Working Papers of the Institute of Business Informatics

Editors: Prof. Dr. J. Becker, Prof. Dr. H. L. Grob, Prof. Dr. K. Kurbel,
Prof. Dr. U. Müller-Funk, Prof. Dr. R. Unland, Prof. Dr. G. Vossen

Working Paper No. 31

Semantics-Based Locking: From Isolation to Co-operation

Rainer Unland

University of Münster, Institute of Business Informatics
Grevener Str. 91, D-48159 Münster, Germany, Tel. (0251) 83-9750, Fax (0251) 83-9754

März 1994

Contents

Abstract

'Advanced database applications', such as CAD/CAM, CASE, large AI applications or image and voice processing, place demands on transaction management which differ substantially from those of traditional database applications. In particular, there is a need to support 'enriched' data models (which include, for example, complex objects or version and configuration management), 'synergistic' cooperative work, and application- or user-supported consistency. This paper deals with a subset of these problems. It develops a methodology for implementing semantics-based concurrency control on the basis of ordinary locking. More specifically, it will be shown how conventional locking can step by step be improved and refined to finally reach our initial goal, namely a comprehensive support of synergistic cooperative work by the exploitation of application-specific semantics.

In addition to the 'conventional' binding of locks to transactions we consider the binding of locks to objects (object related) and subjects (subject related locks). Object related locks can define persistent and adaptable access restrictions on objects. This permits, among others, the modeling of different types of version models (time versions, version graphs) as well as library (standard) objects. Subject related locks are bound to subjects (user, application, etc.) and can be used among others to supervise or direct the transfer of objects between transactions.

Keywords: *transaction management; non-standard applications; semantics-based concurrency control; nested transactions; synergistic cooperative work*

1 Introduction

The conventional model of transactions as presented by [Gray79] is based on the ACID principle ([HäRe83]) which, in short, means that a transaction is an atomic unit of reads and writes against a database. To support **a**tomicity and **i**solated execution a transaction is not allowed to see changes performed by a concurrent transaction. Additionally, the all or nothing principle must be obeyed. Either all changes are reflected in the database or none. As long as transactions process within a few seconds this model is suitable. Because transactions are shortliving little work will be lost in case of a backup. Additionally, concurrent transactions have to wait only for a very short time if they are blocked due to a conflict with a concurrent transaction.

Non-standard applications, like CAD/CAM or software development, however, have totally different requirements. Typically, transactions are interactive and of long duration. Most of the work performed by a user, for example, a designer, is only expressed within this transaction and documented nowhere else. Therefore, of course, long waits or rollback in case of a transaction or system failure cannot be accepted. Another major requirement of most non-standard applications is a far-reaching support for synergistic, cooperative work. Usually tasks are much to complex to be solvable by a single user. Instead, groups of users have to cooperatively work together to solve the problem. Of course, this means that they need comprehensive support for teamwork.

Traditional concurrency control, as well as all ACID-based proposals fail to adequately support cooperative-work since they cannot consider (application-)specific semantics of operations. However, cooperative work as well as concurrency could substantially be enhanced through the use of these semantic information about operations. A common approach to the integration of application-specific semantics into concurrency control is semantics-based concurrency control. However, most approaches to this more expressive form of concurrency control have the common disadvantage that they tend to be rather complex and error-prone. This paper will present a strategy which evolves from a well-known, but more conventional technique. We will show that the locking approach can be adapted and enhanced in a way that it is able to capture and express application specific semantics. Indeed, our approach can be used as a low level technique to implement semantics-based concurrency control. Furthermore,

it will be shown how the proposed approach can be utilized to realize cooperative environments within nested transactions. While within such an environment synergistic cooperative work and creativity can fully be accomplished this environment, nevertheless, can perfectly be shielded from and to the outside.

In addition to the 'conventional' binding of locks to transactions we consider the binding of locks to objects (object related) and subjects (subject related locks). Object related locks can define persistent and adaptable access restrictions on objects. This permits, among others, the modeling of different types of version models (time versions, version graphs) as well as library (standard) objects. Subject related locks are bound to subjects (user, application, etc.) and can be used among others to supervise or direct the transfer of objects between transactions.

Comparison with semantics-based concurrency control

In object-centered database systems objects are usually treated as instances of abstract data types. An abstract data type is characterized by a set of specified operations which represent the only way in which a user can access and manipulate the instances of that type. Since these operations are treated as part of the database their semantics can be exploited to achieve greater concurrency or to permit a more cooperative style of work in specific environments. A common approach to the integration of application-specific semantics into concurrency control is **type-specific** or **semantics-based concurrency control** (c. f., [BaRa90], [BÖHG92], [ChRR91], [Garc83], [HeWe88], [SkZd89], [SpSc84], [Weih88]). With semantics-based concurrency control controlling the concurrent execution of transactions involves the control of execution of the operations invoked on the objects. Whether or not two operations, invoked by different transactions, can be allowed to execute concurrently depends on the effect of one operation on the other, and the effect of the operations on the object. This stands in contrast to conventional database systems. They define *read* and *write* operations as the level of abstraction at which application programs can interact with the database. As a consequence, serializability theory has produced algorithms that are cast in terms of the semantics of reading and writing; i.e., whatever the semantics of operations is they are mapped on a common basis, namely read and write accesses to the database or, more generally, on the set of lock modes provided by the concurrency control scheme. Of course, this mapping entails that higher level semantics of operations cannot be exploited. A common example is a bank account. If two

concurrent transactions each add an amount to the same account, this can be done in an interleaved way since these operations are *commutative*. Nevertheless, the conventional locking scheme will not permit such an interleaving since both operations are update operations and update operations are not compatible.

Semantics-based concurrency control, however, performs on a semantically higher level than reading and writing. Therefore, it allows the definition of weaker and more flexible notions of conflict among operations. For instance, operations invoked by two transactions can be interleaved as if they commuted, if the semantics of the application allows the dependencies between the transactions to be ignored. The incorporation of general or state-dependent commutativity in the conflict definition between operations is a first solution. A more sophisticated approach is to substitute commutativity by the more general concept of compatibility ([Garc83], [SkZd89]). **Compatibility** allows two operations to be treated as compatible if their execution order is insignificant from the application's point of view (even if they are not commutative). Clearly, semantics-based concurrency control will, in general, not guarantee serializability. However, it nevertheless preserves consistency. At first glance this seems to be an attractive means for increasing performance in a complex information system. However, unfortunately, there are also some drawbacks asso-

