In: RITA - Journal for Theoretical and Applied Computer Science, Vol. 2, No. 1, Brazil, Jan. 1995. pp. 37-76.

Concurrency Control Techniques and the KBMS Environment:

A Critical Analysis

Fernando de Ferreira Rezende

Department of Computer Science - University of Kaiserslautern

P.O.Box 3049 - 67653 Kaiserslautern - Germany

e-mail: rezende@informatik.uni-kl.de

June 1993

Abstract

Knowledge Base Management Systems (KBMSs) are a new product generation with recognized applicability in several different areas, like medicine, geology, engineering design, robotics, etc. As long as this applicability is growing, the necessity for knowledge sharing is also becoming a crucial point to be taken into account by those systems. Thus, knowledge sharing and connectivity are going to be the key features of the intelligent information systems of the future. This means that many users should be able to access the KBMSs simultaneously. Nevertheless, arbitrary concurrent access to a resource can lead to many inconsistencies, and undesirable behavior. The main purpose of this paper is to identify the essential problems to be addressed by concurrency control techniques. Moreover, some of the well-known concurrency control techniques are investigated, and a critical analysis is made about their (in)adequacies, if they were to be directly applied to the KBMS environment.

Contents

In: RITA - Journal for Theoretical and Applied Computer Science, Vol. 2, No. 1, Brazil, Jan. 1995**.** pp. 37-76.

1. Introduction

Concurrency control techniques are well-known in the database community, and there is a vast amount of literature in this area. The main problem addressed by concurrency control is the coordination of actions that operate in parallel, access shared data, and doing so potentially interfere with each other. This interference can lead the database to an undesirable, inconsistent state, if it is not controlled at all. Concurrency control problems arise in many different areas. Particularly, in this work we examine the problems that arise when large, multi-user knowledge bases (KBs) are used.

After motivating, in Sect. 2, the necessity for concurrency control in Knowledge Base Management Systems (KBMSs), we present an overview of the problems that should be addressed by concurrency control (Sect. 3). Thereafter, we provide some important issues which should be taken into account when talking about concurrency control methods for KBMSs (Sect. 4). By doing so, we are then able to analyze many of the concurrency control techniques and their (in)adequacies to the KBMS environment (Sect. 5). After this analysis, we then conclude the paper in Sect. 6, pointing out some important aspects to be considered by a concurrency control mechanism tailored for KBMSs, and finally giving some directions for future work.

Note that we are most interested in performing a critical analysis of the actual concurrency control techniques, if they were to be directly applied to KBMSs. In other words, we point out problems, leaving the proposal of a possible panacea to a later opportunity.

2. The Need for Multi-User KBMSs

KBMSs are a new product generation, which arrived due to the addition of new functionalities to databases. Taking a different point of view, the incorporation of database-like features in expert systems also helped in the development of such systems. This is one of the two approaches to the development of KBMSs cited by [BM86], namely, the *evolutionary approach*, which starts with an existing technology and moves toward a KBMS by extending its functionality. The other one, the *revolutionary approach*, starts with a knowledge representation language and then builds around it appropriate KBMS facilities. Anyway, not only the incorporation of functionalities to the conventional database systems, or the addition of facilities to some knowledge representation language, will be needed to the success of KBMSs. It is also necessary to adapt such systems to real-life production environments. This is what the research in the direction of KBMSs is trying to achieve [MB90].

As already happened with Data Base Management Systems (DBMSs), where data sharing and interleaved execution of an arbitrary number of transactions have become fundamental prerequisites for each successful system, so it will happen with KBMSs. Knowledge sharing and connectivity are going to be the key features of the intelligent information systems of the future. Moreover, it will be imperative for efficiency reasons [Ch91].

This means that the KBs should be accessible not only to one user at a time, what is neither viable (economical reasons) nor desirable (restrict access). It would be very inefficient to obligate users to access valuable resources in mutual exclusion. If some system can process just one transaction at a time, it will spend a great deal of unnecessary time waiting for the disk accesses to complete [Gr78]. Instead, what is really expected is to have a single KB being shared by multiple users. So, KBMSs should receive queries and updates in an interleaved fashion and control their concurrent execution against the KB. Thus, multiple transactions should be able to run at the same time for better performance of a system. By this way, the deployment of computing resources (both CPU cycles and space) is optimized [CHM92].

As a practical example, suppose we have a KB, like the one sketched in Fig. 1 (we only used a generalization/classification hierarchy in order to simplify our examples, although the expansion of this KB to represent aggregation and/or association is trivial). This KB contains information about different means of transportation. Thus, it is just impracticable to require that a user desiring to work on some vehicle may not concurrently access this KB while another one is working on, for example, some aircraft. Moreover, an automobile is composed of several parts (not represented in the illustration), and it is normally required to have many users working on the different parts of it, obviously at the same time. The applicability of some KB system is severely constrained if its access is limited to only one user at a time.

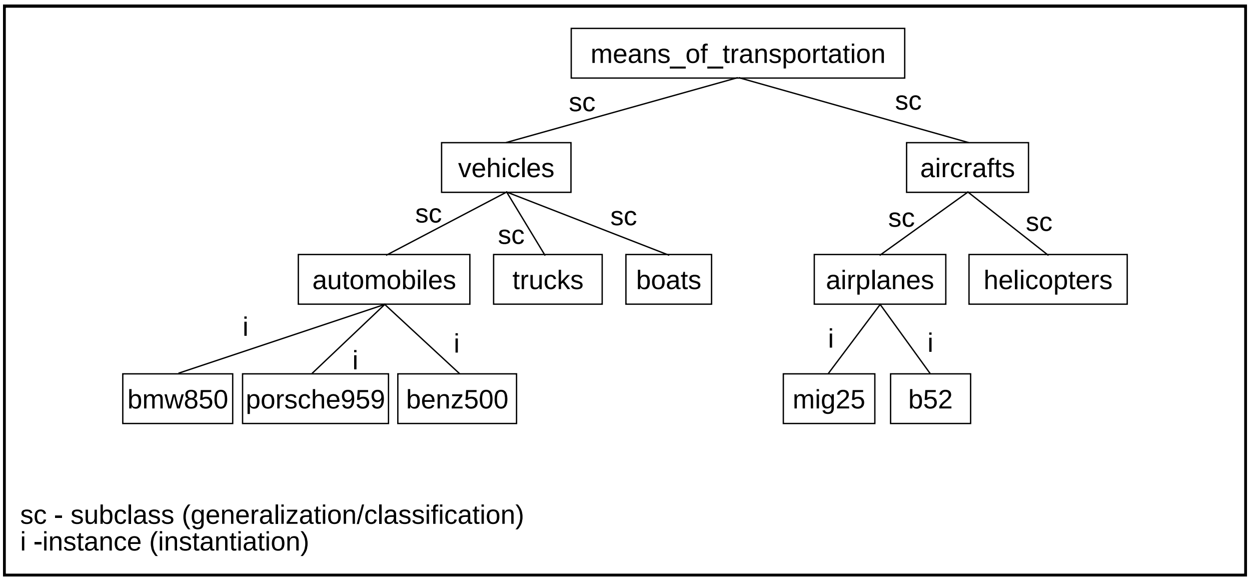

Figure 1: An Example Knowledge Base [1].

Therefore, it is just unacceptable to require that several users working on the same KB should access it in mutual exclusion. On the contrary, knowledge should be shared among several users. To allow such situations of knowledge sharing, concurrency control mechanisms must be developed and tailored to the KBMS environment.

3. Concurrency Control Problems

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. To avoid this interference has been called the concurrency control problem.

[1] Although a little modified, this KB was originally inspired in the one presented in [Ma88].

In the context of database systems, this problem has been studied extensively [2]. To better understand such a problem, let us look at some examples.

Using the example KB sketched in Fig. 1, let us think of some situations with multiple users accessing this KB, and let *transaction* refer to the execution of a user program on a KB. Suppose a user, say Mary, reads and writes the object bmw850 (hypothetically represented through the operations read (bmw850) and write (bmw850)). After that, and before Mary finishes her transaction, another user, say John, reads the same object (now translated to read (bmw850)). This situation is sketched in the following, where *begin*, *commit*, and *abort* are transaction boundaries (begin stands for the start of a transaction, commit for its successful end, and abort for its abnormal end):

- Problem 1:

John	Mary
---	begin
begin	read (bmw850)
---	write (bmw850)
read (bmw850)	---
---	---
commit	---
---	abort

This situation illustrates the first concurrency control problem. John reads a value of the object bmw850 which in fact does not exist, because the transaction that wrote that value aborted, and so the effects of it are obliterated from the knowledge reservoir. This problem is commonly known as *Dirty Read* or *Inconsistent Retrieval* [3]. This anomaly occurs whenever a transaction reads an object after another transaction has updated it, and commits before the end (commit/abort) of the updating transaction. The retrieval transaction cannot be sure about the consistency of the objects it read. This occurs due to the dependency between the operations (write -> read), which conflict with each other. Two operations are said to conflict if, generally speaking [4], the computational effect of their execution depends on the order in which they are processed [BHG87].

