1) are classified all possible conflicts involving conflicting implicit locks of the same type. We use as example the assumed conflict: Ti holds implicit R_c [O] and Tj holds implicit W_c [O]. Since O is implicitly locked by Ti in R_c mode, Ti owns an explicit R_c lock on some proper ancestor Y of O and, additionally, explicit R_c locks on the set of nodes Z, ancestors of O and proper descendants of Y, which are bastards (rule (4)). Since O is implicitly locked by Tj in W_c mode, Tj owns an explicit W_c lock on some proper ancestor Y' of O and, additionally, explicit W_c locks on the set of nodes Z', ancestors of O and proper descendants of Y', which are bastards (rule (6)). There are three subcases: (a) $Z = Z' = \emptyset$, (b) $(Z \neq \emptyset \vee Z' \neq \emptyset) \wedge Z \cap Z' = \emptyset$, and (c) $Z \cap Z' \neq \emptyset$. Subcase (a) has in addition three subcases: (a1) Y = Y', (a2) Y is a proper ancestor of Y', and (a3) Y' is a proper ancestor of Y. Subcase (a1) is excluded by the given compatibility matrices, therefore impossible, because Ti and Tj are holding conflicting R_c and W_c locks (respectively) on Y = Y'. Subcase (a2) is impossible, because Tj must own IW_c [Y], which conflicts with R_c [Y]. Subcase (a3) is impossible, because Ti must own IR_c [Y'], which conflicts with W_c [Y']. Subcase (b) is*

obviously impossible because if neither Z nor Z' are empty, they must have at least one common node, otherwise, it would be impossible to implicitly lock O from two different paths. Subcase (c) is impossible because Ti and Tj would be holding conflicting R_c and W_c (respectively) locks on a same node $z \in (Z \cap Z')$. Thus, the assumed conflict is impossible. The other conflicts listed as case (1) are similar to this one, and follow therefore the same arguments.

***Case 2**. Under case (2) are classified all possible conflicts involving an implicit lock and an explicit lock, both of the same type. We use as example the assumed conflict: Ti holds implicit R_c [O] and Tj holds explicit W_c [O]. Since O is implicitly locked by Ti in R_c mode, Ti owns an explicit R_c lock on some proper ancestor Y of O and, additionally, an explicit R_c lock on every node Z, ancestor of O and proper descendant of Y, which is a bastard (rule (4)). Since O is explicitly locked by Tj in W_c mode, Tj owns IW_c locks on a path from O to the root (rule (6)). There are three subcases: (a) Tj holds IW_c [Y], (b) Tj holds IW_c [Z], and (c) Tj holds both IW_c [Y] and IW_c [Z]. All of them are impossible because the lock types IW_c and R_c conflict. The other conflicts listed as case (2) are similar to this one, and follow therefore the same arguments.*

***Case 3**. Under case (3) are classified all possible conflicts involving implicit locks of different types. We use as example the assumed conflict: Ti holds implicit R_c [O] and Tj holds implicit W_s [O]. Since O is implicitly locked by Ti in R_c mode, Ti owns an explicit R_c lock on some proper ancestor Y of O, which lies in a path p ($p \subset A_c$) from O to the root (rule (4)). Since O is implicitly locked by Tj in W_s mode, Tj owns an explicit W_s lock on some proper ancestor Y' of O, which lies in a path q ($q \subset A_s$) from O to the root (rule (6)). By definition (1), $A_c \cap A_s = \emptyset$, therefore $p \neq q$. By definition (10), O is a bastard. By rule (4), Ti must own R_c [O] before accessing O. By rule (6), Tj must own W_s [O], what is impossible because Ti owns a conflicting lock on O (namely, R_c [O]). Hence, the assumed conflict is impossible. The other conflicts listed as case (3) are similar to this one, and follow therefore the same arguments.*

***Case 4**. Under case (4) are classified all possible conflicts involving an implicit lock and an explicit lock, both of different types. We use as example the assumed conflict: Ti holds implicit R_c [O] and Tj holds explicit W_s [O]. Since O is implicitly locked by Ti in R_c mode, Ti owns an explicit R_c lock on some proper ancestor Y of O, which lies in a path p ($p \subset A_c$) from O to the root (rule (4)). Since O is explicitly locked by Tj in W_s mode, Tj owns IW_s locks on a path q ($q \subset A_s$) from O to the root (rule (6)). By definition (1), $A_c \cap A_s = \emptyset$, therefore $p \neq q$. By definition (10), O is a bastard. By rule (4), Ti must own R_c [O] before accessing O, what is impossible because Tj owns W_s [O], which conflicts with R_c [O]. Hence, the assumed conflict is impossible. The other conflicts listed as case (4) are similar to this one, and follow therefore the same arguments.*