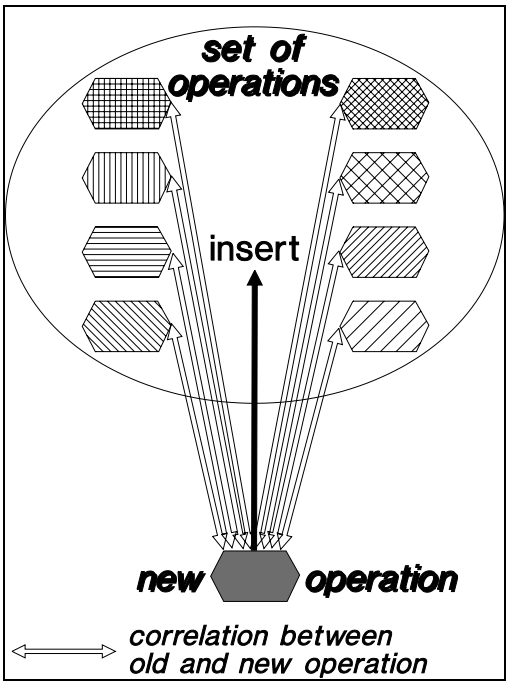

Figure 1.1: Semantics-based Concurrency Control

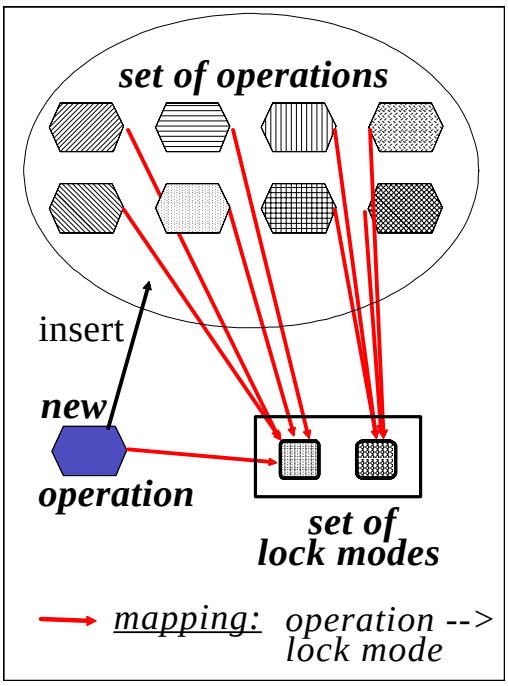

Figure 1.2: Traditional Concurrency Control

ciated with this approach. The most important is that it eliminates the common and neutral basis of the locking approach. If, for instance, a new operation is to be integrated which displays all transactions on an account which took place in a given period of time, this operation cannot just be mapped on the read operation of the database. Instead, all other operations on the abstract data type *Account* need to be considered to decide what kind of concurrency control measures are to be implemented (see Figure 1.1). In other words, conventional concurrency control provides a common and static basis (a number of lock modes) on which each operation of an application can be mapped (see Figure 1.2). Everything else is left to the concurrency control scheme.

In semantics-based concurrency control such a common basis is not existent. Instead, the compatibility of operations has to be defined on the level of the operations. This level is neither static (it may increase or decrease due to the insertion or deletion of new operations) nor is its semantics directly visible to the programmer. Instead, it is hidden in the implementation of the operations. Therefore, an insertion of a new operation is highly complex and error-prone. An operation like DEPOSIT of transaction T_1 will not conflict with a WITHDRAW operation of a concurrent transaction T_2 as long as both operations are the only operations of T_1 and T_2. But, if T_1 also wants to perform a WITHDRAW operation this operation may be rejected since it may violate a constraint (for example, the balance must not be negative) which would not have been violated had the WITHDRAW operation of T_2 not been permitted to execute in the meantime. The difficulty here comes from the fact that more than one operation is performed within one transaction (instead of treating an operation as an equivalent to a transaction).

Our approach provides another solution for the inclusion of application-specific semantics into concurrency control. It provides an enhanced set of basic lock modes as the common basis of all operations. These lock modes are not only finer grained but they can also individually be adapted to the requirements of the operations. Even more flexibility is gained by the fact, that we allow rules to be bound to locks. This permits an individual reaction on lock requests; i.e., the system can react differently on different users.

We have in common with the concept of semantics-based concurrency control that we think of an application layer as a layer which leans on the concept of abstract data type. Therefore, we do not want the user to explicitly handle locks. Instead, we assume that the handling of locks is

hidden within the implementation of the operations. Moreover, operations on objects of the database can only be performed within transactions.

Other related work:

Advanced transaction models have been developed to support non-standard applications. However, most earlier approaches (cf. [HaLo81], [HäRo87], [LoPl83], [KLMP84], [KoKB85], [KSUW85]) concentrate first of all on modularity, failure handling, and concurrent execution of subtasks. Cooperative work is usually only supported on a low level since these approaches basically insist on strict serializability or weaken strict serializability only in special situations, for example, to facilitate a controlled lending, transfer, or exchange of objects.

Some more recent proposals, however, also provide means for a comprehensive support of cooperative work.

The **Cooperative Transaction Hierarchy** or **Transaction Groups** concept ([SkZd89], [NoZd90], [NoRZ92]) defines a nested framework for cooperating transactions in a design environment. The Transaction Groups model structures a cooperative application as a rooted tree called a *cooperative transaction hierarchy*. The leaves of the hierarchy represent the transactions associated with the individual designers, called **cooperative transactions**. The internal nodes are the **transaction groups**. Each transaction group contains a set of members that cooperate to execute a single task. It actively controls the interaction of its cooperating members. Cooperative transactions need not to be serializable; instead the transaction group of the cooperative transactions defines a set of rules that regulate the way the cooperative transactions should interact with each other; for instance, within a transaction group, member transactions and subgroups are synchronized according to some semantic correctness criteria appropriate for the application. The criteria are specified by a set of *active patterns* and *conflicts*. **Conflicts** are like locks in the sense that they specify when certain operations cannot occur. **Patterns** specify operation sequences that must occur in a history for it to be correct. Patterns and conflicts in a transaction group are specified by LR(0) grammars. The observance of the rules is enforced by a recognizer and conflict detector, which must be constructed for each application. The implementation of transaction groups is supported by replacing classical locks with non-restrictive *lock mode, communication mode* pairs. The lock mode indicates whether

the transaction intends to read or write the object and whether it permits reading while another transaction writes, writing while other transactions reads and multiple writers of the same object. The communication mode specifies whether the transaction wants to be notified if another transaction needs the object or if another transaction has updated the object. Transaction groups and the associated locking mechanism provide suitable low-level primitives for implementing a variety of extended transaction models.

As our approach the model of cooperative transaction hierarchies places particular emphasis on the support of cooperative work. However, their approach is similar to semantics-based concurrency control which means that it has the same disadvantages. They extend this approach by supporting the specification of sequences of operations, a concept which is not volunteered by our approach. The low level lock modes are a subset of the lock modes provided by our approach and the concept of communication mode is covered by the rules and constraints concept.

The **Semantic Locking for EVEnt Synchronisation** concept ([Skar93]) develops a methodology for implementing semantic synchronization specifications with ordinary *Read* (Share) and *Write* (Exclusive) locks, and it describes a software system SLEVE that implements the methodology. An application developer who defines a conflict relation over an arbitrary set of synchronization primitives, such as abstract type operations, events, or semantic lock types, can use SLEVE to implement the specification on an underlying system that supports only Read/Write locking. Their approach is similar to our one in that they try to implement semantics-based concurrency control on top of locking. However, they assume conventional locking as the basis of implementation, while we first substantially extend the set of lock modes and its semantics and furthermore allow rules to be bound to locks. In this our approach is much more flexible, however, requires a deep integration into the database system (at least for an efficient realization) while the approach of Skarra can be realized on top of an existing database system.

