A Lock Method for KBMSs Using Abstraction Relationships' Semantics

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

Abstract

Knowledge Base Management Systems (KBMSs) are a growing research area finding applicability in different domains. As a consequence, 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. We show how we obtain serializability of transactions providing many different locking granules, which are based on the semantics of the abstraction relationships. The main benefit of our technique is the high degree of potential concurrency, to be obtained through a logical partitioning of the KB graph and the provision of lock types used for each referenced partition. 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.

1 Introduction

KBMSs are a new product generation which is finding ever more applicability in many different areas. 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, on 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 concurrency control (CC) techniques for KBMSs play a crucial role, because they are among the most important means for allowing large, multi-user KBs to be widespread.

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 database as some serial execution of the same set of transactions, it is correct, because serial executions are 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). This paper is organized as follows. After providing some particular CC issues in KBMSs (Sect. 2), 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 panacea, we finally conclude the paper (Sect. 5).

2 Particular Concurrency Control Issues in KBMSs

2.1 The Abstraction Concepts

Abstractions turned out to be fundamental tools for knowledge organization. They are expressed as relationships between objects, and have as main purpose the organization of such objects in some form. In the following, we provide a brief description of the abstraction concepts. In order to illustrate these concepts, we use as example a restaurant application[1].

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

[1]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].

2.1.1 Classification

Classification is achieved by grouping simple objects (instances) that have common properties into a new composite object (class) for which uniform conditions hold [Ma88]. Classification establishes an *instance-of* (i, for short) relationship between instances and class. Hence, it creates a 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 allows a more complex composite object (superclass) to be defined as a collection of less complex composite ones (subclasses). 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* (sc) 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* and *liquors*. In this case, we can generalize these objects, creating a superclass named, for example, *beverages* (Fig. 1). 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 is also valid for classification, i.e., the instances inherit the properties of their classes, which inherit from their superclasses, and so forth.

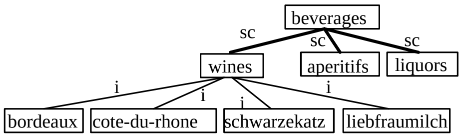

Figure 1: Example of classification and generalization.

2.1.3 Association

There are two types of association, namely element- and set-association [Ma88]. Element-association allows the introduction of an object (set) to describe some properties of a group of objects (elements). It suppresses the details of the element objects whereas emphasizing the properties of the group as a whole. Hence, element-association creates a one-level hierarchy, and between elements and set, an *element-of* (e) 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). On the other hand, set-association builds a more complex set object (superset) in order to represent properties of a group of set objects (subsets). Set-association establishes a *subset-of* (ss) relationship between subsets and superset. In addition, it may be applied recursively, thereby building an n-level hierarchy. For example, we could group the sets referenced in Fig. 2 into a superset representing *wine-origins*.

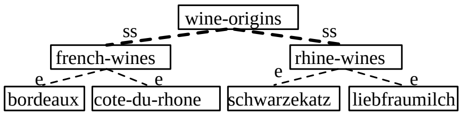

Figure 2: Example of element- and set-association.

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 (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 (component), we are applying the element-aggregation concept, and the relationship between the parts and the component object is called *part-element-of* (*part-of* or p, for short). Element-aggregation builds a one-level hierarchy. In turn, component objects (subcomponents) may be used to build a more complex higher-level object (supercomponent). This characterizes the component-aggregation concept, and between subcomponents and supercomponent, a *subcomponent-of* (c) relationship is established. Since this concept may be applied recursively, it creates an n-level hierarchy. Nevertheless, 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.

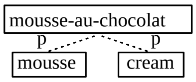

Figure 3: Example of element-aggregation.

2.1.5 An Example Knowledge Base

KBMSs manage complex and structured objects, and 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 KB 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. To illustrate one such a KB graph, we introduce a more detailed example in Fig. 4, complementing the ones we have seen so far. 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[2].