***Case 5**. Under case (5) are classified all possible conflicts immediately excluded by the compatibility matrices. All of them are obviously impossible.* ❑

4.7 Coping with Insert and Delete Operations - The Phantom Problem

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 and delete existing relationships and objects. These operations are supported by the following complementary rules (Table 4).

Table 4: Locking rules for insert and delete operations.

8	Before inserting an object, the requester must hold the parent in $IW_{c\|s\|a}$, $RIW_{c\|s\|a}$ or $W_{c\|s\|a}$ mode.
9	Before deleting an object, the requester must hold it in $W_{c\|s\|a}$ mode.
10	Before inserting an edge, the requester must hold the descendant in $W_{c\|s\|a}$ mode, and the parent in $IW_{c\|s\|a}$, $RIW_{c\|s\|a}$ or $W_{c\|s\|a}$ mode.
11	Before deleting an edge, the requester must hold the descendant in $W_{c\|s\|a}$ mode.

4.7.1 The rules for insertion and deletion of objects (8-9)

The eighth rule copes with the insertion of objects in the KB. Before explaining this rule, it is convenient to notice that firstly, we assume that an object is always inserted with at least one relationship (when no relationship is provided by the user, the object is considered to be an instance of *global*, see Sect. 2.1), and secondly, when the user specifies more than one relationship, the LARS protocol treats such cases as being an insertion of an object with one relationship, followed by as many insertions of edges (governed by rule 10) as asserted by the user. Hence, the eighth rule must always handle the object and its single parent. Finally, it states that before inserting an object, its parent must be held in at least intention write mode (and so recursively until the root object is reached). The type of such an intention write mode is dictated by the abstraction relationship being inserted, i.e., C_type for classification/generalization, S_type for association, or A_type for aggregation. Fig. 13 provides an example of the lock requests needed for inserting an object. 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 lock on *wines*, the parent of *cote-de-provence*. In turn, this IW_c lock must be covered by IW_c locks 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, the lock manager grants a W_c lock on this object to T1, which holds it until it terminates. An important point to be considered here is the possible occurrence of phantoms in this operation. Phantoms are discussed in Sect. 4.7.3.

In turn, the ninth rule deals with deletion of objects in a similar way, with the extra requirement that the object itself must be held in write mode. Notice that such a write lock implies intention write locks on a parent, on a parent of the parent, and so forth until the root is reached. Also similar to the insertion, when an object with several parents is to be deleted, the LARS protocol treats such cases as many deletions of edges (governed by rule 11) as necessary, until the object has only one parent. Finally, the type of such write and intention write locks are dictated by the abstraction relationship in question.

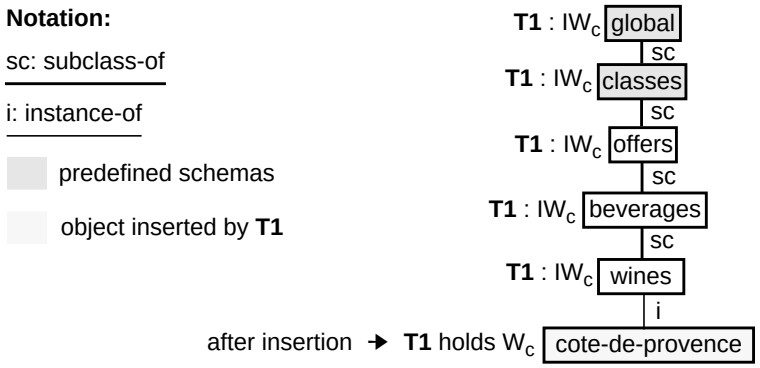

Figure 13: Locks for the insertion of an object.