The **Database Conversations** concept [KLRW94] was developed to explicitly support cooperative problem solving. Conversations embrace multi-party modifications of common data in such a way that transactional consistency is guaranteed. Similar to check-out/check-in, a requestor accessing data concurrently being modified is not blocked. Rather, the requestor is notified about the modification (they call it conversational state). Opposed to check-out/check-in, the requestor is solicitated to contribute to the modification. Such conversations are best

understood as a tight framework for joint data modifications in contexts larger than single transactions without creating any commit or abort dependencies between transactions. This is because the semantics of the framework is assumed to be known by the participants of conversations. This is a rather interesting approach to cooperative problem solving. However, it is especially tailored to such applications and, therefore, less attractive for other forms of cooperation (like design applications). Moreover, it puts a lot of responsibility for a correct and consistent work on data on the user which again restricts its general usage.

The remainder of this paper is organized as follows. Section 2 will introduce a set of flexible and adaptable lock modes. Moreover, these locks will be further analyzed and synthesized in order to be able to express more semantics. In section 3 it will be shown how this new view on locks can be exploited within nested transaction. Especially, it will be shown how synergistic cooperative work can massively be supported while, nevertheless, being able to isolate a cooperative environment from other environments within the nested transaction. Section 4 introduces rules and notification services which can be bound to locks. As will be shown this is a major step in the direction of semantics-based locking. Indeed, the resulting locking scheme can even be used as an implementation strategy for semantics-based concurrency control. Section 5 presents a solution for the treatment of special objects (as standard or library objects). In section 6 it will be shown how objects can safely be transferred from one application to the next. Finally, section 7 concludes this paper.

This work is part of a larger projects which aims at the development of a flexible and adaptable tool kit approach for transaction management. The tool kit is meant to allow a sophisticated applications designer or database implementor to develop individual, application-specific transaction managers for all kinds of non-standard applications. Much attention was paid to a comprehensive support of different facets of cooperative work.

The main characteristics of the tool kit are:

1. It supports the definition of a large number of different *transaction types*. These transaction types are meant to reflect the requirements of different application areas. For this reason, a basic set of characteristics are identified by which transaction types may differ from each other. However, since the tool kit is extensible this set can be augmented if additional demands need to be met.

2. The different transaction types can be combined with each other in every hierarchical order to form a *heterogeneously structured transaction hierarchy* which is capable of supporting such different concepts as strict isolation of (sub)transactions (in the sense of serializability) and (non-serializable) cooperative work in one hierarchy. For this reason a general set of rules was proposed which has to be obeyed by each transaction type.

Interested readers are referred to [Unla91] and [UnSc92] for further information.

2 Lock modes

Conventional database systems provide two lock modes namely the exclusive or X- and the shared or S- lock mode (hierarchical locks are not considered here). Since transactions in these environments are usually short-lived, the two modes are sufficient. Advanced database applications, however, behave completely different. Transactions are typically interactive and of long-duration which means that objects need to be locked for a substantially longer period of time. Therefore, it is essential that a lock mode fits as exactly as possible to the operations which will be executed on the object.

Example 2.1:

Let us consider a CASE environment in which a number of software engineers work on the development of some software package. The following situations may occur. A programmer wants to implement a module from which he knows that a similar one was already implemented some time ago. He wants to use this module as a model. Moreover, he needs the currently valid specification of another module since he wants to use it in his program. In both cases the object has to be read, however, with different semantics. In the first case it is of minor relevance for the programmer whether the module is currently being modified by a concurrent transaction. In the second case, however, the read needs to be a consistent read.

Another situation may be a group of programmers that work cooperatively on the implementation of a program. Here it is rather desirable that a programmer of the group is allowed to read each of the commonly used modules even if this module is still under devel-

opment. On the other hand, people who do not belong to the group should not be in a position to read modules which are still under development.

As the above example clearly indicates a proper lock technique needs to consider the intention (semantics) of an operation as well as the environment within which the operation is meant to be executed.

2.1 Extended set of lock modes

In this section we will introduce an extended set of basic lock modes which can be regarded as inevitable in the context of most non-standard applications. We assume that updates are not directly performed on the data of the database but that some form of shadowing is realized.

Of course, the shared (S-) and exclusive (X-) lock will remain useful for all kinds of database applications.

shared lock (S-lock): only permits reading of the object

exclusive lock (X-lock): permits reading, modification, and deletion of the object.

Sometimes an application or user is just interested in the existence of an object but not in its concrete realization. For example, if a programmer wants to integrate an already existing procedure into 'his' software package he may only be interested in the existence of the procedure but not in its actual implementation. Therefore, from the user's point a view it is irrelevant whether the procedure will be modified concurrently (at least as long as the modification stays within some limits (for example, no modification of the interface of the procedure)). Such a demand can be satisfied if a lock mode is provided which permits the modification of an object but not its deletion. This leads to an update lock (U-lock):

update lock (U-lock): permits reading and modification of the object.

Another often mentioned requirement is a dirty read (see example 2.1) or browse which allows the user to read an object irrespective of any lock currently granted for that object:

browse lock (B-lock): permits browsing of the object (dirty read)

Especially design applications often want to handle several states of an object instead of one; i. e., in such environments an object is represented by its version graph. In [KSUW85] it was shown that the conventional S-/X-lock scheme is not sufficient in such an environment since the derivation of a new version corresponds not only to an insert operation (of the new version of the object) but also to an update operation (the version graph of the object is modified). Thus, it is desirable to enable a transaction to exclude others from simultaneously deriving a new version from a given version *v*; other transactions may only read *v*. Therefore, we include a further lock mode:

derivation lock (D-lock): permits reading of the object and the derivation of a new version of the object.

Although this extended set of lock modes clearly allows the applications designer to capture more semantics it is still on a rather coarse level. Especially cooperative environments can hardly be supported. The next section presents a more flexible and convenient solution.

2.2 The two effects of a lock

Since lock protocols rely on conflict avoidance they regulate access to data in a relatively rigid way. As a matter of principle, a transaction, first of all, has no rights at all on data of the data-base Such privileges can only be acquired via an explicit request for and assignment of locks. In the remainder, we will distinguish between an **owner of a lock** (**owner** for short) and a **competitor for a lock** (**competitor** for short). An owner already possesses some lock on an object O whereas a competitor is each concurrent transaction, in particular each transaction that competes for a lock on O.

If we analyze the semantics of a lock, it becomes clear that a lock on an object O has always two effects:

1. it allows the owner to perform certain operations on O and
2. it restricts competitors in their possibilities to work on O.

This decomposition of the semantics of a lock makes it possible to differentiate between the rights which are assigned to the owner of a lock and the restrictions which are imposed on competitors. From now on, No. 1. will be called the **internal effect** of a lock request while No. 2. will be called the **external effect**.

Example 2.2:

An X-lock has the internal effect in that it allows the owner to read, modify, and delete the locked object. The external effect ensures that competitors cannot lock the object in whatever mode.

An S-lock has the same internal and external effect since it allows the owner (internal effect) as well as competitors (external effect) to just read the object.

This distinction between the internal and the external effect of a lock makes it possible to establish the rights of an owner without simultaneously and automatically stipulating the limitations imposed on concurrent applications. We gain the freedom to determine the external effect of a lock individually.