Let us continue to find more problems of concurrency, without control. Suppose now that John gets the object porsche959 twice. After John read it once, and before reading it again and finishing his work, Mary reads and then writes the same object. The following illustrates this situation:

- Problem 2:

John	Mary
begin	---
read (porsche959)	begin
---	read (porsche959)
---	write (porsche959)
read (porsche959)	---

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

[3] [BHG87] say that the above execution is not *recoverable*, because the commit of a transaction (John's transaction, in our example) does not follow the commit of every transaction (Mary's transaction) from which it read.

[4] The term *general* is normally used because there are some operations, for example two writes, that might not conflict, depending on the semantics of such operations, e.g., two writes that write the same value, or two incremental operations, etc.

---	commit
commit	---

This leads to the second problem of arbitrary interleaving of transactions. The value to be returned by the second read performed by John will not be the same that he has read previously, but the one written by Mary. This problem is known as *Unrepeatable Read* [5]. This type of interference occurs whenever a retrieval transaction reads an object before another transaction updates it, and reads the object again after the other transaction has updated it. The two read operations return different values for the object. This happens due to another kind of dependency between the operations (read -> write), which also conflict with each other.

To find another problem, suppose now that John reads the object benz500. After that, Mary also reads the same object. But before committing their transactions, both write the object. Let us take a look at it schematically:

- Problem 3:

John	Mary
begin	---
read (benz500)	begin
---	read (benz500)
write (benz500)	---
---	write (benz500)
commit	---
---	commit

This is the third (although not the last) problem of interleaving transactions, and not controlling their consequences on each other. In this case, the write operation performed by John is superposed by Mary's write, i.e., John lost his update. Due to this peculiarity, this problem is known as *Lost Update* [6]. This phenomenon occurs whenever two transactions, while attempting to modify an object, both read the old value of the object before either of them has written its new value. This is also another kind of dependency between conflicting operations (write -> write).

There are also other problems of interleaved execution of transactions. The *Phantom* problem is one of them, but we will delay the commentaries about these problems for appropriate subsequent sections. Nevertheless, there is also one way to avoid these interference problems: not allowing the transactions to interleave at all. If all transactions execute serially, i.e., one after the other, no problems may arise [7]. Thus, serial executions are correct because each transaction individually is correct and if they execute serially, they cannot interfere with each other.

Taking this assumption that a serial execution of a set of transactions always produces correct results, we could think that for all above mentioned problems to be resolved, we only need to obligate the system to

[5] [BHG87] say that the above execution does not *avoid cascading aborts*, because a transaction (John's transaction, in our example) does not read only those values that were written by committed transactions (Mary's transaction), so an unfinished transaction affected other transactions.

[6] [BHG87] say that the above execution is not *strict*, because both reads and writes of a transaction on an object (the write performed by Mary's transaction, in our example) are not delayed until all transactions that have previously written the object (the write by John's transaction) are committed or aborted.

[7] Discarding the possibility that the transactions themselves might be wrong, because if they were, we could also find a lot of problems with serial executions. Fortunately, the concurrency control theory discards this possibility and assumes that the transactions are always correctly written, and the correctness of their code is the programmers' responsibility.

process the transactions serially, one at a time. This could be a practical solution for very small systems. But if we imagine some system with thousands of objects and hundreds of users, this solution is just impractical. Such a system would make poor use of its resources, and would be also too inefficient. It is just impossible to convince some user that he must wait to execute his program until all other users have completed their work.

Resolving the concurrency control problems, and, of course, allowing interleaved execution of transactions, the *Serializability Theory* was developed. This theory was first presented in [GLPT76], and posteriorly refined by many other publications. It states that if an execution produces the same output and has the same effect on the database as some serial execution of the same transactions, it is correct, because serial executions are also correct [BHG87]. Such executions are called *serializable executions* (we will not prove this theory, since there are already a lot of proofs in the literature, and that is beyond our goal; see [BHG87] for a convincing one).

The transaction executions shown as examples above, which illustrated the dirty read, unrepeatable read, and lost update problems, are not serializable, because neither of them produces the same output, nor has the same effect on the KB, as some serial executions of the same set of transactions.

There are a lot of techniques for controlling concurrent executions of transactions, the so-called concurrency control techniques, which obtain the effect of serializable, correct executions, avoiding thus the many above mentioned problems. But before analyzing some of these techniques, let us take an overlook at the KBMS environment and its particularities. This is the topic of the next section.

4. Concurrency Control Issues for KBMSs

Some of the issues arising when multi-user KBMSs are built have already been identified in the studies of databases, whereas others arise due to the new features and application domains supported by KBMSs. Unfortunately, there is no commonly agreed architecture for KBMSs. Some initial ideas on KBMS architectures have been proposed in [BM86]. [Ma91] presents a concrete KBMS architecture, implemented in a prototypical KBMS at the University of Kaiserslautern. Thus, we will not concentrate our discussion on architectures for KBMSs, but on the different characteristics arising when working with these systems.

In comparison with, for example, relational DBMSs [Co70], KBMSs manage more complex and structured objects, and also different types of abstraction relationships. In fact, the abstraction concepts are the most important constructs to be supported by KBMSs [Ma88]. The object descriptions embody (or at least should do it) all abstraction concepts so that objects can play different roles at the same time (i.e., an object can be, at the same time, a class, a set, a component, an instance, etc.), depending on the relationships they have to other objects [Ma91]. These characteristics lead to the conclusion that most KBs features can be visualized as graphs [CHM92], i.e., graphs can be taken as an abstract representation for KBs. Note that we are talking about graphs, not about hierarchies, because it is common to have objects in a KB with more than one parent.

Another important aspect of KBMSs is their active behavior, be it used for redundancy control or for user-defined reactions to in- and outside events of the real world (which could be translated to the rules, methods, demons, etc. commonly handled by KBMSs). All of them are important tasks to be managed by KBMSs, and must also be taken into consideration when talking about concurrency control.

The power of the languages provided by the KBMSs is also greater than the ones usually provided by common systems, e.g., relational languages. The query formulation can make use of the different abstraction relationships and the complex structure of the objects. Additionally, reasoning mechanisms (deduction and inference mechanisms) are also available, which can be used to draw conclusions or generate new knowledge from facts and suppositions.

Still about reasoning, but now based on the existence of the abstraction relationships, another important feature of KBMSs is their built-in reasoning facilities, which can also be used to make deductions about objects [Ma91], e.g., inheritance is used to reason the internal structure of instances, membership stipulations are used to reason beliefs about elements, implied predicates are used to make conclusions about aggregation objects based on the monotony of properties, etc.

As we have been advocated, KBMSs are especially useful when applied to multi-user environments. Thus, the synchronization aspect plays an important role. Queries could, for example, be applied to the complex objects and their abstraction relationships, producing the evaluation of rule sets with great processing units, which, in turn, claim large granularities of synchronization and concurrency. Therefore, KBMSs claim the use of special synchronization mechanisms and control structures, adaptable to their environments. Such structures could not adequately be built using normal, flat ACID transactions [HR83]. Probably, spheres of control [Da73, Da78] or nested transactions [Mo85] provide better control structures and tiny processing granules, which could be better suitable for the KBMS environment.

The semantic knowledge of transactions is another factor to be thought of, which could also be used to increase the concurrency [Ga83, Ly83, FÖ89]. The main idea behind this use of transactions' semantics is to allow nonserializable schedules, which preserves consistency and which are acceptable to the system users. To produce such semantically consistent schedules, the transaction processing mechanism receives semantic information from the users in the form of transaction semantic types, compatibility sets, steps, countersteps [Ga83], and also breakpoints [FÖ89]. With respect to KBMSs, the methods could be a starting point to the applicability of this approach. The semantics of user- or system-defined methods could be considered in order to allow more general, non-serializable schedules of methods to be produced. Such a semantic knowledge use could significantly decrease the transaction response time, and could then be useful when the cost of producing only serializable interleavings is unacceptably high.