4.7.2 The rules for insertion and deletion of edges (10-11)

The tenth rule copes with the insertion of edges in the KB graph, i.e., the creation of new relationships between objects. It states that for inserting an edge, the object must be held in write mode, and the parent in at least intention write mode, according to the abstraction relationship being inserted. Fig. 14 illustrates the use of this rule, complementing the last example. Suppose that T1 wants to connect the recently created

object *cote-de-provence* as an element of *french-wines*. Following the rule 10, T1 must request a write lock on this object, preferentially 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. In such particular cases, a transaction is allowed to acquire a write lock of another type. Since *cote-de-provence* is an instance of *wines*, 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. Thereafter, T1 must require an IW_s lock on *french-wines*, the new parent of it. In turn, this intention lock requires intention on the parents, recursively. Finally, after holding all the required locks, T1 creates the new relationship. As we can see, the insertion of an edge is a bit more complicated operation, because the transaction does not know a priori which are the roles of the object itself and its parent in the current state of the KB. The same does not happen for the deletion of an edge, which is treated by the rule 11. In such cases, the transaction does know the current role of the objects, and by this way the path it must traverse for requesting locks.

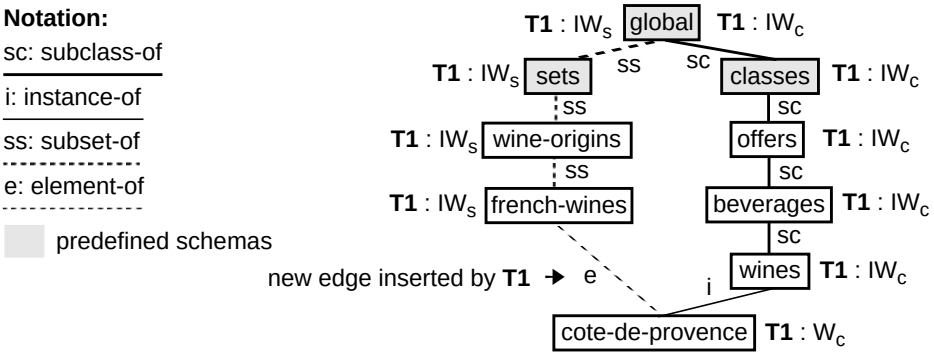

Figure 14: Locks for the insertion of an edge.

4.7.3 The Phantom Problem

As stated by Bernstein et al. [BHG87], the *phantom problem* is the concurrency control problem for dynamic databases. Granular locks provide physical locks, and being so we have problems with the so-called phantoms also in the LARS protocol. 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 [EGLT76], which also proposed the *predicate locks* for elegantly coping with such situations. Since we do not have predicates in the LARS protocol, we must deal with them 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 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 no other transaction is able to access any descendant of such an object, with respect to the working graph. Exemplifying, if a transaction wants to insert an instance of some class, it must request a W_c lock on the class. Thereafter, any other transaction may neither read nor write any instance of this class, due to the W_c lock hold by the inserting transaction on it, and hence no phantom appears. On the other hand, such a measure may significantly decrease the concurrency because the object as well as a certain subset of its descendants stay inaccessible to other transactions for the time the insert or delete transaction is running. Therefore, due to such pros and cons, we decided to delegate to transactions the choice among either greater concurrency with phantoms or lower concurrency without phantoms.

4.8 Final Considerations and Future Work

4.8.1 Including Indices in the KB Graph

Thus far, we have pretended that all accesses against a KB take place through the abstraction relationships. However, the 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. Therefore, we must expand the LARS protocol to work for KB graphs with indices. Adapting the main ideas of the GLP with respect to indices to the KBMS environment, we first come to the following observations, which could be used as a starting point for a solution:

(1) The indices could be treated as root nodes. Hence, we would have several roots in the KB graph, and not only one as supposed so far. The main root would still be the predefined schema *global* (*global root*), and all indices would build other roots (*index roots*).

(2) Typed read ($R_{c|s|a}$) locks on a node would require that all nodes on *at least one path* from that node to *at least one (any) root* be covered by typed intention read ($IR_{c|s|a}$) locks.

(3) Typed write ($W_{c|s|a}$) locks on a node would require that all nodes, first, on *at least one path* from that node to the *global root*, and, second, on *all paths* from that node to *all index roots* be covered by typed intention write ($IW_{c|s|a}$) locks.