The lock modes which were discussed in the previous section come with the following internal effects:

exclusive lock (X-lock): permits reading, derivation of a new version, modification, and deletion of the object.

update lock (U-lock): permits reading, derivation of a new version, and modification of the object (not its deletion).

derivation lock (D-lock): permits reading and derivation of a new version of the object (not its deletion or modification). This lock mode is only useful if the data model supports a version mechanism.

shared lock (S-lock): permits reading of the object (neither its deletion or modification nor the derivation of a new version).

browse lock (B-lock): permits reading of the object in a dirty mode (neither the consistent reading, modification, or deletion of the object nor the derivation of a new version).

The examination of the external effect leaves some leeway for further discussion. Conventional database systems enforce the operational integrity to be entirely ensured by the database management system. To be able to support the needs of advanced database applications, however, it is inevitable to weaken this rigid view; i.e., to transmit some responsibility for the correct and consistent processing of data from the database system to the application. Especially, in design environments users want to work on data in a way which does not automatically guarantee serializability (cooperative work). But, of course, the database system has to ensure that concurrent work on data can preserve consistency as long as the applications take care of their part in consistency control. In this sense, an update and a read operation on the same object may be compatible as long as the reader is aware of the concurrent updater. A simultaneous modification of the same object by different transactions is usually prohibited, at least as long as the data is handled by the system as an atomic unit. However, if concurrent application are capable of merging the different states of an object before it is checked in on the next higher level, concurrent updates can also be permitted.

In order to be able to precisely describe a lock mode in the remainder it is necessary to specify the internal effect as well as the external effect. Therefore, **X/Y** denotes a lock which combines the internal effect X with the external effect Y.

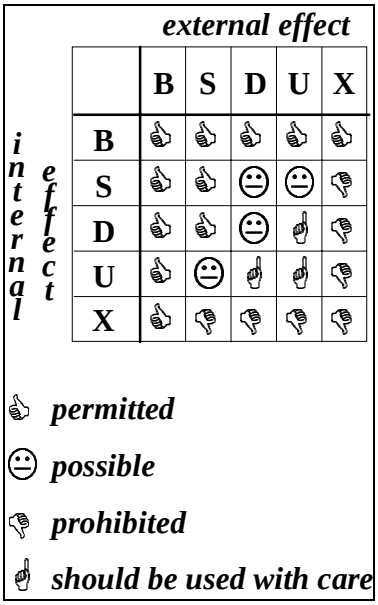

Table 2.1: Compatibility matrix internal/external effects

Table 2.1 lays down which internal effect can be combined with which external effect. A 👍 (👎) indicates that the given internal effect can always (never) be combined with the corresponding external effect. A 😐 signifies that the validity of such a combination should depend on the design and abilities of the application. If the application is prepared to accept some responsibility for the consistent processing of data a 😐 can be replaced by a 👍 (for example, to model cooperative environments). However, the sequence of 👍 in a row must be continuous; for example, in case an S/U lock is permitted an S/D must be a possible lock too. A ☝ corresponds to a 😐, but indicates that this combination should be used with care. It especially requires measures which assure that the concurrent modifications of the data cannot disturb the consistency of the data(base).

2.3 The Semantics of the Lock Modes

Since the internal effect **B** does not impose any restrictions on competitors, it is not a lock in the literal sense of the word. It neither requires an entry in the lock table nor a test whether the object is locked. Since we assume that updates are not directly performed on the original object (but on a copy), a B-lock guarantees that the state of the object to be read is either still valid or was valid at some period in the past (*validity interval*). However, if an application acquires several B-locks for different objects there is no guarantee that the validity intervals of these objects do overlap.

As an external effect, the **B-lock** is the strongest choice, since it does not allow any concurrent application to access the locked object in any form. However, the B-lock still allows concurrent applications to browse the object. By that, applications can be supported, which require objects to be generally "browsable" (dirty read), regardless of whether they are locked. If a dirty read is to be prohibited this can also be modeled. Since we consider our approach as to provide the basis for the definition of semantically richer operations (for example, in the sense of

object-oriented methods), the browsing of objects can be prevented by simply not offering such a method.

An internal effect **S** requires a compatibility check and an entry in the lock table, since it prevents competitors from acquiring at least an X-lock. Since an S/U-lock is compatible with a U/S-lock, it may also realize some kind of dirty read. However, in contrast to the B-lock a read is only possible if the updater as well as the reader agree to it. Therefore, the S/U (U/S) lock is supposed to be used especially in cooperative environments.

An internal effect **U** permits the modification of the object. A concurrent S-lock can be prohibited. An X-lock, additionally, grants the owner the right to delete the object. Since this is the only difference between these two lock modes, the X-lock is meant to be used only in case the object will be deleted. With this interpretation in mind it becomes clear why an X-lock prohibits a concurrent S-lock. Since the object will most probably be deleted, a read operation makes no sense.

2.4 A short discussion of consistency aspects

The main motive for the introduction of our approach is that we want to provide a basis for a skillful applications designer which allows him to satisfy the demands of an application in a proper way. This requirement can only be met if the tool kit enables a designer to transfer some of the responsibility for consistency from the system to the application. Only by such a transfer, for example, non-serializable cooperative work can be supported. On the other hand, our tool kit also supports consistency level 3 ([GLPT76]) if all 😐 (and ☝) are replaced by a 👎 and, additionally, the B-lock is not used.

2.5 Dynamic Assignment of an External Effect (Open Lock)

In addition to the possibility of fixing the external effect when the lock is acquired, it is also possible to leave it open for the moment and fix it at a later time. In this case, the system assumes the strongest external effect, as a default. If, however, a conflict arises which can be solved by a weaker external effect the owner is asked whether he accepts this weaker external

effect. This allows the owner to decide individually whether he wants to accept concurrent work on the object. Such a decision may depend on the competitor's profile or the current state of the object. A lock with a fixed external effect is called **fixed lock,** while a lock with an undecided external effect is called **open lock**.

2.6 Upgrading a lock

In order to be able to decide whether a given lock is stronger than another one we first need to define whether a given (internal/external) effect is stronger than another one.

Definition 2.1:

An internal effect is **stronger (weaker)** than another one if it concedes

- ☞ *more* (*fewer*) rights on the locked object to the owner.

According to this definition, the internal effect *B is the weakest* one while *X is the strongest* one; i. e., the internal effects increase from B to X. The external effect, however, lays down which lock modes can still be granted to competitors.

Definition 2.2:

Therefore, an **external effect** is **stronger (weaker)** than another one if it concedes

- ☞ *fewer* (*more*) rights to a competitor.

Since the external effect *X* still allows a competitor to acquire each internal effect, it is the *weakest* one while *B* is the *strongest* one (it only allows competitors to read the object in a dirty mode); i.e., the external effects increase from X to B.

In the previous section it was said that a 👌 in table 2.1 indicates that the given internal effect can be combined with the external effect which goes with it without risk. However, in case of a new request X/Y it is not only necessary that X is compatible with each already granted external effect. Moreover, Y must be compatible with each already granted internal effect as the following example shows:

Example 2.3:

A D/D-lock allows competitors to concurrently create their own new version of the given object. However, a conflict will arise if a competitor wants to acquire a D/S lock. Since this lock would exclude others from concurrently deriving a new version a D/S lock cannot be granted in case a D/D was already granted.