Anyway, structures and techniques for concurrency control are to be developed for the KBMS environment. Nevertheless, this new research direction must be aware of the KBMS environment itself, its active behavior, characteristics, reasoning facilities, abstraction relationships, methods, demons, rules, etc. In other words, these techniques must make use of the KBMS features in order to obtain better performance and optimization.

5. Concurrency Control Techniques

In this section, we will discuss some of the well-known concurrency control techniques for databases, and their (in)adequacies to the KBMS environment. First of all, we will present a technique, and thereafter apply it directly to our environment. Thus, we will show where this technique could be useful and, if this is the case, where it is just intolerable, making a kind of critical analysis of them. Let us start with the most popular technique in commercial products, i.e., two-phase locking.

5.1 Two-Phase Locking

Locking is a mechanism commonly used to resolve the problem of synchronizing access to shared data [BHG87]. The *Two-Phase Locking* (2PL, for short) protocol was first introduced in [EGLT76]. Besides grouping the actions of a process into sequences, called *transactions*, which are units of consistency, this lock protocol introduces an additional set of actions, namely, *lock* and *unlock*. In turn, two distinct types of locks can be differentiated, the so-called eXclusive (X) and Shared (S) locks. Exclusive locks are used for updating an object, whereas shared locks are used for reading it. Further, the transactions are required to ask for (acquire) a lock on each object, before executing operations on it. By this way, a lock on an object is used to ensure that, when granted to some transaction, this particular transaction has access to the requested object (in the mode (X/S) defined by the lock operation). Finally, the unlock operation releases this object, enabling so its access by other transactions, meaning that this transaction will no longer need to access it.

With respect to conflicts between different lock modes, the only not conflicting lock pair is the shared one (S - S), meaning that two transactions may read the same object at the same time. The other ones (S - X, X - S, and X - X) conflict with each other, and so can only be granted to one transaction at a time, i.e., if a transaction must acquire a conflicting lock, it must wait until the transaction that owns that lock releases it. In summary, two operations conflict if they operate on the same object and at least one of them is a Write. The system (scheduler) thereby ensures that only one transaction can hold a Write (X) lock on an object at a time, and so only one transaction can update this object at a time.

The essence of the 2PL protocol is that consistency requires that a transaction must be constructed to have a growing and a shrinking phase [EGLT76] (thus the name *two-phase*). During the growing phase, a transaction can request new locks. However, once a lock has been released, the transaction cannot request a new one (then the shrinking phase begins, where all locks are being released).

In simplified terms, 2PL works as follows (suppose the lockable units are data items):

(1) Each data item has a distinct lock associated with it. Before accessing some data item, a transaction must require a lock on it;

(2) If an S lock on a data item is granted to a transaction, no other transaction may access that data item in X mode, while the first one holds that lock (notice that several S locks on the same data item are allowed);

(3) If an X lock on a data item is granted to a transaction, no other transaction may access that data item in any mode, while the first one holds that lock;

(4) Once a transaction releases any lock, it cannot acquire any additional locks.

It can be proven that 2PL ensures serializability, and so produces consistent executions. It is beyond our goal to prove the correctness of 2PL. Proofs can be found in [EGLT76], and elsewhere [BSW79, Pa79, BHG87, GR93].

The protocol above described is the basic 2PL. There is also a variant of this protocol, used in almost all implementations of 2PL [BHG87], called *strict 2PL*. It differs from the basic one in that it requires the transaction to release all its locks together, when it terminates (either commits or aborts). Among the reasons for adopting this policy, the most important one is to guarantee a strict execution. Guaranteeing a strict execution means avoiding cascading aborts and producing recoverable executions [BHG87]. Due to that, this strict variant is the most commonly used in the implementations of 2PL. From now on, whenever we mention the 2PL protocol, we will assume this strict variant of it.

Now, let us apply the 2PL protocol in a practical example using our KB, building a new scenario. Suppose John wants to read an airplane (mig25) and after that to update an automobile (benz500). In a contrary way, Mary wants to read an automobile (benz500) and thereafter to update an airplane (mig25). The situation can be sketched as follows:

John	Mary
begin	---
S-lock (mig25)	begin
read (mig25)	S-lock (benz500)
---	read (benz500)
X-lock (benz500)	---
wait	X-lock (mig25)
wait	wait
. . .	. . .

This is probably the easiest and most classical way in which locks are acquired using 2PL so that a *deadlock* may arise. John’s transaction will be waiting for the release of a lock held by Mary’s transaction, and at the same time Mary’s transaction will be waiting for a lock held by John’s transaction. Deadlocks happen whenever there is a cyclical sequence of transactions each waiting for the next to release a lock it must acquire (transaction 1 waits for transaction 2 that waits for 3 (. . .) that waits for 1). In our example, if no action were taken, John and Mary would wait for each other forever.

Nevertheless, 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 aborts it (although imprecise in the detection of deadlocks, it works). *Waits-for-graph* [Ho72] is another one, 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* [8], aborts it, obliterating its effects from the database, and restarts it again.

There are also other strategies for detecting deadlocks, but the best known are both previously mentioned. In spite of the existence of many such strategies for deadlock detection and resolution, the problem with 2PL is that it does not avoid them. [GR93] advocate that deadlocks are very, very rare events, but [Ya82] says that it is very easy to construct scenarios where deadlock arises, and [SK80] state that deadlock detection and recovery in general is an expensive task and should be avoided whenever it is possible. Anyway, deadlocks may ever arise if the lock protocol does not avoid them, and when they happen the

[8] 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*. Shortly described, 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].

system must have at hand some strategy to detect them and resolve the problem. A mechanism to deal with such events, be it simple or not, consume resources, which may probably be needed by other applications.

In Sect. 3, we have mentioned the main concurrency control problems, but one of these problems was not exactly detailed there, and postponed to subsequent sections, namely, the phantom problem. Now it is time to talk about phantoms! Usually, real databases do not contain static structures, where just read and write operations are allowed. Instead, they in fact contain dynamic structures, where records can be inserted and deleted at any time. The problem due to this dynamism can better be understood with an example. Suppose our usual users, John and Mary, are working concurrently in our example KB. John wants to know how many red automobiles are available in the KB and the total price of them. But after initiating his transaction and before terminating it, Mary inserts a new object in the KB, e.g., the object ferrari, which is red and also an instance of automobiles. The following sketches this situation:

John	Mary
begin	---
S-lock (bmw850)	---
read (bmw850)	---
S-lock (porsche959)	---
read (porsche959)	---
S-lock (benz500)	begin
read (benz500)	insert (ferrari)
---	commit
commit	---

The problem here is that John will end his transaction with an inconsistent result, not reflecting the new object inserted by Mary. The situation shown above is a simple one, but it is not so hard to imagine situations with even more drastic inconsistent results. If we use 2PL, we cannot prevent someone else from inserting new records in the database [9], just because there is no lock on nonexistent records. The same problem can happen if we use a delete operation, where the record may seem to have disappeared for the retrieval transaction. Such new or deleted records are called phantoms, because they seem to appear and disappear like a ghost. The phantom problem was first introduced in [EGLT76], which also proposed a new locking policy, namely predicate locks (see next section), for dealing with such situations. The phantom problem is the concurrency control problem for dynamic databases [BHG87], and the 2PL protocol seems to not always guarantee correct executions for dynamic databases. [GR93] state that there is no pure record-locking mechanism which can avoid phantoms.

Deadlocks and phantoms are really a problem to be handled by 2PL implementors. But there is another problem even worse than this, if we consider 2PL in an environment like the one typical of KBMSs. In a matter of fact, the 2PL policy does not take into account the structure (graph) the objects build in a KB (it is just unaware about the existence of such a structure and its semantics). This has serious performance implications for the type of transactions likely to be applied to KBs. Let us better understand this problem with some examples. Suppose a user wants to lock either some class, and all its instances, or some aggregate and all its components. Such lock requests on the basis of 2PL are just problematic to be realized. In the first case, the 2PL does not know what a class is and not even that a class may have any number of instances. It is just aware about objects (or records), but not about the abstraction relationships between

[9] Notice that we are talking about the original, pure version of 2PL. In particular, the predicate lock protocol (to be discussed in the next section) cope very well with such a problem.

these objects. To grant these locks, the lock manager would have to require locks on all objects individually, i.e., the 2PL does not recognize a class and all its instances as a single lockable granule. This same observation is also valid in the second case, i.e., there is no possibility to request a lock on an aggregate and all its components as a unit.