This is the simplest adaptation of the GLP to our environment. On one side, the acquirement of read locks is made easier, because locking just one path to any index root in intention read mode suffices, and such a path is potentially shorter than one to the global root. On the other side, write locks are costly to be acquired, because all paths to all index roots must be locked in intention write mode, beyond a path to the global root itself. Unfortunately, this topic is more complicated than it may seem. First of all, special index structures must be tailored for the KBMS environment, and thereafter the CC protocol must be adapted to correctly work with such structures. At the actual point of our work, we have not made enough considerations about this topic, so that to make a reasonable and concrete proposal here. We are working on this topic, and leave such a proposal for a later publication.

4.8.2 Multiple Inheritance

The multiple abstraction relationships to an object may lead to a particular situation where a transaction requests incompatible locks on the same object. When multiple inheritance is allowed, this situation may happen even more frequently. Suppose the simple scenario sketched in Fig. 15. There, transaction T1 holds a W_c lock on object O4, which was granted via the path O1-O2-O4. After holding such a lock, T1 traverses the path O1-O3-O4, and requests again a W_c lock on the same object O4. In such cases, the lock manager must be aware of which transaction is requesting the incompatible lock. If the requesting transaction is the same one which holds the incompatible lock, then the lock manager must simply grant the lock to it again. On the other side, if the lock manager just checks whether the object is already locked in an incompatible mode or not, it may produce situations where a transaction must wait for itself, and even worst, a transaction may get deadlocked with itself. Therefore, the lock manager of the LARS protocol verifies, in case of conflicting lock requests on the same object, which particular transaction is currently holding the lock. If both transactions are the same, such a lock request is treated as being granted, if not, the requesting transaction must wait until the conflicting lock is released.

4.8.3 Deadlocks

Deadlocks happen whenever there is a cyclical sequence of transactions each waiting for the next to release a lock it must acquire (T1 → T2 → ... → T1). As a matter of fact, granular locks are subject to deadlocks. Thus, the LARS protocol is also susceptible to deadlocks. However, it suffers from a kind of deadlock which does not happen in the GLP, namely, deadlocks with (implicitly locked) bastard objects. The

difference is that the rules 4, 5, and 6 of the LARS protocol require also explicit locks on those implicitly locked bastards, what is not done by the GLP.

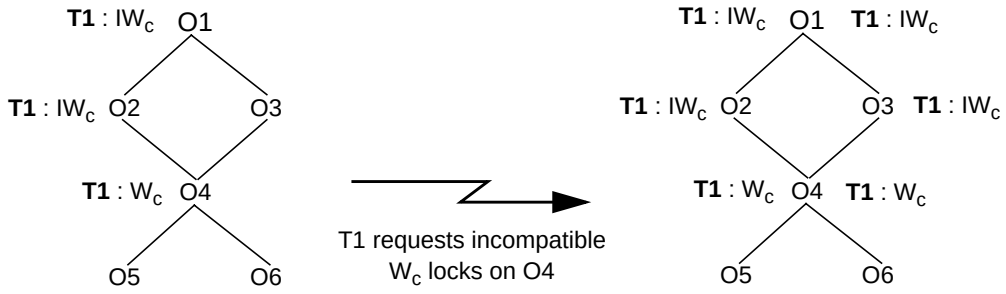

Figure 15: Incompatible lock requests by the same transaction via different paths.

Fig. 16 illustrates a scenario where such a deadlock occurs. There, T1 locked the object *dishes* in W_c mode, and, according to the semantics of this lock, all its subclasses and instances were implicitly locked. Hence, T1 starts performing some modifications. But when accessing *turtle-soup*, it realizes that this object has another parent (*sunday-menu*), and requires then a W_c lock on this object, which is granted because this object is free. In the meanwhile, T2 locked the object *sunday-menu* in W_a mode, and by this way all its subcomponents and parts implicitly. In a similar way, when accessing *steak-au-poivre*, T2 detects that this object has another parent (*main-courses*), and requests thus a W_a lock on it, which is granted. Thereafter, T2 tries to access the object *turtle-soup*. In the same manner, it must request a W_a lock on it, due to its other parent. This lock is not granted because T1 holds a lock on this object which conflicts with the one requested by T2. T2 must then wait until T1 terminates. On the other side, T1 goes ahead and requests a W_c lock on the object *main-courses*, also due to its other parent, which is granted. Thereafter, it similarly requests a W_c lock on *steak-au-poivre*. Unfortunately, T1 must wait until T2 terminates because a conflicting lock on this object is held by T2. Therefore, if no corrective measures are taken in such a situation, T1 and T2 will wait for each other forever, what characterizes a deadlock.