To avoid situations like the one in example 2.3 we need to define the compatibility of locks:

Definition 2.3:

Two locks are **compatible**,

1. if the owner's external effect permits the internal effect of the competitor
2. if the competitor's external effect permits the internal effect of the owner.

Example 2.4:

Let us assume that an S/U-lock was already granted to user P. If competitor C wants to acquire a D/S-lock, this request can be granted. The external effect of P's lock permits the internal effect of C's lock (condition 1: a D is weaker than a U). Moreover, the external effect of C's lock permits the internal effect of P's lock (condition 2: an S is compatible with an S).

In order to support the upgrade of a lock mode we need to define under which circumstances a lock is stronger (weaker) than another one.

Definition 2.4:

A lock L1 is **stronger** (**weaker**) than a lock L2, if

1. the *internal* effect of L1 is at least (*at most*) *as strong as* the *internal* effect of L2,
2. the *external* effect of L1 is at least (*at most*) *as strong as* the *external* effect of L2,
3. L1 is different from L2.

Example 2.5:

Since the internal effect U is stronger than the internal effect D and the external effect B is stronger than the external effect S the U/B lock is weaker than the D/S lock.

For the following discussion we will assume that each ☝ in table 2.1 is replaced by a 👎 since most applications will not be able to guarantee consistency in case of concurrent modifications on the same object. This assumption is no real restriction since the following discussion can easily be transferred to the extended compatibility matrix.

On the basis of the above definition, the different lock modes, which can be inferred from the (restricted) compatibility matrix (table 2.1), can be arranged in a linear order (according to their strength):

$$(B/X) \rightarrow (S/U) \rightarrow (S/D) \rightarrow \begin{pmatrix} S/S \\ D/D \end{pmatrix} \rightarrow (D/S) \rightarrow (U/S) \rightarrow (U/B) \rightarrow (X/B)$$

Exceptions are the exclusive read lock (S/S-lock) and the shared derivation lock (D/D-lock). The S/S-lock has a stronger external effect than the D/D-lock but a weaker internal effect. If an owner of an S/S lock wants to switch to a D/D lock (or vice versa) he needs to acquire the next stronger lock (the D/S-lock).

3 Locks in the Context of Nested Transactions

In this section it will be shown how the decomposition of a lock mode can be exploited by the concurrency control scheme in order to support a more cooperative style of work. Consider the transaction/application hierarchy of figure 3.1. Let us assume that transaction T5 has acquired some object O in lock mode U/B from its parent (not visible in the figure). Now it wants some work on O be done by its child T10. To do so, T5 has to transmit the object/lock pair O/(U/B) to transaction T10. This leads to the following situation:

1. O is locked on the level of T5 in lock mode U/B.
2. O is available to the descendant tree of T10 in lock mode U/B.

Feature 1 is a necessary restriction, since it prevents other children of T5 (as well as T5 itself) from modifying O. Feature 2, however, is an unnecessary obstacle to the task of T10 for the following reason:

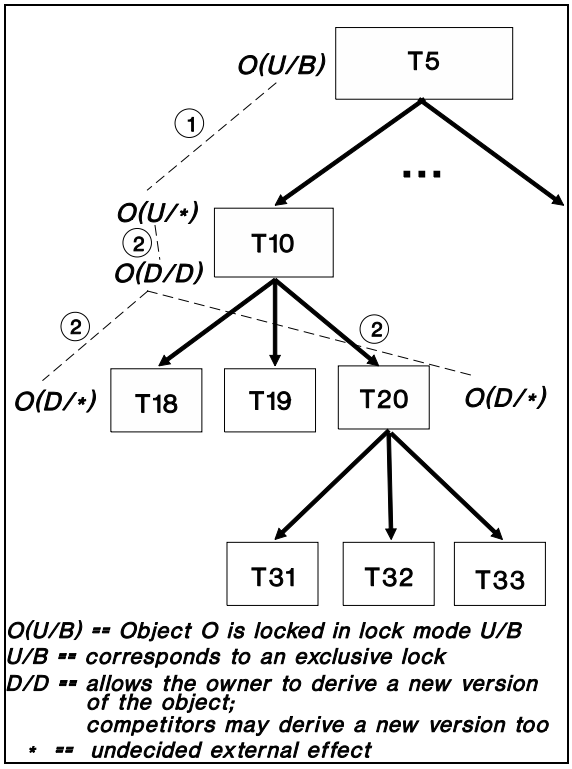

Figure 3.1: Acquisition of an object from the parent

T5 is only interested in the results of the work of T10 on O, but not in how these results will be achieved. T10 needs O and the permission to work on O in a way which corresponds to its task. However, it should be left to T10 to decide how the work on O can be performed best. For example, if T10 decides to develop several alternatives of O simultaneously, for example in T18 and T20, and to select afterwards the best alternative, such a proceeding should be permitted. In the scenario above this is prohibited, since the U/B lock on O prevents T18 and T20 from concurrently acquiring the necessary locks on O. If we take a closer look at the semantics of the subtask which T5 assigned to T10 it becomes clear that this task is sufficiently described by the object O and the internal effect of the lock on O.

However, the downward inheritance of the external effect of the lock is a useless obstacle, since it does not result in any advantage for T5. Instead, it unnecessarily restricts T10 in performing its task. Consequently we decided to transfer only the internal effect to the child transaction. Therefore, if T10 acquires an object O from T5 in mode U/B, this lock is set only on the level of T5 (see figure 3.1). T10 simply inherits the internal effect of the lock. The external effect is left undecided (this results in U/* (①)). T10 may allow its children to acquire every lock on O with an internal effect equal to or weaker than U and every external effect which T10 wants to concede to its children. In figure 3.1, T10 decides to allow its children to acquire concurrently O in D/D mode, therefore, to derive concurrently new versions from O (②) (a D/D lock allows the owner to derive a new version of the object; competitors may derive (concurrently) new versions, too). While this proceeding allows T10 to execute its work on O

autonomously, it does not allow T10, for example, to transmit several alternatives of O to T5, since O is locked on the level of T5 in mode U/B (which does not permit the derivation of several alternatives).

4 Rules on Locks and Notification Services

Synergistic cooperative work can only be supported adequately if applications can actively *control* the preservation of the consistency of data/objects. This, however, requires the concurrency control component to provide as much support to applications as possible. For example, our approach permits update operations to be compatible just by assigning the appropriate lock (U/U) to them. However, if concurrency control relied on pure locks, it would be too inflexible.

Example 4.1

Let us consider an abstract data type *PRICE-PRODUCT* on which the following four operations are defined (see Figure 4.1):

INCREASEVAT (op_1):	needs internal effect U
INCREASEPRICE (op_2):	needs internal effect U
COMPUTEPRICEINCLUSIVELYVAT (op_3):	needs internal effect S
COMPUTEPRICEEXCLUSIVELYVAT (op_4):	needs internal effect S

The compatibility of these operations (as shown in figure 4.1) cannot be completely modeled yet. The problem is that we can define that op_1 and op_2 should acquire a U/U lock (which means that both update operations can be performed concurrently (which, of course, is desirable)). op_4, however, should be compatible to op_1 which means that we must assign an S/U lock to op_4. This, however, implies that op_4 is compatible to op_2, too (which, of course, is not correct).