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

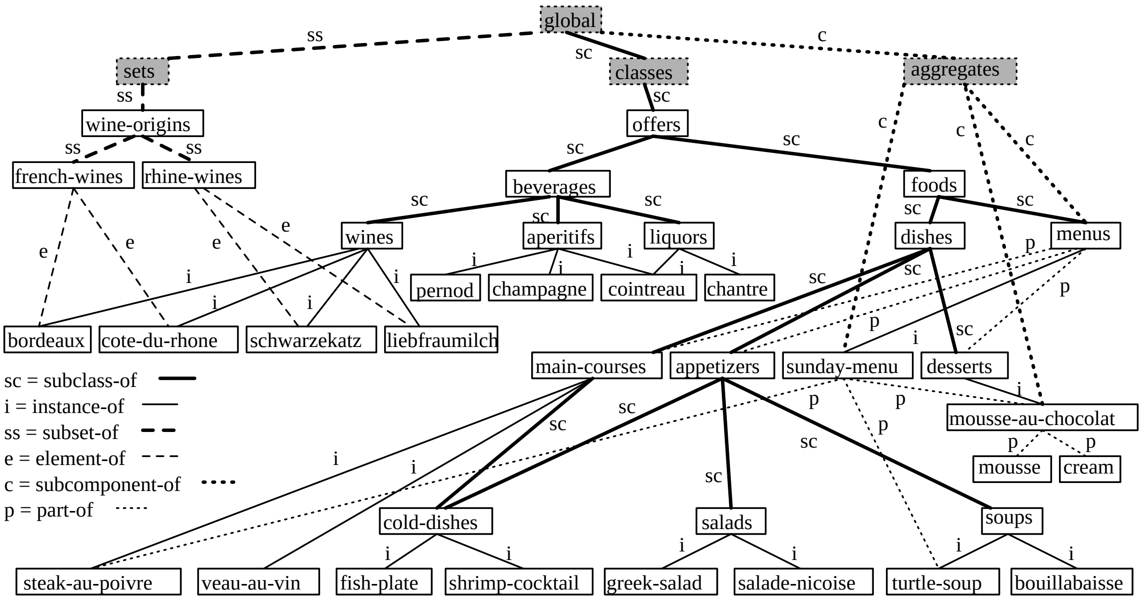

Figure 4: The restaurant knowledge base.

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, FO89]. The main idea behind this use of applications' 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. 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. At the actual point of our work, we did not make any use of 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, 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).

2.3 Operations Types

In KBMSs, there are basically two main types of operations that may be requested against a KB. The first type of operations relates to a tiny granule, normally in the order of an object. Those represent direct operations on objects, and mostly refer to reads and writes of an object, its slots, methods, and aspects, not influencing the subhierarchies. On the other hand, the second type of operations handles a coarse granule, and refer mostly to an object and its subhierarchies, through navigational accesses to the objects one at a time. Particularly, the operations involving inheritance are a good example of this second type. Therefore, in summary, we can characterize the accesses to a KB as referring either to an object, or to a set of objects related through any abstraction relationships. Finally, a CC technique for KBMSs should pay attention to such types of operations, and provide adequate lock modes to cope well with them.

3 Related Work

To the best of our knowledge, there is only one CC protocol tailored for KBMSs already published, namely the Dynamic Directed Graph (DDG) policy of Chaudhri et al. [CHM92]. Due to space limitations, we will not provide an exhaustive discussion about this protocol here[3]. 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. 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. Third, phantoms are not taken into consideration.

Now, due to the lack of publications in this area, let us analyze some CC protocols of a related area, namely Object-Oriented DBMSs (OODBMSs). Again, we will not extensively discuss these protocols here[4]. ORION [Ki90] extended the *Granular Locks Protocol* (GLP, for short) of Gray et al. [GLPT76] and by this way, it provides implicit locks [GK88]. 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.

The main benefit of CC in O_2 [BDK92] 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 of a component or element of an object must be made in mutual exclusion, although not necessarily

[3] The reader is asked to see [Re94] for a detailed discussion and critical analysis of this protocol.

[4] See [Re94a] for a more detailed discussion about OODBMS CC techniques and their behavior in the KBMS environment.

conflicting. In addition, no kind of implicit locks is provided for subclasses of a class.

The OODBMS GemStone [BOS91] protects its concurrent transactions using a combination of optimistic and pessimistic CC techniques. First of all, optimistic methods may show very poor performance due to, among other things, the possibly very 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 large number of locks. Moreover, its limited number of lock types restricts the parallelism, and it is unaware about the semantics of the relationships between objects.