Analyzing the behavior of the 2PL in these situations, we come up with two major problems. First, the lock manager may run out of storage with the necessity of acquiring many locks. If we imagine a class with thousands of instances, the lock manager would have to acquire thousands of locks, which is certainly a great problem to be handled and thought of. Second, although some locks may not semantically conflict, they will be handled as conflicting ones. Suppose the above mentioned class and aggregate are the same object. If one user wants to exclusively access this object through its instance relationships (i.e., the object as a class and all its instances), and the other one through its component relationships (i.e., the object as an aggregate and all its components), they would be obligated to serially perform their transactions. This is due to the fact that the 2PL simply does not know that an object may simultaneously be a class, an aggregate, etc. Hence, the 2PL should recognize an object and its several relationships to other objects as different locking granules, and not as a simple object (or record) that must be locked. Such behavior may reveal drastic performance problems and at the same time significantly reduce the concurrency.

As we have seen, the 2PL protocol does not seem to be appropriate for the KBMS environment. First, it does not take into account the objects and their different abstraction relationships to other objects, i.e., the semantics of the structure built by these objects. Second, it does not avoid deadlocks, and detecting and resolving them consume resources, even if a simple, cheap mechanism is used. Third, it is not capable of dealing with phantoms.

5.2 Predicate Locks

The idea of using predicate locks for database concurrency control was first introduced in [EGLT76]. Although there are not many commercial systems which use predicate locking as a primary method for concurrency control, it is an elegant solution to the phantom problem [GR93].

The basic idea behind predicate locks is that transactions can require locks for a specific subset of the database to which the lock applies, rather than locking individual records. In order to better understand this protocol, consider, for example, our last scenario, where John wants to know how many red automobiles there are in the KB and their total price. Using this protocol, he would then request a lock with a predicate like: lock all red automobiles. By this way, all red automobiles of the KB would be locked, and also the nonexistent ones (considering thus possible phantoms). Thus, Mary would have to wait until John’s retrieval transaction terminates, in order to include a new red automobile in the KB. This situation is illustrated in the following:

John	Mary
begin	---
S-lock all red automobiles	---
read (bmw850)	---
read (porsche959)	begin
read (benz500)	X-lock red automobiles
---	wait
commit	wait

---	insert (ferrari)
---	commit

With this situation, we can observe that the phantom problem is resolved. In general terms, what the system has to do is to compare whether the predicates conflict or not. If they conflict, the transaction must wait until this predicate lock is released. If not, the lock is granted to the transaction, and all objects satisfying this predicate will be locked by it. Essentially, two predicate locks are compatible if [GR93]:

(1) The transaction which is requesting the lock is the same that holds it; or

(2) Both predicates are in shared mode (shared locks do not conflict); or

(3) No object satisfies both predicates.

Whatsoever out of the three cases above is said to conflict. In summary, a system implementing predicate locks works so that each time a transaction requests a predicate lock, it compares this request with the other granted and waiting predicate lock. If the request is compatible with all others, it is added to the granted set and immediately granted. If not, it is added to the waiting list. Whenever a transaction terminates, its predicate is removed from the granted set, and the system considers again each predicate lock in the waiting list. The system then grants each predicate which is compatible with the new granted set, and adds them to this set. If a system follows such a protocol it produces correct schedules of concurrent transactions. We will not prove the correctness of this protocol, it is beyond our goal (such a proof can be found in [EGLT76]).

Despite of resolving the phantom problem and providing the I (isolation) of ACID [HR83], predicate locks have three great shortcomings, leading to its lack of applicability:

1. It is not computationally efficient to check overlap between two predicate locks, and maintaining a predicate lock table is also very costly. [Mo90] advocates that comparing a new predicate against a predicate lock table of some reasonable size (e.g., containing 100 expressions) is just prohibitive. Moreover, [GR93] state that predicate satisfiability is known to be NP-complete (the best algorithms for it run in time proportional to 2^N). In addition, there may be some very complex predicates that it may become too difficult to decide whether two distinct predicates define overlapping sets or not (and hence whether they conflict as locks).

2. Predicate locks are somewhat pessimistic [GR93]. Two predicates may be incompatible following the rules for predicate comparison, but there may be some integrity constraint which make them compatible, i.e., the system just does not understand some integrity constraints (which are commonly used in KBMSs, and might help the system to improve concurrency). [GR93] cite a nice example of such cases: “If I lock the mothers in a department and you lock the fathers in the same department, the lock manager may not know that fathers can’t be mothers”.

3. Predicates may be arbitrarily complex, so that to discover them may be also a very difficult task.

Due to these drawbacks, predicate locks are just inapplicable in commercial systems, and implementations in this direction have not been very successful. At a first sight, predicate locks might seem to work well in the KBMS environment. It resolves the phantom problem, and it seems good for working in hierarchies. Unfortunately, the obstacles to its applicability are too great: its execution cost is unpayable; it is too pessimistic; and the predicates may be too complex to be tractable.

5.3 Granular Locks

Granular locks could be said to be a practical way of using predicates in a lock mechanism. They were first introduced in [GLPT76], and are also known as Directed Acyclic Graph (DAG) protocol, or as Multigranularity Locking. The basic idea of this protocol stays in choosing lockable units, which are atomically locked by the system to ensure consistency and to provide isolation. Lockable units could be, for example, databases (or KBs), files, sets, subsets, records, fields, and so on.

When choosing the lockable units for implementing this protocol, one will always be 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 a transaction would have to acquire and maintain a large number of locks [GR93], which imply 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 granular lock protocol satisfies both of these situations, allowing lockable units of different granularities to coexist in the same system.

After choosing the lockable units, preferably fine and coarse ones, these are organized as a hierarchy. Using our example KB (Fig. 1) as a very simple illustration for this case, we could say that each node of this hierarchy is a lockable unit, and so each one of them can be locked. If one requests eXclusive/Shared access to a particular node, as soon as the request is granted, the requester has eXclusive/Shared access to that node, and implicitly to each of its descendants [GLPT76]. These two access modes lock an entire subtree rooted at the requested node. In our example, by putting one lock at the class vehicles, we implicitly lock all descendants of it, i.e., automobiles, bmw850, porsche959, benz500, trucks, and boats, thus saving the locking overhead.

Moreover, in order to lock a subtree rooted at some node in shared or exclusive mode, it becomes important to prevent locks on the ancestors of this node which might implicitly lock it in an incompatible mode. This was achieved with the invention of a new lock mode, one different from shared or exclusive modes. This new lock mode was called *intention mode* [GLPT76], and it represents the intention to set locks at a finer granularity, thereby preventing implicit or explicit locks on the ancestors. Intention mode was refined to *Intention Share mode* (IS) and *Intention eXclusive mode* (IX) to indicate shared or exclusive access at the descendants. And finally, *Share and Intention eXclusive mode* (SIX) was introduced for transactions that want to read an entire subtree but will update only a few of the items. The compatibility matrix for granular locks is shown in Fig. 2, where, for completeness, a *null mode* (NL) is also used, to represent the absence of requests of a resource by a transaction [GLPT76].

		Granted Mode					
		NL	**IS**	**IX**	**S**	**SIX**	**X**
Requested Mode	**NL**	yes	yes	yes	yes	yes	yes
	IS	yes	yes	yes	yes	yes	no
	IX	yes	yes	yes	no	no	no
	S	yes	yes	no	yes	no	no
	SIX	yes	yes	no	no	no	no
	X	yes	no	no	no	no	no

Figure 2: Compatibility Matrix for Granular Locks.

Summarizing, the lock modes are [GLPT76]:

- **NL** - Gives *no access* to a node, representing the absence of a resource request.
- **IS** - Gives *intention share access* to the requested node, and *allows* the requester to *explicitly lock* descendant nodes in S or IS mode.
- **IX** - Gives *intention exclusive access* to the requested node, and *allows* the requester to *explicitly lock* descendant nodes in X, S, SIX, IX, or IS mode.
- **S** - Gives *share access* to the requested node, and *implicitly to all* descendant nodes without setting further locks.
- **SIX** - Gives *share and intention exclusive access* to the requested node (i.e., *implicitly locks all* descendant nodes in share mode and *allows* the requester to *explicitly lock* descendant nodes in X, SIX, or IX mode).
- **X** - Gives *exclusive access* to the requested node, and *implicitly to all* descendant nodes without setting further locks.