The problem is that locks as such define compatibility still on a too global level. An S/U lock is compatible with a U/U lock regardless of whether the second lock is acquired by an operation op_1 or op_2. If we really want to exploit the semantics of applications, the *all-or-none principle* (a lock allows either all concurrent transactions to access the object in a given mode or none) must be replaced by a more expressive *yes-if principle*, i.e., concurrent transactions are allowed to access the object if certain conditions are fulfilled. For this reason, our approach supports the binding of rules to locks. The rule mechanism is similar to ECA rules (event - condition - action rules, cf. [DaBM88]).

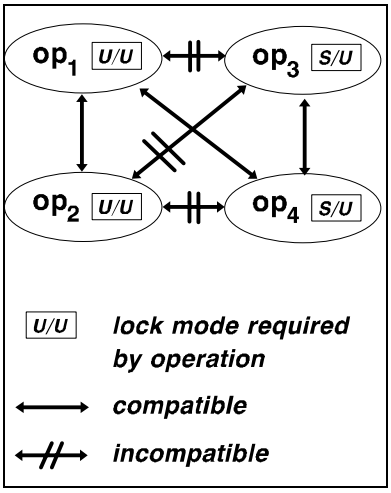

Figure 4.1: Relationship between operations

on event {[**case** condition] **do** [action1] [action2]}*

An ***Event*** can be an action which is triggered when an operation is performed on a(n):

lock: request / release / up- / downgrade / transmission (lending, transfer, return)

object: modification / transfer / deletion

transaction: begin / end / suspend / resume / (partial) rollback

Conditions can be

special users: a certain user / application / transaction has triggered the event

special operations: a certain operation has triggered the event

object states: the object is in a certain state (for example, compiled / tested/ etc.)

The condition specification is optional, which means that an event may directly trigger an action.

The ***action*** section can comprise two parts:

☞ In ***action1*** an exception can be expressed regarding the underlying lock. An exception can be *positive* or *negative*. With positive (negative) exception, a lock mode can be weakened

(tightened) by explicitly declaring which event can cause which kind of weakening (tightening) and under which circumstances.

☞ ***Action2*** allows the system to react on events by (additionally) sending messages. To be able to do so the rule mechanism is accompanied by a *notification service* by which applications/users/agents can be informed about certain facts or can be asked to do certain things.

The system will react differently on an event if different conditions and actions are specified in the **case do** part.

Example 4.2

a. negative exception

Given is an S/U lock on object O. The following negative exception tightens the lock:

on *lock-request*

{[**case** <predicate P_m>] **do** [**prohibit** (U/*);] [**notify-Request** <t_m>]}*

This condition specifies that if a concurrent application wants to acquire some lock with internal effect U on object with OID O (would allow the requesting process to update O), the request will be rejected if predicate P_m is true. The *notify-Request*(ing process) *clause* specifies that the message t_m will be sent to the requesting process.

b. positive exception

Given is an S/S lock on object O. The following positive exception weakens the lock:

on *lock-request*

{[**case** <predicate P_m>] **do** [**permit** (U/<B);] [**notify-Self** <t_m>]}

This condition specifies that if a concurrent application wants to acquire some lock with internal effect U, the request can be granted if the external effect of the requested lock is weaker than B (<B; for example, S or U) and predicate P_m is true. The *notify-Self clause* specifies that the message t_m will be sent to the owner of the S/S lock.

c. Solution for example 4.1

The scenario of figure 4.1 can be modeled as follows (it is assumed that both op_3 and op_4 will acquire an S/S lock on instances of *PRICE-PRODUCT*):

When op_1 is executed it must bind the following rule to its lock on the object dealt with:

1. **on** *lock-request*

   case op_3 **do** **prohibit** (U/>B); **notify-Request** "VAT is being modified"

When op_3 is executed it must bind the following rule to its lock on the object dealt with:

2. **on** *lock-request*

   case op_2 **do** **permit** (S/U)

Rule 1 assures that op_3 cannot acquire a lock on object O as long as op_1 holds its lock on O. However, if op_3 is the first operation to request a lock on O, then op_1 cannot concurrently acquire its lock on O, since the necessary U/U lock is not compatible with the already granted S/S lock (of operation op_3). Finally, if op_2 wants to acquire a lock on O, the lock can be granted since rule 2 permits this exception.

Similar rules must be installed for op_2 and op_4.

Another good example for the usefulness of rules is the open lock. An open lock was defined as a lock whose external effect is left undecided. Compatibility with other requests is decided individually each time a request is submitted. The decision may depend on the profile of the requesting process and the current state of the object, respectively. One possibility is that the owner of the lock himself makes the decision. A better solution would be to let the system automatically decide on the basis of predefined rules. This frees the owner from being (frequently) disturbed in his work by concurrent processes.

The extended lock concept can be used as a solid basis to implement higher-level concepts, like semantics-based concurrency control (cf. [ChRR91], [HeWe88], [SpSc84], [Weih88]).

5 Object-related Locks

Usually, locks are bound to transactions. In cooperative environments, however, it seems to be reasonable to think about an extension of this rule in a direction that locks can also be bound to objects. Similar to the life of a human being who is born single and who may, at some later time, acquire marriage status (which rules out a return to the status "unmarried"), objects may also go through several states in their lifetime. They may be 'born' without any restrictions on the way in which they can be treated. However, during their lifetime, some restrictions may come into force (see example 5.1).

Example: 5.1

Especially in design applications the concept of versioning is of a great relevance.

1. *Version graph*

   In the context of version graphs it is commonly required that a non-leaf node (inner node) cannot be modified to prevent the successors of that node from being invalidated (since the predecessor is no longer the version from which they were created). Here an object is born without any restrictions (leaf) and, later, changes to an object which can no longer be modified (non-leaf node).

2. *Time versions*

   Time versions only permit one version of an object to be valid at a given time. If the currently valid (latest) state of an object is to be modified, a new version is created. This results in a linear sequence of versions. In this case the object is born with the restriction to be not changeable. Later on, it will change to an object which can neither be modified nor used as a basis for the derivation of a new version (non-leaf node).

3. *Standard or library objects*

   Many application classes, especially design environments, put standard or library objects at the users' disposal. Such objects can only be read. They are born with the restriction: *modification prohibited*.

Our idea is to link locks permanently to objects. This kind of lock will be called **object-related lock** (**OR-lock** for short). An OR-lock, once imposed on an object, can neither be weakened nor released, it can only be upgraded. The lock remains valid as long as the object exists. OR-locks behave like conventional locks (locks linked to transactions, called **transaction-related locks** or **TR-locks** for short in the following, which only have an internal effect). If an OR-lock is granted, the community of all (potential) transactions can be seen as the owner of the lock. All rights of the internal effect can still be acquired while all other rights are no longer "grantable". Since an OR-lock is persistent we will distinguish it from a transaction lock by placing a P in front of the signature.

The following OR-locks can directly be adopted from the set of conventional locks:

PU-Lock: prohibits deletion of the object. All other operations are permitted.

PD-Lock: prohibits deletion and modification of the object. All other operations are permitted.

PS-Lock: prohibits deletion, modification, and derivation of a new version of the object. It only permits the read operation.