4 Locks Using Abstraction Relationships' Semantics

4.1 Generalization of Granular Locks

Granular locks were introduced by Gray et al. [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 isolation. 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 hierarchy or a Directed Acyclic Graph (DAG) (Fig. 5) [GLPT76].

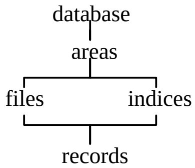

Figure 5: A non-hierarchical lock graph for granular locks.

4.2 Lock Modes and Compatibilities

As already discussed in Sect. 2.3, we have basically two main types of operations in a KB. Accordingly, our protocol supports two main types of lock modes, the first one related to a single object, and the second one to a set of them.

4.2.1 Conventional Locks

The first type of lock modes we have gives respect to an object as a closed unit. These lock modes are the conventional R (Read) and W (Write) locks (thus the name *conventional locks*), and their semantics are presented in Fig. 6.

R	gives shared access to the requested object.
W	gives exclusive access to the requested object.

Figure 6: Conventional locks' semantics.

In turn, the compatibility between these lock modes is dictated by the matrix sketched in Fig. 7.

		Granted Mode	
		R	W
Requested Mode	R	yes	no
	W	no	no

Figure 7: Compatibility matrix for the conventional locks.

4.2.2 Typed Locks

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, named the *classification* (which includes also generalization), *association*, and *aggregation* hierarchies. 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. Exemplifying, suppose there are three transactions running in our KB (Fig. 4). 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. 8.

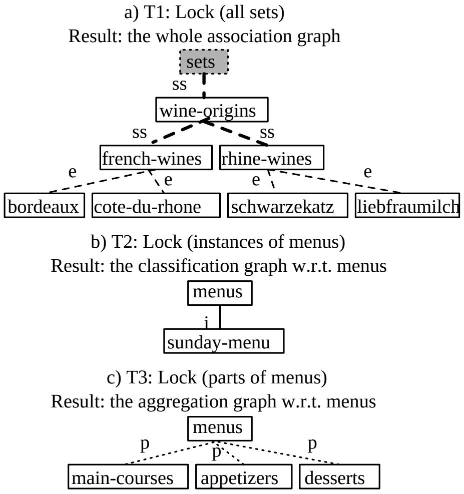

Figure 8: Different transactions' viewpoints.

Thus, for the second type of operations in a KB, we have created three distinct sets of lock types, which are based on the logical partitioning of the graph previously introduced. 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. In contrast, the GLP refers to a single structure roughly incorporating classification relationships. 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 Fig. 9, we present the semantics of each one of them.

Compatibility of locks on the same sets of objects

With respect to the compatibility of the above mentioned lock types, we have two distinct situations to cope with. 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.

IRc	gives intention shared access to the requested object and allows the requester to explicitly lock both direct subclasses of this object in Rc, IRc, or R mode, and direct instances in Rc or R mode.
IWc	gives intention exclusive access to the requested object and allows the requester to explicitly lock both direct subclasses of this object in Wc, RIWc, Rc, IWc, IRc, W or R mode, and direct instances in Wc, Rc, W or R mode.
Rc	gives shared access to the requested object and implicitly to all direct and indirect subclasses and instances of this object.
RIWc	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 Wc, RIWc, Rc, IWc, IRc, W or R mode, and direct instances in Wc, Rc, W or R mode).
Wc	gives exclusive access to the requested object and implicitly to all direct and indirect subclasses and instances of this object.
IRs	gives intention shared access to the requested object and allows the requester to explicitly lock both direct subsets of this object in Rs, IRs, or R mode, and direct elements in Rs or R mode.
IWs	gives intention exclusive access to the requested object and allows the requester to explicitly lock both direct subsets of this object in Ws, RIWs, Rs, IWs, IRs, W or R mode, and direct elements in Ws, Rs, W or R mode.
Rs	gives shared access to the requested object and implicitly to all direct and indirect subsets and elements of this object.
RIWs	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 Ws, RIWs, Rs, IWs, IRs, W or R mode, and direct elements in Ws, Rs, W or R mode).
Ws	gives exclusive access to the requested object and implicitly to all direct and indirect subsets and elements of this object.
IRa	gives intention shared access to the requested object and allows the requester to explicitly lock both direct subcomponents of this object in Ra, IRa, or R mode, and direct parts in Ra or R mode.
IWa	gives intention exclusive access to the requested object and allows the requester to explicitly lock both direct subcomponents of this object in Wa, RIWa, Ra, IWa, IRa, W or R mode, and direct parts in Wa, Ra, W or R mode.
Ra	gives shared access to the requested object and implicitly to all direct and indirect subcomponents and parts of this object.
RIWa	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 Wa, RIWa, Ra, IWa, IRa, W or R mode, and direct parts in Wa, Ra, W or R mode).
Wa	gives exclusive access to the requested object and implicitly to all direct and indirect subcomponents and parts of this object.