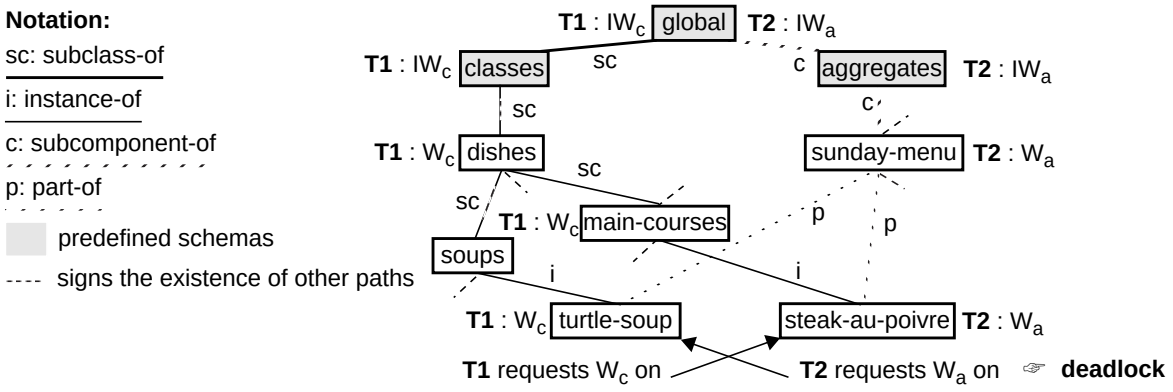

Figure 16: Deadlocks with bastard objects.

With respect to deadlocks, Gray and Reuter [GR93] advocate that these are very, very rare events, and Härder [Hä84] states that there are no general purpose deadlock-free locking protocols that always provide a high degree of concurrency. On the other side, Yannakakis [Ya82, Ya82a] holds the opinion that it is very easy to construct scenarios where deadlocks arise, and Silberschatz and Kedem [SK80] state that deadlock detection and recovery in general is an expensive task and should be avoided whenever possible. Anyway, our technique unfortunately does not avoid deadlocks, and so we will have an extra overhead for detecting and resolving them. There are a lot of strategies to detect deadlocks. One of them is *timeout*, where the system, finding that a transaction is waiting too long for a lock, just guesses that there may be a deadlock involving this transaction and simply aborts it and restarts it later again (although imprecise in the detection of deadlocks, it works). *Waits-for-graph* [Ho72] is another strategy, where the system maintains a graph

showing which transactions are waiting for other ones. When a cycle is found in this graph, it means precisely that the transactions in the cycle are deadlocked. The system then chooses one of them as a *victim*[13], aborts it, obliterating its effects from the database, and restarts it later again. Particularly, we feel that timeout does not always offer an optimal solution to deadlocks. Although being very easy to implement, the number of transactions that may be unnecessarily aborted and restarted again may be unacceptably high, due to the impreciseness of this technique. On the other side, waits-for-graph shows a very good precision for all kinds of transactions, independently of their duration. An implementation and comparison between both techniques is subject of our future work.

4.8.4 Lock Conversion

An important concept covered by the GLP is *lock conversion* [GLPT76]. Lock conversions are normally used to increase (upgrade) the access mode a transaction has to an object (for example, if a transaction has read some object and wants to modify it afterward, it can request to the system to upgrade its lock on this object from shared to exclusive mode). Thus, all the system must do is a comparison between the currently granted lock mode of the requester to the resource and the newly requested lock mode. The new mode will be either the most restrictive between the old and the requested mode, or a combination of both.

In the LARS protocol, lock conversions are handled accordingly to the type of lock. To put it another way, a transaction may upgrade a C_type lock to another one of this same C_type, but not to one of A_type or S_type, for example. Fig. 17 shows the lock conversion table for typed locks. To give an example, if one has an IW_c lock on an object and requests an R_c lock on it, then the new mode is RIW_c.

Requested Mode [c | s | a]

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

Figure 17: Lock conversion table.