It is particularly worthwhile taking a closer look at the PD-lock. This lock permits the derivation of a new version of the object locked. However, it is not laid down whether only one derivation can be produced or more. But such a distinction is extremely useful, since it makes it possible to automatically control the observance of the rules of different version models (time versions and version graphs). Due to the great significance of version models, a distinction seems to be reasonable. Therefore, the PD-lock is split into two lock modes:

PSD-lock: only permits derivation of exactly one new version; i.e., after the first derivation of a new version the lock mode is converted to a PS-lock (forms a sequence of versions).

PMD-lock: permits derivation of any number of new versions (forms a version graph).

Finally, a PX-lock is introduced as the lock with which every object is 'born':

PX-lock: is a pseudo lock which does not impose any restrictions on the object (needs not to be considered by the concurrency control component).

The introduction of a PB-lock does not make any sense, since it has the same effect as the PS-lock.

Table 5.1 describes the compatibility between an OR-lock and (the internal effect of) a conventional lock:

Similar to the external effect of conventional locks, OR-locks increase from PX to PS since each step on this way concedes fewer rights on the object.

As already mentioned, with the help of OR-locks concurrency control overhead can be reduced. For instance, with a PS-lock the object has no longer to be considered by the concurrency control component, since the only applicable operation is the read operation. A PS-lock is especially favorable if standard objects need to be handled.

Let us consider figure 5.1, in which the standard object O2 is a shared subcomponent of several complex objects (CO1, CO2, CO3). If, for example, some application T locks CO1 in an exclusive mode (U/B-lock), this lock prevents each competitor C from locking any of the other complex objects which also include O2 (here CO2 and CO3). However, if a PS-lock is imposed on O2, the lock manager no longer has to consider O2. Therefore, the concurrent access can be granted. Situations like this are rather frequent, especially in design environments.

object lock \ transaction lock	B	S	D	U	X
PX	+	+	+	+	+
PU	+	+	+	+	–
PMD	+	+	+	–	–
PSD	+	+	o	–	–
PS	+	+	–	–	–

+ = *permitted*
– = *prohibited*
o = *single derivation only*

Table 5.1: Compatibility matrix object/transaction-related locks

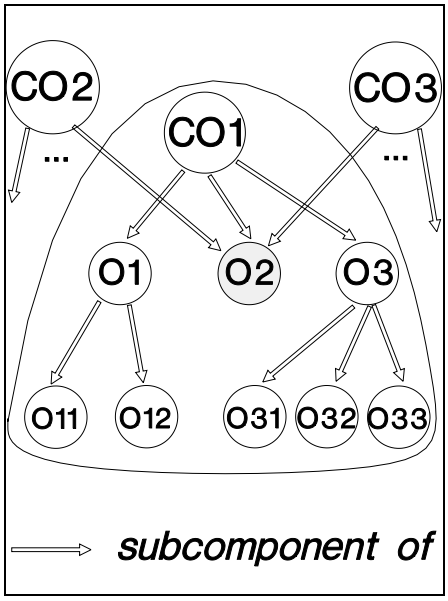

Figure 5.1: Standard object O2 as *subcomponent* of *several* complex objects

6 Subject-Related Locks

Non-standard applications are usually extremely complex in comparison with traditional applications. One frequently finds complex tasks which need to be split into several units of work, each of which, nevertheless, may still be complex and of long duration. Since these units of work represent *one* complex task, they often have a complex control flow; i.e., some units can be executed concurrently while others need to be executed in succession. Each unit of work accesses a (large) number of objects. Some of them are only used within *one* given unit. However, the more central objects with respect to the task are often used in (nearly) all subtasks. Consequently, we must be able to safely transfer objects from one unit of work to the next. If units of work correspond to (sub)transactions, we need a mechanism which allows an object to be safely transferred from one transaction to the next (see also [WäRe92]). Consider, for example, an administration department. If someone makes an application for something (for example, a business trip), he has to fill in the application form. Then the application form usually has to pass through several stages; it has to be countersigned by a manager, registered and checked for correctness, some computations have to be performed, etc. Often each subtask is performed by a different person/department. This means that we need to control the correct flow of the application form; i.e., a safe transfer from one subtask (transaction) to the next has to be ensured. For this reason we provide subject-related locks. A **subject-related lock** (**SR-lock** for short) again is defined by an internal/external effect pair. It is bound to a subject for some time. During this period, the subject is the owner of the lock and can decide autonomously how to use the locked object (for example, in which application). A **subject** can be anything that can be identified by the concurrency control component as such, for example, an application, a user (group), an agent, a named sequence of actions (transactions) etc.

A subject-related lock corresponds to a transaction-related lock (TR-lock) in that it is defined by an internal/external effect pair. It is bound *temporarily*, however, to a *subject* and not to a transaction. Since an SR-lock functions as a place holder, it reserves an object O for (later) use by its owner in one or more (consecutive) (trans)actions. Every subject can ask for an SR-lock at any time. The SR-lock can be granted if no TR-, OR-, or SR-lock is (currently) assigned to some competitor in an incompatible mode. Here, a competitor is either a transaction/application (in case of a TR-lock) or a subject (in case of a concurrent SR-lock) or an

object (in case of an OR-lock). The owner of an SR-lock can hold the lock as long as he wants; i.e., an SR-lock is released by an explicit command of its owner. This can be done at any time.

An SR-lock imposes restrictions on the way in which competitors are allowed to work on the locked object. Consequently, the external effect of an SR-lock corresponds to the external effect of a TR-lock. The internal effect determines which TR-lock on the object can be granted to a transaction at most. This means, that an SR-lock does not assign any directly usable rights to its owner. Instead, rights can only become valid if they are supplemented by a TR-lock within a concrete transaction. However, since the SR-lock is a place holder it is guaranteed that the corresponding TR-lock can immediately be granted, provided that the requested TR-lock is not stronger than the SR-lock. If it is stronger, of course, the TR-lock has to compete with all already granted locks on that object. Each internal effect/external effect-pair which is a valid TR-lock represents also a valid SR-lock.

The additional acquisition of the TR-lock is necessary to ensure that the locked object O can only be manipulated within a transaction. Moreover, it guarantees that the owner of the SR-lock cannot improperly exploit his rights, for example, by using O in an incompatible way within different transactions. The TR-lock ensures that O can only be handled in a correct way. Only after the TR-lock is released, O is once again put at the disposal of the owner of the SR-lock and can, therefore, be employed in another transaction. Of course, since O can only be used if a proper TR-lock is granted, an SR-lock can be released at any time. Either O is currently not used by some transaction or it is still protected by the additional TR-lock.

Example 6.1:

Let us assume that a user P has acquired the SR-lock *U/S* on O (allows P to modify the object, concurrent transactions may still read it). If P wants to modify O in transaction T he can directly supplement the SR-lock by the corresponding TR-lock. This prevents P from afterwards acquiring an incompatible lock mode on O in a concurrent transaction.

7 Conclusion

In this paper we have shown how the conventional locking approach can step by step be adapted to a semantics-based methodology of concurrency control. The first step was to enhance the coarse and small set of conventional lock modes (shared and exclusive lock) by other useful lock modes. However, this was only a small step in the direction of being able to support more application-specific semantics. The next step was to consider the semantics of locks and to differentiate between an internal and an external effect of a lock. This permits a much more precise adaptation of a lock to the requirements of an application. It especially permits the creation of shielded cooperative environments within nested transactions. Finally, our initial goal - a comprehensive support of synergistic cooperative work by the exploitation of application-specific semantics - was attained by allowing locks to be extended by rules.