Figure 9: Typed locks' semantics.

S_type, or A_type vs. A_type), then the compatibility matrix to be followed is identical to the GLP's one known from the literature [GLPT76, Gr78] (Fig. 10).

Granted Mode [c | s | a]

Requested Mode [c | s | a]	IR	IW	R	RIW	W
IR	yes	yes	yes	yes	no
IW	yes	yes	no	no	no
R	yes	no	yes	no	no
RIW	yes	no	no	no	no
W	no	no	no	no	no

Figure 10: Compatibility matrix for locks of the same type.

Compatibility of locks on distinct sets of objects

The second situation with respect to the compatibility of the typed locks is the one where both locks refer to distinct sets of objects (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 between both granted and requested mode is dictated by the matrix in Fig. 11. There, 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.

Granted Mode [c | s | a]

Requested Mode [s or a | c or a | c or s]	IR	IW	R	RIW	W
IR	yes	yes	yes	yes	yes
IW	yes	yes	yes	yes	yes
R	yes	yes	yes	yes	no
RIW	yes	yes	yes	yes	no
W	yes	yes	no	no	no

Figure 11: Compatibility matrix for locks of distinct types.

Let us provide an example (Fig. 12) to justify this compatibility matrix. Suppose we have two transactions operating in our KB (Fig. 4). T1 wants to modify the object *sunday-menu*, instance of *menus*. Hence, T1 signs its intention to write the object *sunday-menu* in its class *menus* with an IW_c lock. Thereafter, T1 is allowed to request a W_c lock on the instance of *menus*, and so it does. In turn, T2 wants to write the object *menus* itself and all parts of it. To do that, T2 must request a W_a lock on *menus* from the lock manager. Now, the compatibility of both locks on *menus* must be checked. In turn, we can notice that albeit T1 did sign an intention write in *menus*, the IW_c lock mode allows it to request write locks only on the instances of *menus*, and **not** on its parts. On the other hand, the write lock requested by T2 in *menus* does not embody any of the objects to be accessed by T1, since it (W_a) implies implicit write locks only on the parts of *menus*, and **not** on its instances. Consequently, both locks are compatible, and may be simultaneously granted. 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. 11 may be shown to be compatible according to the given table. We leave this task for the reader.

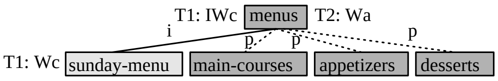

Figure 12: Concurrent transactions operating on different sets of objects.

Compatibility of conventional and typed locks

The last compatibility matrix is the one for conventional and typed locks. Fig. 13 presents the compatibility between the conventional (R and W) locks and the typed (either C_type, or S_type, or A_type) locks.

	Granted Mode [c | s | a]					
		IR	IW	R	RIW	W
Requested Mode	R	yes	yes	yes	yes	no
	W	yes	yes	no	no	no

Figure 13: Compatibility matrix for conventional and typed locks.