In [GR93] a new lock mode was introduced to the previous basic set, namely *Update* (U) lock mode. The main reason of its existence is to prevent a common form of deadlock, which appears when two transactions read some *hot-spot* (frequently updated records/pages) at nearly the same time, getting a share mode lock on it, and thereafter both convert it in exclusive mode to perform some update on it. Using the U lock mode transforms many deadlocks into lock waits [GR93]. In summary, it gives the requester read authority to the node and to its descendants (like S does), and prevents others from holding non-shared locks (one of {X, U, SIX, IX, IS}) on this node or its descendants (in reality, it represents an intention to update the node in the future). In particular, we will not give further attention to this new lock mode here, because it in fact does not change the essence of the granular lock protocol, but, as already said, just prevents some common deadlocks.

Finally, transactions obeying the granular lock protocol must follow the rules below:

(1) Request locks from root to leaf.

(2) Release locks from leaf to root.

(3) Before acquiring an S or IS mode lock on a non-root node, at least one parent (and by induction a path to a root) must be held in IS mode or higher (one of {IS, IX, S, SIX, X}).

(4) Before acquiring an X, SIX, or IX mode lock on a non-root node, all parents (and as a consequence all ancestors) must be held in IX mode or higher (one of {IX, SIX, X}).

Notice that: first, transactions are not allowed to leap into the middle of the tree and begin locking nodes at random (this would lead the protocol to malfunction); second, leaf nodes are never requested in intention mode since they have no descendants. If the transactions follow this protocol, correct executions are produced, and isolation is provided (again it is beyond our goal to prove the correctness of this protocol here, proofs can be found in, e.g., [GLPT76], [BHG87], and [GR93]). In addition, the rules showed above are already extended to work in DAGs [10]. By this way, they may be used for hierarchies, trees, as well as for DAGs.

The idea behind these rules is that share locks on a node require that all nodes on at least one path from that node to a root be covered by locks chosen from the set {IS, IX, S, SIX, X}. On the other hand, exclusive locks on a node require that all nodes on all paths from that node to all roots be covered by locks chosen from the set {IX, SIX, X}.

Another important concept covered by the granular lock protocol is *lock conversion* [GLPT76]. Lock conversions are normally used to increase (upgrade) the access mode a transaction has to a record (for example, if a transaction has read some record and wants to update it afterward, it can request the system to upgrade its lock on this record from S to X). 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 the supremeness of the old and the requested mode (see Fig. 3). So, for example, if one has IX mode and requests S mode, then the new mode is SIX [GLPT76].

		Old Mode				
		IS	**IX**	**S**	**SIX**	**X**
Requested Mode	**IS**	IS	IX	S	SIX	X
	IX	IX	IX	SIX	SIX	X
	S	S	SIX	S	SIX	X
	SIX	SIX	SIX	SIX	SIX	X
	X	X	X	X	X	X

Figure 3: Lock Conversion Table for Granular Locks.

A last 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

[10] A tree node has only one parent, a DAG node not necessarily.

several lockable units covered by the current lock mode [11]. In such cases, they may get a single, coarse-granularity lock. The past history of a transaction's behavior can also be 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 1,000 according to [GR93]) of a given granularity, then 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 concurrency control method and for preventing the lock system to run out of storage when millions of locks are acquired.

Summarizing, the granular lock protocol provides most of the benefits of predicate locks (among them, the avoidance of the phantom problem), and at the same time avoids the high cost of predicate comparisons. Clearly, the fundamental motivation of the granular lock protocol is the minimization of the number of locks to be set in accessing the database [Gr78]. For example, when most of the instances of a class are to be accessed, it makes sense to set one lock for the entire class, implicitly locking its instances, rather than one lock for each instance of the class. This protocol (and some extensions of it) has been a very popular approach in object-oriented databases. One such example is ORION [KBCGW89] which uses this protocol with the five basic lock modes (IS, IX, S, SIX, and X), and some extensions necessary to deal with its composite objects and class lattices [GK88].

But now, let us analyze some problems arising when the granular lock protocol is used in the context of complex objects and so in the KBMS environment. In order to lock an object, a transaction must set intention locks on all the parents. This is a serious limitation when an object is likely to be used by many of the parent objects. For example, consider an aggregation relationship. An object, say screw, will be used in thousands of assemblies all of which will be its parents. In such situations, it is very inefficient to set intention locks on all the parents [HDKRS89].

Another shortcoming of the granular lock protocol with respect to the aggregation relationship is that it does not recognize an object and its parts as a single lockable granule, like a class or an instance of a class [KBCGW89]. To lock a composite object using this protocol means either locking all component classes on an aggregation hierarchy, or locking all constituent objects within a composite object. Clearly, neither option is satisfactory. The former results in the locking of all composite objects that belong to the composite object hierarchy, whereas the latter can result in a large number of locks. Therefore, a composite object should be used as a unit of locking to reduce the system overhead associated with concurrency control.

Another problem is that the implicit locks on a child object are visible only if it is accessed by a specific set. For example, using an extension of our KB (Fig. 4), suppose we have a new automobile, say a 007 one, which runs, sails, and flies. Thus, this new means of transportation will be an instance of automobiles, of boats, and of airplanes, at the same time (if one wants a more realistic example, there are a lot of amphibious vehicles which could also be used). Now, suppose one of our usual users, say John, wants to read all instances of automobiles. All he must do is to acquire an S lock on automobiles (along with IS locks on vehicles and means_of_transportation) and all instances of it will then implicitly be locked on share mode without setting further locks. Note that the lock itself stays on the class automobiles, and not on its instances. Just after John got his lock, Mary wants to update all instances of airplanes. To do so, she sets an X lock

[11] 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.

on airplanes (along with IX locks on aircrafts and means_of_transportation) and in the same way all instances of it will then implicitly be locked in exclusive mode without setting further locks. The following scenario sketches this situation:

John	Mary
begin	---
IS-lock (means_of_transportation)	begin
IS-lock (vehicles)	IX-lock (means_of_transportation)
S-lock (automobiles)	IX-lock (aircrafts)
read (bmw850)	X-lock (airplanes)
read (porsche959)	write (b52)
read (benz500)	write (mig25)
read (007)	write (007)
---	---
---	commit
commit	---

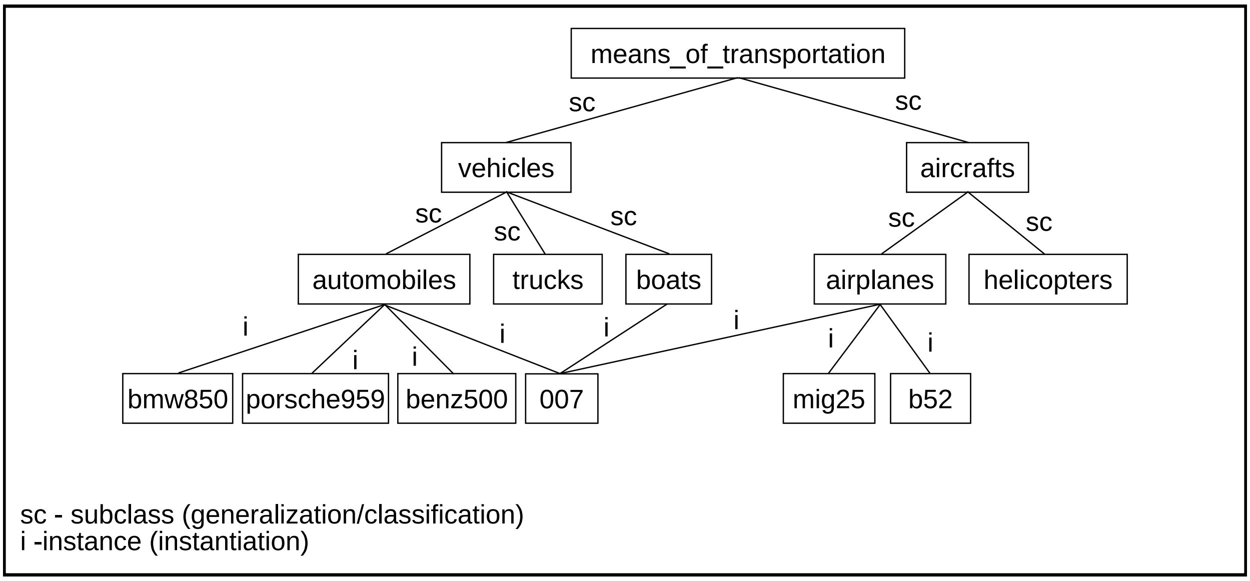

Figure 4: A New Example Knowledge Base.