4.8.5 Lock Escalation

Another important issue with respect to granular locks is *lock escalation* [GR93, BHG87]. A system employing granular locks must decide the level of granularity at which transactions should be locking. Generally, a fine-granularity locking is used as default, unless the system has some hint that the transaction is likely to access several lockable units covered by the current lock mode[14]. In such cases, they may get a single, coarse-granularity lock. The past history of a transaction's behavior can be also used to predict the need for coarse-granularity locks [BHG87]. After the transaction has acquired more than a certain number of locks (usually set to 1000 according to Gray and Reuter [GR93]) of a given granularity, the system executes some kind of heuristic to convert fine-granularity locks to coarse locks, requesting locks at the next higher level of granularity. This process of trading fine-granularity locks for coarse ones is called *lock escalation*. Albeit it may cause waiting or lead to deadlocks, it is an important aspect for improving the performance of the CC protocol and for preventing the lock system to run out of storage when millions of locks are acquired.

---

13. This choice is not always a simple decision due to another problem, worse than deadlock, because it is harder to detect and wastes resources, named *livelock*. Livelocks are situations where each member of the livelock set may soon want to wait for another member of the set, resulting in another abort and restart [GR93].
14. As an example of a hint, in some implementations of SQL, there is an SQL statement to explicitly lock an entire table. With such a hint, the system can immediately try to lock the whole table.

Lock escalation is also taken into consideration by the LARS protocol. By means of it, for example, if a transaction has acquired an IW_c mode lock on a class, and starts requesting too many W_c locks on its instances, the LARS protocol will try to grant a W_c lock to this transaction on this class, implicitly locking all its instances and alleviating its task of requesting so many explicit locks. Nevertheless, there is an aspect we must still cope with. As stated by the locking rules 4, 5, and 6 of the LARS protocol, the transaction must request also explicit locks on implicitly locked bastards. Due to this requirement, a lock escalation in the LARS protocol alleviates the transaction from requesting locks just on purebred descendants, but the same is not true for the bastards, because the transaction, obeying the locking rules, must still request locks on those objects. We believe this aspect will not prejudice too much the performance of lock escalation in the LARS protocol. Notwithstanding, if the LARS protocol is likely to be faced with situations like, for example, a class has thousands of instances and all of them are bastards, then it may labor under difficulties, because even with a lock escalation, a transaction requesting a lock on this class will be forced to request thousands of locks on its instances, due to their other parents. A possible solution to this problem is to provide a lock escalation covering all the parents of the objects being locked, and not only on the parent the transaction has acquired an intention lock. In this case, with all parents being explicitly locked, their descendants could be accessed without further synchronization needs. However, this proceeding, besides the possibility of being very costly, could also decrease the concurrency, because too many objects would be explicitly locked by just one transaction. A detailed investigation of this aspect is also another point we will take into consideration in our future work.

5. 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 a CC technique tailored for KBMSs. The most important characteristic of our technique 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, we have captured more of the semantics contained in the KB graph in the sense that we do 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 our technique, by means of which we can really obtain a high degree of concurrency, with a full exploitation of all inherent parallelism in a knowledge representation approach.

In addition, we have enumerated and commented some important challenges to be coped with by a CC technique for KBMSs. First, the LARS protocol offers different granules of locks. Second, it considers implicit locks, alleviating the task of managing too many locks due to the high number of objects in real world applications. Thirdly, it 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. Fourth, it interprets the relationships between objects with respect to their semantics, providing typed locks for all abstraction concepts. Finally, such power, flexibility, and parallelism are by no means proffered by the related works we criticized.

At last, we have also discussed some topics of our future work in the final considerations. All those aspects are going to be considered in the implementation of our technique. Particularly, we are going to use the

KBMS prototype KRISYS [Ma89], developed at the University of Kaiserslautern, as a practical vehicle for the implementation. Of course, we shall not forget that the fundamental attribute of each successful CC mechanism implementation is performance. Therefore, in our implementation, we are going to take into account whatever aspect we can to improve throughput and decrease response time. We hope, in the near future, to have the opportunity of preparing another paper with detailed aspects of the implementation of our technique, considering pros and cons of the concepts and realization strategies we follow.

Acknowledgments

We would like to thank J. Reinert for the helpful discussions about the correctness concerns of our protocol.