We illustrate the meaning of the compatibility matrix of Fig. 13 with an example (Fig. 14). Suppose, using our restaurant KB (Fig. 4), that T1 wants to modify a slot of the object *offers*. In addition, suppose that this slot is used to represent a property of the own object, and not of its subclasses. In other words, this slot should not be inherited by its subclasses, like for example a slot *comment*, used to describe some particularity of *offers*. T1 can modify this object requesting a conventional W lock on it. In particular, the deletion of *offers* by T1 would imply modifications on other objects not covered by such a W lock, and would be therefore not allowed. On the other side, T2 wants to modify the object *liquors* and its instances. It must then sign in the objects *offers* and *beverages* its intention to write some descendants of them. It starts then requesting an IW_c lock on *offers*. Due to the fact that the conventional W lock on *offers* hold by T1 refers to nothing but this object, the IW_c requested by T2 is compatible with it, and thus granted. Thereafter, T2 acquires IW_c on *beverages*, and finally W_c on *liquors*. At last, any operations of T1 touching the subclasses or instances of *offers* would require a W_c lock on it and, as a consequence, implicit locking and the detection of conflicts by means of the intention locks.

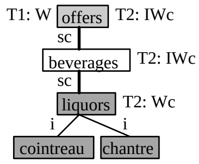

Figure 14: Transactions operating on an isolated object and on a set of objects.

With respect to this example, there is an important point to be considered. Suppose our protocol would not provide the conventional locks. In this case, T1 would have to require a W_c lock on *offers*. On one hand, T1 might have to wait too long for receiving this lock. It would be granted if, and only if, no other transaction is accessing any class or instance of the KB, since *offers* is the only root of the whole classification graph. On the other hand, once granted, T1 would impede any other transaction from accessing any other class and instance of the KB, and of course unnecessarily, because T1 in fact wants to modify only the object *offers*, and nothing else. Therefore, this example ratifies the importance of the conventional locks for allowing the execution of operations on isolated objects, and as a complement for the typed locks.

4.3 Accessing Implicitly Locked Objects

As a matter of fact, multiple abstraction relations 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. Actually, an interference arises whenever an object with two or more parents is implicitly locked by one of them. The implicit lock on a child object is only visible if it is accessed through a specific path of the graph. To illustrate this problem, let us refer to Fig. 15. There, both T1 and T2 required an IW_c lock on *beverages*, and were granted. Thereafter, T1 followed the path to *aperitifs* and locked it in W_c mode. Then, it received automatically write access rights 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 (*cointreau* and *chantre*) too. Here, T1 and T2 may get into troubles with one another, and for example a *lost update* may happen in the object *cointreau*. The problem is that none of those 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.

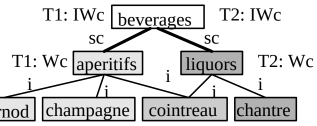

Figure 15: The problem with implicit locks in graphs.

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. Finally, we could follow basically five possible approaches. Let us discuss all of them separately.

1. Lock all referenced objects

The first and most simple approach is to explicitly lock all referenced objects. In the example of Fig. 15, if either T1 or T2 locks all schemas explicitly, the interference in *cointreau* is detected. This practically vanishes the semantics of implicit locks, but it solves the problem[5]. Nevertheless, this method leads to a great overhead, since many locks are required.

2. Search for conflicts

The second approach is, before accessing any implicitly locked objects with multiple parents (i.e., the ones where potential conflicts may arise), to climb up the structure in order to find out possible conflicts. In this case, a conflict is detected if any of those objects is already implicitly locked by any other ascendant in a conflicting mode. In the above example (Fig. 15), 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, but it requires testing locks. Of course, it is very expensive if an object has multiple parents, which in turn have multiple 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. In the current example (Fig. 15), 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.

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 those with more than one parent[6]. In the

[5] This alternative is followed by ORION for its class lattices.

[6] This alternative was pointed out by Garza and Kim [GK88] for

current example (Fig. 15), this means that when T1 sets a W_c lock on *aperitifs*, it needs also to set the same lock on *cointreau*, the only descendant of *aperitifs* with multiple parents. When following the same proceeding, T2 detects the conflict and must then wait until T1 terminates. This approach 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 those descendants with multiple parents. In addition, it is also 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. 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 object with multiple parents, it must verify whether this object is already locked in a conflicting mode by another transaction or not. If so, it must wait until this lock is released. If not, it sets an explicit lock on this object, signalling that it has accessed it. This lock acts like a tag in the object indicating that it has been already accessed via another parent of it. The main difference between this alternative and the previous one is that a transaction needs to explicitly lock only those descendants with multiple parents which it actually accesses, leaving the others for the concurrent access by other transactions. In the current example (Fig. 15), 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 object by T1 is already released, for example because T1 has already modified this object 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 descendants with multiple parents 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. Apparently, this approach also leads to deadlocks (to be discussed in Sect. 4.5.1). 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 our protocol.