The situation here makes clear that it is possible to have an object (007 in the example) implicitly locked in two conflicting modes. In this way, the concurrency control problems arise again, and so we can obtain inconsistent results with the interleaving of transactions. In general, using the granular lock protocol, if an object has multiple parents, it may be accessed via a parent that does not have any locks on it, thus violating the implicit locks and leading to inconsistencies [12].

A last and fatal problem to the appliance of the granular lock protocol to the KBMS environment is the own structure assumed by this policy. The generic extension of the granular lock protocol, presented in this section, is able to deal with directed acyclic graphs [13]. As the name itself says, cycles are not allowed. It is not difficult to see that using the locking rules of this policy, it is impossible to lock any node on a cycle.

[12] ORION's implementors have addressed this problem in their work [KBCGW87, GK88], and [GR93] have changed a little of the semantic of the lock modes of the granular lock protocol in order to avoid such a problem.

[13] Simple versions of this protocol work only on hierarchies and trees.

Unfortunately, the structure of a KB will contain cycles, for example, in the inference graph generated for a collection of recursive rules [CHM92]. Another example of cycles can easily be found with user-defined attributes. Suppose one creates an attribute *likes* for an object *person*, meaning the set of persons this person likes. As soon as two persons like one another a cycle is found. Due to this lack of functionality, the DAG policy cannot directly be applied to KBMSs.

In this way, the granular lock protocol in its basic essence does not seem to be appropriate for the use in KBMSs. First of all, it cannot handle cycles in the underlying structure. Implicit locks may be violated when some object is accessed via a parent that does not have any locks on it. Intention locks may show poor performance when an object is likely to be used by many of the parent objects. Additionally, although not discussed, granular locks are subject to deadlocks, especially when lock conversion is supported.

5.4 Dynamic Directed Graph Protocol

The Dynamic Directed Graph (DDG) policy is an algorithm for concurrency control specifically designed for KBMSs. It was proposed in [CHM92] and is an extension of the locking protocol for hierarchical database systems of [SK80]. The former is able to cope with cycles and updates in the underlying structure, what is not considered by the latter.

First of all, the DDG policy assumes that a KB is a directed graph with a set of nodes and edges [14] (both generally denoted entities). The operations (insert, delete, and access) to be executed against these entities are considered to be atomic. In addition, the transactions performing these operations are assumed to be *defined* in the current KB state, i.e., they do not insert (delete, access) an entity that already exists (does not exist) in the KB [15] [CHM92]. Furthermore, lock and unlock operations for an entity are also defined, meaning respectively the acquisition of a lock on an entity and the release of it.

The locking rules of the DDG policy assume that the underlying graph (the working structure) is always connected and has a single root. [CHM92] present some rules for restricting the KB to a rooted and connected graph. First, there are preprocessing rules, applied to the KB when it is initially started, to convert its structure to this restricted form. Second, there are structure maintenance rules to guarantee that the graph will stay in this form, as changes undergo over time. We will not give details of these rules here, and will assume that the KB already presents this restricted form (a comprehensible description of them and reasons for their existence can be found in [CHM92]).

Before specifying the locking rules of the DDG policy, it is necessary to give some definitions presented in [CHM92]. First of all, a cycle is dealt with by considering the *strongly connected component* (SCC) of the rooted and connected graph, a part of the graph containing all the nodes on that cycle [16]. A *dominator* of a set of nodes in the graph is the one such that all paths from the root node to each node of this set of nodes

[14] The different types of relationships between the nodes are not taken into consideration. Thus, they do not distinguish the different types of edges that a KB may have.

[15] This may be considered a limitation of the protocol, because there may be operations which are known to be not defined only after submitting them for execution and becoming an exception, i.e., an unexpected return value. For example, how to know that an object X does not already exist in a KB? The only way is to perform an access operation on it, and receive an answer, which can signal its existence or not (an exception).

[16] Just non-trivial SCCs are considered, i.e., SCCs having more than one node.

pass through it. *Entry point* of an SCC is a node such that there is at least one edge binding this node to some node out of the SCC. Now, the locking rules can better be understood [CHM92]:

(1) The first node to be locked by a transaction is the dominator of the set of nodes to be accessed by this transaction with respect to the graph.

(2) Before a transaction performs any operation (insert/delete/access) on an entity (node/edge), it has to lock this entity.

(3) A node can be locked if and only if all its predecessors in the present state of the graph, that do not lie on the same SCC as it, have been locked by the transaction in the past, and the transaction is presently holding a lock on at least one of them. All nodes on an SCC are locked together in one step, provided all the entry points of that SCC have been locked. A node that is being inserted can be locked at any time.

(4) Each node can be locked at most once by each transaction.

Following these rules, the system produces correct executions, and is free from deadlocks (proofs can be found in [CHM92]). In addition, transactions are allowed to release locks, and proceed locking other objects. The fact that transactions are able to acquire locks even after releasing some of them shows a clear improvement over 2PL.

Let us consider these rules in more detail. The first rule has two basic points. First, the set of nodes to be accessed by a transaction needs to be previously known, i.e., one always has to have at hand all nodes to be accessed by the transaction. This has the clear shortcoming that transactions are not allowed to be open-ended, or to acquire locks on demand. With such transactions, some important aspects of KBMSs cannot be supported adequately, for example, the World concept [Th91, Re92]. This key restriction is due to the second important point of this first rule, namely, a transaction has to start locking the node that dominates the rest of the nodes that it is ever going to access. By doing this, the dominator needs to be known [17]. And if, while executing, a transaction realizes that it must access a node that is not dominated by the node from where it started, it must be aborted and must start from a higher node [Ch93]. This proceeding may be dangerous because, due to concurrent updates on the KB, some of the dominator information may become stale, and so there may be a great percentage of transactions that must be aborted. Chaudhri is estimating this effect in his current work on implementation and performance of the DDG policy [Ch93].

The second rule is basically for guaranteeing that a transaction does not operate on a node or edge not previously locked by it. Thus, transactions are required to be *well-formed*. Particularly, a transaction is said to be well-formed if all its actions on objects are covered by locks, and if each lock action is eventually followed by a corresponding unlock action [GR93]. The former requirement is covered by this second rule, and the latter one is satisfied because transactions can release locks on demand, or as soon as they terminate.

The third rule is the heart of the DDG policy. First of all it assumes the present state of the rooted and connected graph, meaning that changes on it can undergo over time, and it still works correctly. Locking

[17] Dominators are found using a bit-vector algorithm given in [ASU85]. Moreover, if more than one dominator are found, the closest dominator is used [Ch93]. For example, if a node A has two dominators B and C, such that B is a descendant of C, then the node B is considered. This is because there is no point locking more nodes than really needed.

predecessors out of the SCC is necessary, among other things, to guarantee the correct behavior of the policy, and so to provide isolation. Requiring all nodes on an SCC to be locked in one step (by locking together all the entry points of the SCC) is basically needed for deadlocks avoidance. Further, the nodes to be inserted have to be locked to maintain the correctness of this policy [Ch93]. The reason such nodes may be locked at any time is that they are not yet a part of the structure of the graph, and therefore, the usual locking rules (all predecessors locked in the past, and one of them currently locked) do not apply to these.

Locking predecessors works as follows [Ch93]. When a transaction begins execution, the lock manager locks the dominator node of the transaction. For subsequent requests, if all the predecessors are not locked, the lock manager issues requests for lock on those predecessors (this process may continue recursively, and must terminate because the dominator is locked). Thus, there are some lock requests issued by the transaction, and the others issued by the lock manager itself to enforce the locking rules. Lock on a predecessor can be released if the transaction does not need it, and no other successor would require it to be locked later on [18].

Particularly, the information about the entry points is generated at compile time using a depth first search algorithm [Ch93]. This information is incrementally maintained using an incremental algorithm. So, while enforcing the locking rules, this information is already available. The costs for generating this information can be disregarded, because it is made only once and at compile time. Nevertheless, the costs for maintaining this information up-to-date may consume reasonable amounts of resources, if the KB is likely to be frequently structurally changed (although this is not always the case according to [Ma90]).