References

[BCGKWB87] Banerjee, J., Chou, H.-T., Garza, J.F., Kim, W., Woelk, D., Ballou, N.: Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, Jan. 1987. pp. 3-26.

[BDK92] Bancilhon, F., Delobel, C., Kanellakis, P. (Eds.).: *Building an Object-Oriented Database System: The Story of O_2*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1992.

[BHG87] Bernstein, P.A., Hadzilacos, V., Goodman, N.: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, MA, USA, 1987.

[BMOPS+89] Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Willians, E.H., Willians, M.: *The GemStone Data Management System*. In: Kim, W., Lochovsky, F.H. (Eds.), Object- Oriented Concepts, Databases, and Applications, ACM Press, New York, USA, 1989. pp. 283-308. (Chapter 12).

[BOS91] Butterworth, P., Otis, A., Stein, J.: The GemStone Object Database Management System. *Communications of the ACM*, Vol. 34, No. 10, Oct. 1991. pp. 64-77.

[CF92] Cart, M., Ferrié, J.: *Integrating Concurrency Control into an Object-Oriented Database System*. In: [BDK92]. pp. 463-485.

[Ch93] Chaudhri, V.K.: *On the Performance of a Multi-User Knowledge Base Management System*. Internal Report, University of Toronto, Toronto, Canada, Dec. 1993.

[Ch94] Chaudhri, V.K.: *Transaction Synchronization in Knowledge Bases: Concepts, Realization and Quantitative Evaluation*. Doctor Thesis, University of Toronto, Toronto, Canada, 1994.

[CHM92] Chaudhri, V.K., Hadzilacos, V., Mylopoulos, J.: Concurrency Control for Knowledge Bases. In: *Proc. of the 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, Cambridge, USA, 1992.

[Dx90] Deux, O. et al.: The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990. pp. 91-108.

[Dx91] Deux, O. et al.: The Story of O_2. *Communications of the ACM*, Vol. 34, No. 10, Oct. 1991. pp. 34-48.

[EGLT76] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, Vol. 19, No. 11, Nov. 1976. pp. 624-633.

[FÖ89] Farrag, A.A.; Özsu, M.T.: Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, Vol. 14, No. 4, Dec. 1989. pp. 503-525.

[Ga83] Garcia-Molina, H.: Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, Vol. 8, No. 2, June 1983. pp. 186-213.

[GK88] Garza, J.F., Kim, W.: Transaction Management in an Object-Oriented Database System. In: *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, Chicago, USA, June 1988. pp. 37-45.

[GLPT76] Gray, J.N., Lorie, R.A., Putzolu, G.R., Traiger, I.L.: Granularity of Locks and Degrees of Consistency in a Shared Data Base. In: *Proc. of the IFIP Working Conference on Modeling in Data Base Management Systems*, Freudenstadt, Germany, Jan. 1976. pp. 365-394.

[GR93] Gray, J.N., Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1993.

[Gr78] Gray, J.N.: *Notes on Database Operating Systems*. In: Operating Systems: An Advanced Course, Springer Verlag, Berlin, 1978. (Lecture Notes in Computer Science No. 60).

[Hä84] Härder, T.: Observations on Optimistic Concurrency Control Schemes. *Information Systems*, Vol. 9, No. 2, 1984. pp.111-120.

[HDKRS89] Herrmann, U., Dadam, P., Küspert, K.M., Roman, E.A., Schlageter, G.: *A Lock Technique for Disjoint and Non-Disjoint Objects*. Technical Report No. TR.89.01.003, IBM Heidelberg Research Center, Heidelberg, Germany, Jan. 1989.

[Ho72] Holt, R.C.: Some Deadlock Properties in Computer Systems. *ACM Computing Surveys*, Vol. 4, No. 3, Sep. 1972. pp. 179-196.

[HR83] Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, Vol. 15, No. 4, Dec. 1983. pp. 287-317.

[KBCGW87] Kim, W., Banerjee, J., Chou, H.-T., Garza, J.F., Woelk, D.: Composite Objects Support in an Object-Oriented Database System. In: *Proc. of the 2nd Int. Conf. on Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA'87), Orlando, Florida, USA, Oct. 1987.

[KBCGW89] Kim, W., Ballou, N., Chou, H.-T., Garza, J.F., Woelk, D.: Features of the ORION Object-Oriented Database System. In: Kim, W., Lochovsky, F. (Eds.), *Object-Oriented Concepts, Databases, and Applications*, ACM Press, New York, USA, 1989. pp. 251-282. (Chapter 11).