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 object with multiple parents are already explicitly locked by a transaction, this transaction does not need to set an explicit lock on this object when accessing it. In fact, all paths reaching this object should already be covered, and therefore, potential conflicts would be already detected. However, this proceeding seems too difficult to be realized and too expensive.

the class lattices in ORION, implemented for test purposes, but discarded.

4.4 The Locking Rules

Having presented the general lines of our protocol, we are finally able to expose its complete rules to be followed by transactions when requesting locks on objects in a KB (Fig. 16).

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.

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.

There is an important point appearing in the third rule to be considered. Whereas the GLP requires that *all paths* from a node to the root must be covered by the corresponding intention locks for writing [GLPT76], our protocol relaxes this requirement, advocating that just *one path* to the root object is enough for lock conflict resolution. The reason for it is that our protocol, as part of our lazy evaluation strategy discussed in Sect. 4.3, takes special care when dealing with implicitly locked objects, requiring also explicit locks on such objects which have more than one parent. This requirement is coped with by the rules 4, 5, and 6. Therefore, since our protocol does not rely solely on the intention locks for conflict resolution, we do not need to set intention locks in all the paths to the root, but just in only one (any) of them.

The fourth rule states 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. Thereafter, it explicitly requires locks on the descendants with multiple parents[7]. The fifth and sixth rules have a similar meaning, but for the typed RIW and W locks, respectively.

Before passing on to the explanation of the following rules, we provide an example (Fig. 17), 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 notices that this object has another parent, and, as stated by the rule 6, it requests a W_c lock on this object, and is

[7]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 have multiple parents, no matter if the parents are the same object.

1	The root object can be locked directly in any mode.
2	Before requesting an IRc | IRs | IRa mode lock on a non-root object, the requester must hold a path to the root in IRc or IWc | IRs or IWs | IRa or IWa mode.
3	Before requesting an IWc | IWs | IWa mode lock on a non-root object, the requester must hold a path to the root in IWc or RIWc | IWs or RIWs | IWa or RIWa mode.
4	Before requesting an Rc | Rs | Ra mode lock on a non-root object, the requester must hold a path to the root in IRc or IWc | IRs or IWs | IRa or IWa mode. In addition, before accessing any implicitly locked descendant object with multiple parents, the requester must set an Rc | Rs | Ra lock on it.
5	Before requesting an RIWc | RIWs | RIWa mode lock on a non-root object, the requester must hold a path to the root in IWc or RIWc | IWs or RIWs | IWa or RIWa mode. In addition, before accessing any implicitly locked descendant object with multiple parents, the requester must set either a) an Rc | Rs | Ra lock on it, if it is a leaf object, or b) an RIWc | RIWs | RIWa lock, if it is a non-leaf object.
6	Before requesting a Wc | Ws | Wa mode lock on a non-root object, the requester must hold a path to the root in IWc or RIWc | IWs or RIWs | IWa or RIWa mode. In addition, before accessing any implicitly locked descendant object with multiple parents, the requester must set a Wc | Ws | Wa lock on it.
7	An R mode lock can be directly set by the requester on a non-root object with multiple parents, but if it has only one parent, the requester must hold the parent in IRc | IRs | IRa, IWc | IWs | IWa, Rc | Rs | Ra, RIWc | RIWs | RIWa, Wc | Ws | Wa, R or W mode.
8	A W mode lock can be directly set by the requester on a non-root object with multiple parents, but if it has only one parent, the requester must hold the parent in IWc | IWs | IWa, RIWc | RIWs | RIWa, Wc | Ws | Wa, or W mode.
9	Before inserting an object, the requester must hold the parent in IWc | IWs | IWa, RIWc | RIWs | RIWa, Wc | Ws | Wa, or W mode.
10	Before deleting an object, the requester must hold it in Wc | Ws | Wa, or W mode.
11	Before inserting an edge, the requester must hold the descendant in Wc | Ws | Wa, or W mode, and the parent in IWc | IWs | IWa, RIWc | RIWs | RIWa, Wc | Ws | Wa mode.
12	Before deleting an edge, the requester must hold the descendant in Wc | Ws | Wa, or W mode.
13	Release all locks as soon as the transaction terminates (either commits or aborts).