Still with respect to the third locking rule, we can see that the DDG policy is simple as it treats SCCs as a unit of locking. It requires the entry points of an SCC to be atomically locked, providing so a one-step locking of SCCs. This has the side effect of not permitting any concurrency within the cycles. In order to permit concurrency within cycles, [CHM92] presented a variation of this third locking rule, reaching this goal in a limited extent. We will not give details of this variation here, yet just some comments. First of all, this performance improvement is only obtained for SCCs having more than one entry point, and only if some nodes can be accessed through different entry points. Every case not satisfying these two conditions shows no difference in the concurrency. Since this variation requires more bookkeeping effort and show performance gains only in very specific situations, it is not likely to be an applicable one [CHM92]. Thus, it remains the drawback of not allowing concurrency within cycles. And if cycles are too long, involving many objects, this may show a serious limitation in the performance of this policy, because transactions are then required to retain all the locks on an SCC, according to the fourth rule, until it has finished processing it.

Finally, the last (fourth) rule is necessary to guarantee the *safety* of a locking policy [Ya82a], and so to support the proofs of the correctness of the DDG policy found in [CHM92].

With these clarifications about the DDG policy locking rules, we are able to apply them in a practical example. Let us use our last situation mentioned in the previous section, where John wants to access the class automobiles together with all instances of it, whereas Mary wants to access the class airplanes and in the same way all instances of it. In summary, we have the following situation:

- John - wants to access automobiles and all instances of it, i.e., bmw850, porsche959, benz500, and 007.

[18] According to [Ch93], this logic was implemented by extending the data structures of a conventional lock manager.

- Mary - wants to access airplanes and all instances of it, i.e., 007, mig25, and b52.

Due to the assumption of the DDG policy that the underlying graph must be rooted, connected, and the edges directed, we need to modify a little bit our usual example KB by giving a direction to its edges (see Fig. 5). Further, analyzing the set of nodes to be accessed by John's transaction and also by Mary's transaction, we can observe that the only dominator, common to both transactions, is the root node itself, namely means_of_transportation. With respect to SCCs, we can see that this example KB does not have any of them (we have no cycles). Now, let us try to mount an execution plan for both transactions. One possibility is sketched below:

John	Mary
begin	---
Lock (means_of_transportation)	begin
Lock (aircrafts)	Lock (means_of_transportation)
Lock (airplanes)	wait
Unlock (airplanes)	wait
Unlock (aircrafts)	wait
Lock (vehicles)	wait
Unlock (means_of_transportation)	wait
Lock (automobiles)	Lock (vehicles)
Access (automobiles)	wait
Lock (bmw850)	wait
Access (bmw850)	wait
Unlock (bmw850)	wait
Lock (porsche959)	wait
Access (porsche959)	wait
Unlock (porsche959)	wait
Lock (benz500)	wait
Access (benz500)	wait
Unlock (benz500)	wait
Unlock (automobiles)	wait
Lock (boats)	wait
Lock (007)	wait
Access (007)	wait
Unlock (007)	wait
Unlock (boats)	wait
Unlock (vehicles)	wait
commit	Lock (automobiles)
---	Lock (boats)
---	Unlock (boats)
---	Unlock (automobiles)
---	Unlock (vehicles)
---	Lock (aircrafts)
---	Unlock (means_of_transportation)
---	Lock (airplanes)
---	Unlock (aircrafts)
---	Access (airplanes)
---	Lock (007)
---	Access (007)
---	Unlock (007)
---	Lock (mig25)
---	Access (mig25)
---	Unlock (mig25)
---	Lock (b52)
---	Access (b52)
---	Unlock (b52)

---	Unlock (airplanes)
---	commit

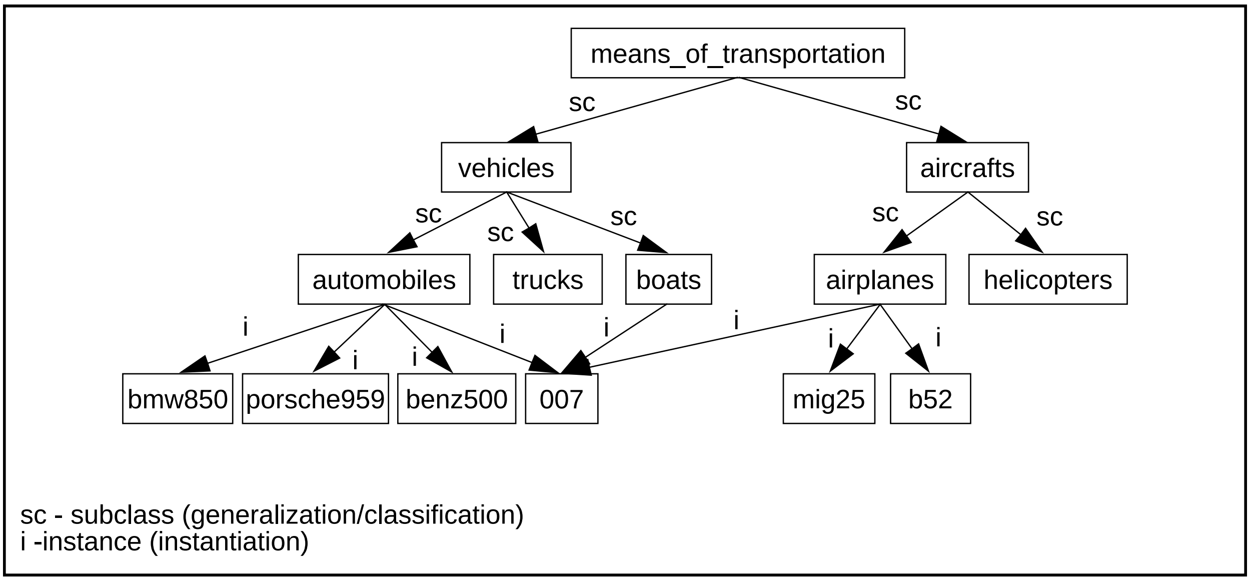

Figure 5: An Example Knowledge Base with Directed Edges.

Notice that there can be more than one way to enforce the DDG locking rules [19]. Therefore, the above sketched execution plan is not the only possibility. As we can see with this example, there are some nodes that need to be locked even though they are not needed by the transaction (this is due to the third locking rule, which requires that a transaction must lock at least once all the predecessors of the objects being accessed during its existence). Another important aspect which can be observed in the example is that, albeit the transactions conflict with each other on only one object (007 in the example), they must almost be processed serially (also due to the third locking rule). With this observation, it raises the question whether the DDG policy can really show better performance results than the others previously explained or not.

Further, although being a very important aspect of concurrency control, the locking conflicting modes are not taken into consideration by the DDG policy [20]. It is always assumed that the operations conflict with each other, whichsoever they are. Clearly, performance gains would be obtained if conflicting lock modes were defined, allowing, for example, many access transactions to work concurrently, accessing their objects at the same time. Thus, using the DDG policy, transactions that read the same set of objects, although not conflicting with each other at all, must be processed serially. By this way, for KBs likely to be frequently queried (the most cases according to [Ma90]), this locking policy may show very poor performance results, due to its disregard with respect to nonconflicting operations and lock modes.

Therefore, although being tailored for KBMSs and correctly working on all situations (cycles and objects with multiple parents are correctly handled), the DDG protocol seems not to fulfil all the requirements that a

[19] There is an algorithm to come up with the above execution plan [Ch93].

[20] [CHM92] ignore lock conflicting modes, but [Ch93] states that studies have been made to obtain a new DDG policy with shared and exclusive locks, called DDG-SX, which is proven to be correct but is not deadlock-free. Anyway, this version has not been published yet, and we only considered then the published one.

concurrency control method for KBMSs should have. In summary, it lacks the support to the following important aspects:

1. 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 as a single lockable unit. Particularly with respect to the aggregates, the same previously mentioned performance problems may arise.
2. Finding dominators and entry points may be an easy task using the algorithms of [ASU85] and bit-vectors, but maintaining these structures may not be an easy one. Furthermore, aborting a transaction whenever it needs to lock a node not dominated by the first one it locked, may be very costly depending on the transaction, and may also provoke a great number of aborts and lost work.
3. Not allowing any concurrency within cycles may also show very poor performance if cycles are likely to be frequent and involving a lot of objects.
4. Lock conflicting modes should be defined. It makes no sense to prohibit two read transactions to be performed concurrently. This may become a bottleneck for very frequently queried KBs.
5. Phantoms are not considered by the DDG policy. To avoid phantoms would require the DDG policy to lock the index entry of the node that is being inserted, in addition to the node itself. Such attitude is not taken by this policy, and so phantoms may happen.
6. No kind of implicit locks is defined [21]. Thus, using the DDG protocol, to lock a class with thousands of instances, thousands of locks will be necessary. This high locking cost may lead the DDG policy to its inapplicability for KBMSs.
7. Furthermore, suppose we have a class C with thousands of instances. If we want to lock an instance of C, first of all we must lock C, and then proceed locking the instance. However, while we are holding the lock on C, we are avoiding the concurrent access to all other instances of it (see the last example). Moreover, if we need later on to lock another instance of C, we must hold the lock on C until this time comes (remember that each node can be locked at most once), avoiding so the concurrent access on C and on all its thousands of instances during this period of time [22]. This may be a fatal problem to the performance of the DDG policy, because it just does not allow the access to the instances of some class to be shared among multiple users.