[KBG89] Kim, W., Bertino, E., Garza, J.F.: Composite Objects Revisited. In: *Proc. of the ACM SIGMOD Int. Conf. on the Management of Data*, Portland, Oregon, USA, 1989. pp. 337-347. ACM SIGMOD Record, Vol. 18, No. 2, June 1989.

[KGBW90] Kim, W., Garza, J.F., Ballou, N., Woelk, D.: Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, Mar. 1990. pp. 109-124.

[Ki90] Kim, W.: *Introduction to Object-Oriented Databases*. MIT Press, Cambridge, MA, USA, 1990. (Series in Computer Systems).

[LLOW91] Lamb, C., Landis, G., Orenstein, J., Weinreb, D.: The ObjectStore Database System. *Communications of the ACM*, Vol. 34, No. 10, Oct. 1991. pp. 50-63.

[Ly83] Lynch, N.: Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control. *ACM Transaction on Database Systems*, Vol. 8, No. 4, Dec. 1983. pp. 484-502.

[Ma88] Mattos, N.M.: Abstraction Concepts: The Basis for Data and Knowledge Modeling. In: *Proc. of the 7th Int. Conf. on Entity-Relationship Approach*, Rom, Italy, Nov. 1988. pp. 331-350.

[Ma89] Mattos, N.M.: *An Approach to Knowledge Base Management - Requirements, Knowledge Representation, and Design Issues*. Doctor Thesis, University of Kaiserslautern, Kaiserslautern, Germany, April 1989.

[Ma90] Mattos, N.M.: Performance Measurements and Analyses of Coupling Approaches of Database and Expert Systems and Consequences to their Integration. In: *Proc. of the 1st Workshop on Information Systems and Artificial Intelligence*, Ulm, Germany, Mar. 1990.

[MB90] Mylopoulos, J., Brodie, M.: Knowledge Bases and Databases: Current Trends and Future Directions. In: *Proc. of the Workshop on Artificial Intelligence and Databases*, Ulm, Germany, 1990.

[Mo90] Mohan, C.: ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In: *Proc. of the 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia, Aug. 1990. pp. 392-405.

[Mo92] Mohan, C.: Less Optimism About Optimistic Concurrency Control. In: *Proc. of the 2nd Int. Workshop on RIDE: Transaction and Query Processing*, Tempe, Feb. 1992.

[MSOP86] Maier, D., Stein, J., Otis, A., Purdy, A.: Development of an Object-Oriented DBMS. In: *Proc. of the Int. Conf. on Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA'86), Portland, Oregon, USA, Sep. 1986. pp. 472-482.

[PR83] Peinl, P., Reuter, A.: Empirical Comparison of Database Concurrency Control Schemes. In: *Proc. of the 9th Int. Conf. on Very Large Data Bases*, Florence, Italy, 1983. pp. 97-108.

[PS87] Penney, D.J., Stein, J.: Class Modification in the GemStone Object-Oriented DBMS. In: *Proc. of the Int. Conf. on Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA'87), Orlando, Florida, USA, Oct. 1987. pp. 111-117.

[Re94] Rezende, F.F.: Concurrency Control Techniques and the KBMS Environment: A Critical Analysis. Submitted to: *Journal of Theoretical and Applied Informatics* (RITA), Porto Alegre, Brazil, 1994.

[Re94a] Rezende, F.F.: Evaluating the Suitability of OODBMS Concurrency Control Techniques to the KBMS Environment. Submitted to: *Journal of the Brazilian Computer Society* (JBCS), Campinas, Brazil, 1994.

[SK80] Silberschatz, A., Kedem, Z.: Consistency in Hierarchical Database Systems. *Journal of the ACM*, Vol. 27, No. 1, Jan. 1980. pp. 72-80.

[Ya82] Yannakakis, M.: Freedom from Deadlock of Safe Locking Policies. *SIAM Journal of Computing*, Vol. 11, No. 2, May 1982. pp. 391-408.

[Ya82a] Yannakakis, M.: A Theory of Safe Locking Policies in Database Systems. *Journal of the ACM*, Vol. 29, No. 3, July 1982. pp. 718-740.