Figure 16: Locking rules.

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.

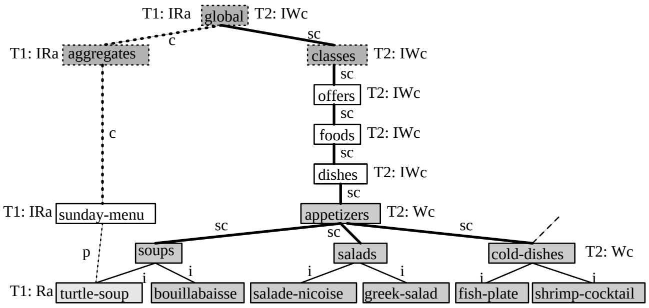

Figure 17: Avoiding conflicts with implicitly locked objects.

The rules for requesting conventional locks (7-8)

Proceeding with the rules, the seventh rule establishes that a conventional read lock may be granted immediately to the requester when the object in question has multiple parents. The reason for that comes from the rules 4, 5, and 6, which state that those objects must be always locked explicitly. Hence, if the object is free from locks, it means that no transaction is accessing it and the conventional read lock may be granted. On the other hand, if some transaction does access it, any conflict may be found directly in the object because it is certainly locked, and so we do not need to traverse a path to the root. Referring to the current example above (Fig. 17), transaction T1 could request a conventional R lock on the object *turtle-soup*, instead of an R_a lock. When requesting this lock, T1 may be immediately granted without further requirements, because the object *turtle-soup* has two parents and is currently free from locks. In turn, as soon as transaction T2 tries to access this object, it realizes that it has two parents, and explicitly requests a W_a lock on it, thereby detecting the conflict. However, if the object has only one parent, one does not know whether any other transaction has already implicitly locked and accessed this object in a conflicting mode, and hence one must traverse a path to the root with intention locks looking for possible conflicts. The type of those intention locks must be requested in accordance to the relationship between the object and its unique parent. Finally, conventional write locks are dealt with in a similar way by the rule 8.

The rules for insertion/deletion of objects (9-10)

The ninth rule copes with the insertion of objects in the KB. Before explaining it, 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 many relationships, our protocol treats such cases as being an insertion of an object with one relationship, followed by as many insertions of edges (governed by rule 11) as asserted by the user. Hence, the ninth 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. 18 provides an example of the lock requests needed for inserting an object. Suppose 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*. 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. Particularly, this operation is susceptible to phantoms (discussed in Sect. 4.5.2).

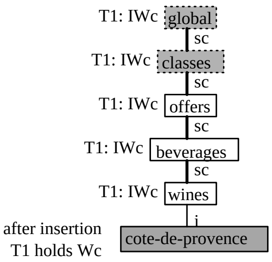

Figure 18: Locks for the insertion of an object.

In turn, the tenth 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 many parents is to be deleted, our protocol treats such cases as many deletions of edges (governed by rule 12) as necessary, until the object has only one parent. Finally, the type of such locks are dictated by the abstraction relationship in question.

The rules for insertion/deletion of edges (11-12)

The eleventh 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 relation being inserted. Fig. 19 illustrates the use of this rule, complementing the last example. Suppose T1 wants to connect *cote-de-provence* as an element of *french-wines*. Following the rule 11, 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 acquires a W_c lock on it (from 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 12. In such cases, the transaction does know the current roles of the objects, and by this way the path it must traverse for requesting locks.

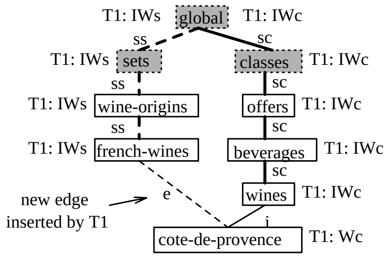

Figure 19: Locks for the insertion of an edge.

The commit rule (13)

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

4.5 Final Considerations

4.5.1 Deadlocks

Our protocol is susceptible to *deadlocks*. However, it suffers from a kind of deadlock which does not happen in the GLP, namely, deadlocks with, additionally, implicit locks. There are a lot of strategies to detect deadlocks. *Timeout* and *waits-for-graph* [Ho72] are the most popular ones. 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.