6. Conclusions and Future Work

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 more and more be applied

[21] Although implicit locks may wrongly work on DAGs (see Sect. 5.3), if appropriate heeds are not taken, they must be considered due to their nice property of significantly reducing the number of locks to be set by a transaction. The reader is asked to see the proposed locking mechanism of ORION [GK88, KBCGW87, KBCGW89, KBG89, Ki90], which can avoid the malfunction of implicit locks working on structures other than hierarchies, and still provide their nice property.

[22] It is not so difficult to apply the locking rules of the granular lock protocol to this situation, and to verify that it shows much better performance than the DDG policy.

in real world applications. By this way, the research for concurrency control methods tailored to the KBMS environment plays a crucial role to this applicability.

However, knowledge sharing is not so simple as it may seem. An arbitrary interleaving of operations of different users on the same KB can lead to many inconsistencies. Some users may read dirty objects, and doing so inconsistently retrieve these objects. Others may have their updates lost, or even be unable to repeat a previously issued read. Furthermore, some objects may seem to appear and disappear, creating the so-called phantoms.

We analyzed some of the well-known concurrency control techniques, which have the main goal of synchronizing access to shared data, avoiding so the above problems, which arise when resources are concurrently accessed by multiple users. We investigated Two-Phase Locking, Predicate Locks, Granular Locks, and the Dynamic Directed Graph Protocol. The applicability of each one to the KBMS environment was discussed. We gave prominence to their adequacies and inadequacies, if they were to be directly applied to KBMSs. Some of them have shown to be better suited for specific situations, whereas malfunctioning and lacking of applicability to other ones.

Probably the main reason for their inapplicability to the KBMS environment as a whole is the inobservance of all aspects related to it, and which could help in improving performance and adaptability. In particular, we have the following aspects as important ones to be taken into account when designing a concurrency control method for KBMSs:

- The semantic knowledge of transactions should be thought of, allowing semantically consistent (although not necessarily serializable) schedules to be produced.
- Deadlocks avoidance is desirable, because detecting and resolving them consume resources.
- Phantoms should be avoided. They may cause transactions to produce incorrect results.
- The protocol should be as precise as it can, i.e., it should lock only the objects subject to anomalies, if concurrently accessed, in each particular case. Too pessimistic protocols may lock objects having nothing to do with the current situation.
- Different types of lock conflicting modes should be taken into account.
- Facilities to work with graphs is also desired, because graphs are a natural way to structurally represent KBs.
- Implicit locks should be considered, and the appropriate heeds to their correct behavior should be taken. They may significantly reduce the number of locks to be set by the transactions, increasing by this way the overall performance.
- A dynamic mechanism to change the granularity of locks according to each specific situation should be designed, e.g., to give support for lock escalation.
- Lock conversions should also be considered, meaning that a transaction should be able to upgrade (downgrade) its locks to more restrictive (less restrictive) ones.
- Cycles in the underlying KB structure should be dealt with. Clearly, allowing concurrency within cycles is also desired.

- The protocol should allow the transactions to be open-ended, and by this way to acquire locks on demand.
- Different abstraction relationships should be considered, allowing, e.g., aggregates to be handled as a single lockable unit.

Of course, meeting all these requirements may not be so easy, and who knows probably even impossible. Above all, the two basic laws of concurrency control should be obeyed [GR93]:

1. *Concurrent execution should not cause application programs to malfunction, and*
2. *Concurrent execution should not have lower throughput or much higher response times than serial execution.*

Moreover, the fundamental attribute of each successful concurrency control mechanism implementation is performance. Thus, such implementation must take into account whatever aspect it can to improve throughput and decrease response time.

As a future work, we are interested in investigating some concurrency control mechanisms implemented in some object-oriented database systems (e.g., ORION [GK88], O_2 [BDK92], IRIS [FBCCC+87], etc.). Such investigation should make clear the limits between desired aspects and implementable ones. Later on, we will start designing a concurrency control mechanism tailored to KBMSs, using as a practical example our prototypical KBMS implemented at the University of Kaiserslautern, named KRISYS (Knowledge Representation and Inference System).

Acknowledgments

This work has been supported by the CNPq (Scientific and Technological Development National Council), of the Secretary for Science and Technology of Brazil. Prof. Dr. Theo Härder read a previous version of this paper, and gave important contributions which led to clarifications of the exposed topics and critics. Joachim Thomas and Clarence Huff also read a previous version of this paper and together with Stefan Deßloch, they provided helpful comments on the elaboration of it. Finally, Vinay Chaudhri patiently explained me via e-mail many doubts I had about their DDG policy. I would like to acknowledge all of them.

References

[ASU85] Aho, A.V., Sethi, R., Ullman, J.D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Massachusetts, USA, 1985.

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

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

[BM86] Brodie, M., Mylopoulos, J. (Eds.). *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*. Springer Verlag, 1986. (Topics in Information Systems).

[BSW79] Bernstein, P.A., Shipman, D.W., Wong, W.S. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, Vol. 5, No. 3, May 1979, pp. 203-216.

[Ch91] Chaudhri, V.K. *Designing Multi-User Knowledge Base Management Systems*. Internal Report, University of Toronto, Toronto, Canada, Feb. 1991.

[Ch93] Chaudhri, V.K. *Communication via Internet*, May 1993. (e-mail: vinay@cs.toronto.edu).

[CHM92] Chaudhri, V.K., Hadzilacos, V., Mylopoulos, J. Concurrency Control for Knowledge Bases. In: *Proc. of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA, USA, 1992, pp. 762-773.

[Co70] Codd, E.F. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, Vol. 13, No. 6, June 1970, pp. 377-387.

[Da73] Davies, C.T. Recovery Semantics for a DB/DC System. In: *Proc. of the ACM 73 National Conference*, Atlanta, GA, USA, Aug. 1973, pp. 136-141.

[Da78] Davies, C.T. Data Processing Spheres of Control. *IBM Systems Journal*, Vol. 17, No. 2, 1978, pp. 179-198.

[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.

[FBCCC+87] Fishman, D.H., Beech, D., Cate, H.P., Chow, E.C., Connors, T., Davis, J.W., Derret, N., Hoch, C.G., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M.A., Ryan, T.A., Shan, M.C. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, Jan. 1987, pp. 48-69.

[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 International Conference 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).

[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 International Conference on Object Oriented Programming Systems, Languages and Applications*, Orlando, FL, 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 International Conference on the Management of Data*, Portland, Oregon, USA, 1989, pp. 337-347. ACM SIGMOD Record, Vol. 18, No. 2, June 1989.

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

[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 International Conference on Entity-Relationship Approach*, Rom, Italy, Nov. 1988, pp. 331-350.

[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 in Information Systems and Artificial Intelligence*, Ulm, Germany, March 1990.

[Ma91] Mattos, N.M. *An Approach to Knowledge Base Management*. Springer Verlag, Berlin, Germany, 1991. (Lecture Notes in Artificial Intelligence Vol. 513).

[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.

[Mo85] Moss, J.E.B. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, USA, 1985. (Series in Information Systems).

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

[Pa79] Papadimitriou, C.H. Serializability of Concurrent Database Updates. *Journal of the ACM*, Vol. 26, No. 4, Oct. 1979, pp. 631-653.

[Re92] Rezende, F.F. *A Transaction Model to Support the World Concept of KRISYS*. Federal University of Rio Grande do Sul, Porto Alegre, RS, Brazil, May 1992. (Master Dissertation).

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

[Th91] Thomas, J. *An Approach to the Representation of Worlds and Viewpoints in the KBMS KRISYS* (in German). University of Kaiserslautern, Kaiserslautern, Germany, June 1991. (Diplomarbeit).

[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.