4.5.2 Phantoms

Another important point is that granular locks provide physical locks, and being so we have also problems with the so-called *phantoms* in our protocol. 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 our 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. 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.

4.5.3 Lock Conversion

In our 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. 20 shows the lock conversion tables for conventional and 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

Old Mode		R	W
	R	R	W
	W	W	W

a) Conventional locks

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

b) Typed locks

Figure 20: Lock conversion tables.

4.5.4 Lock Escalation

Lock escalation is also taken into consideration by our 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, our 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 our protocol must cope with. As stated by the locking rules 4, 5, and 6 of our protocol, the transaction must request also explicit locks on implicitly locked objects with multiple parents. Due to this requirement, a lock escalation in our protocol alleviates the transaction from requesting locks just on those descendants with a single parent, but the same is not true for the objects with multiple parents. We believe this aspect will not prejudice too much the performance of lock escalation in our protocol. Notwithstanding, if our protocol is likely to be faced with situations like, for example, a class has thousands of instances and all of them have any other parent, then our protocol 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 possible 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. 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

We have presented a CC technique tailored for KBMSs. The most important point of our technique is the partition of the KB graph into many logical ones, and the appliance of conventional and granular locks to each one of these partitions, providing 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. By means of this observation, we can really obtain a high degree of concurrency, with a full exploitation of all inherent parallelism in a knowledge representation approach.

In summary, our protocol has four main characteristics. First, it offers different granules of locks, varying from a single object (handled by the conventional locks) to different sets of objects (typed 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 relations to objects, by means of the requirement of explicitly locking objects with multiple parents, 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 are going to use the KBMS prototype KRISYS [Ma89] as a practical vehicle for the implementation.

References

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

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

[BOS91] Butterworth, P., Otis, A., Stein, J.: The GemStone Object Database Management System. *Communications of the ACM*, 34(10), Oct. 1991.

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

[CHM92] Chaudhri, V.K., Hadzilacos, V., Mylopoulos, J.: Concurrency Control for Knowledge Bases. In: Proc. *3rd KRR*, Cambridge, USA, 1992.

[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*, 19(11), Nov. 1976.

[FO89] Farrag, A.A., Ozsu, M.T.: Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM TODS*, 14(4), Dec. 1989.

[Ga83] Garcia-Molina, H.: Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM TODS*, 8(2), June 1983.

[GK88] Garza, J.F., Kim, W.: Transaction Management in an OODBS. In: Proc. *ACM SIGMOD Int. Conf. on the Management of Data*, Chicago, USA, 1988.

[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. *IFIP Conf. on Modeling in DBMS*, Freudenstadt, Germany, 1976.

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

[Gr78] Gray, J.N.: Notes on Database Operating Systems. In: *Operating Systems: An Advanced Course*, Springer, Berlin, 1978.

[Hä84] Härder, T.: Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2), 1984.

[Ho72] Holt, R.C.: Some Deadlock Properties in Computer Systems. *ACM Computing Surveys*, 4(3), Sep. 1972.

[HR83] Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4), Dec. 1983.

[Ki90] Kim, W.: *Introduction to Object-Oriented Databases*. MIT Press, USA, 1990.

[Ly83] Lynch, N.: Multilevel Atomicity - A New Correctness Criterion for Database Concurrency Control. *ACM TODS*, 8(4), Dec. 1983.

[Ma88] Mattos, N.M.: Abstraction Concepts: The Basis for Data and Knowledge Modeling. In: Proc. *7th Int. Conf. on E-R Approach*, Rom, Italy, 1988.

[Ma89] Mattos, N.M.: *An Approach to Knowledge Base Management*. Ph.D. Thesis, University of Kaiserslautern, Germany, April 1989.

[MB90] Mylopoulos, J., Brodie, M.: Knowledge Bases and Databases: Current Trends and Future Directions. In: Proc. *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. *16th VLDB*, Brisbane, Australia, 1990.

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

[PR83] Peinl, P., Reuter, A.: Empirical Comparison of Database Concurrency Control Schemes. In: Proc. *9th VLDB*, Florence, Italy, 1983.

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