In addition to the 'conventional' binding of locks to transactions we consider the binding of locks to objects (*object related*) and subjects (*subject related locks*). Object related locks can define persistent and adaptable access restrictions on objects. This permits, among others, the modeling of different types of version models (time versions, version graphs) as well as library (standard) objects. Moreover, these locks can be exploited by the concurrency control component to increase efficiency. Subject related locks are bound to subjects (user, application, etc.) and can be used among others to supervise or direct the transfer of objects between transactions.

Literature

[BaRa90] Badrinath, B. R.; Ramamrithan, K.: *Performance Evaluation of Semantics-Based Multilevel Concurrency Control Protocols*; Proc. ACM-SIGMOD Int. Conf. on Management of Data; Atlantic City, NJ; May 1990

[BeSW88] Beeri, C.; Schek, H.-J.; Weikum, G.: *Multi-Level Transaction Management: Theoretical Art or Practical Need*; Proc. Intl. Conference Extending Data Base Technology (EDBT); Lecture Notes in Computer Science 303, J. W. Schmidt, S. Ceri, M. Missikoff (eds.); Springer Publishing Company; 1988

[BÖHG92] Buchmann, A.; Özsu, T.; Hornick, M.; Georgakopoulos, D.; Manola, F.: *A Transaction Model for Active Distributed Object Systems*; in [Elma92]

[ChRR91] Chrysanthis, P.; Raghuram, S.; Ramamrithan, K.: *Extracting Concurrency from Objects: A Methodology*; Proc. ACM-SIGMOD Int. Conf. on Management of Data; Denver, Colorado; May 1991

[DaBM88] Dayal, U.; Buchmann, A.; McCarthy, D.: *Rules are Objects too: A Knowledge Model for an Active, Object-Oriented Database Management System*; Proc. 2nd Int. Workshop on Object-Oriented Database Systems; Bad Münster, Germany; Sept. 1988

[Elma92] Elmargarmid, A. (ed.): Database *Transaction Models for Advanced Applications"*; Morgan Kaufmann Publishers; 1992

[ElGR90] Ellis, C.; Gibbs, S.; Rein, G.: *Design and Use of a Group Editor*; in: Engineering for Human-Computer Interaction, G. Cockton (ed.); North-Holland, Amsterdam; 1990

[ElGR91] Ellis, C.; Gibbs, S.; Rein, G.: *Groupware: Some Issues and Experiences*; Communications of the ACM; Vol. 34, No. 1; Jan. 1991

[Garc83] Garcia-Molina, H.: *Using Semantic Knowledge for Transaction Processing in a Distributed Database*; ACM Transactions on Database Systems (TODS); Vol. 8, No. 2; June 1983

[GLPT76] Gray, J.; Lorie, R.; Putzolu, F.; Traiger, I.: *Granularity of Locks and Degrees of Consistency in a Shared Data Base;* in: 'Modelling in Data Base Management Systems'; G. M. Nijssen (editor); North Holland Publishing Company; 1976

[Gray79] Gray, J.: *Notes on Data Base Operating Sytems;* in: Operating Systems - An Advanced Course; Bayer, R.; Graham, R. M.; Seegmüller, G. (editors); Lecture Notes in Compuetr Science 60; Springer Publishing Company; 1979

[GrSa88] Greif, I.; Sarin, S.: *Data Sharing in Group Work*; in: Greif, I. (ed.): *Computer-Supported Cooperative Work: A Book of Readings*; Morgan Kaufmann; San Mateo, CA; 1988

[HaLo81] Haskin, R. L.; Lorie, R. A.: "*On Extending the Functions of a Relational Database System*"; IBM Research Report RJ3182; 1981

[HaPS93] Haghjoo, M.; Papazoglou, M.; Schmidt, H.: *A Semantic-based Nested Transaction Model for Intelligent and Cooperative Information Systems;* Proc. Int. Conf. on Intelligent and Cooperative Information Systems; IEEE Computer Society Press; Rotterdam, The Netherlands; May 1993

[HäRe83] Härder, T.; Reuter, A.: *Principles of Transaction Oriented Database Recovery*; ACM Computing Surveys, Vol. 15, No. 2; June 1983

[HäRo87] Härder, Th.; Rothermel, K.: *Concurrency Control Issues in Nested Transactions*; IBM Almaden Research Report RJ5803, San Jose; Aug. 1987

[HeWe88] Herlihy, M.; Weihl, W.: *Hybrid Concurrency Control for Abstract Data Types*; Proc. ACM Symposium on Principles of Database Systems, 1988

[JaMR92] Jarke, M.; Maltzahn, C.; Rose, T.: *Sharing Processes: Team Coordination in Design Repositories*; Int. Journal of Intelligent and Cooperative Information Systems; Vol. 1, No. 1; March 1992

[KLMP84] Kim, W.; Lorie, R.; McNabb, D.; Plouffe, W.: *A Transaction Mechanism for Engineering Design Databases*; Proc. 9th Int. Conf. on Very Large Data Bases (VLDB); Singapore; Aug. 1984

[KLRW94] Kirsche, T.; Lenz, R.; Ruf, T.; Wedekind, H.: *Cooperative Problem Solving Using Database Conversations*; Proc. 10th Int. Conf. on Data Engineering; Houston, Texas; Feb. 1994

[KoKB85] Korth, H.F.; Kim, W.; Bancilhon, F.: *A Model of CAD Transactions*; Proc. 10th Int. Conf. on Very Large Data Bases (VLDB); Stockholm, Sweden; Aug. 1985

[KSUW85] Klahold, P.; Schlageter, G.; Unland, R.; Wilkes, W.: *A Transaction Model Supporting Complex Applications in Integrated Information Systems*; Proc. ACM-SIGMOD Int. Conf. on Management of Data; Austin, Texas; 1985

[KUSW92] Knolle, H.; Unland, R; Schlageter, G.; Welker, E.: *TOPAZ: A Tool Kit for the Construction of Application-Specific Transaction Managers;* in: 'Objektbanken für Experten'; R. Bayer, T. Härder, P. Lockemann (eds.); Springer Verlag; Informatik aktuell; 1992

[LoPl83] Lorie, R.; Plouffe, W.: *Complex Objects and Their Use in Design Transactions*; Proc. Databases for Engineering Applications; ACM-Database Week, San Jose, California; 1983

[Moss81] Moss, J.E.B.: *Nested Transactions: An Approach to Reliable Computing*; MIT Report MIT-LCS-TR-260, Massachusetts Institute of Technology, Laboratory of Computer Science; 1981 and *Nested Transactions: An Approach to Reliable Distributed Computing*; The MIT Press; Research Reports and Notes, Information Systems Series; M. Lesk (Ed.); 1985

[NoZd90] Nodine, M.; Zdonik, S.: *Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications*; Proc. 15th Int. Conf. on Very Large Data Bases (VLDB); Brisbane, Australia; Aug. 1990

[NoRZ92] Nodine, M.; Ramaswamy, S.; Zdonik, S.: *A Cooperative Transaction Model for Design Databases*; in [Elma92]

[Skar93] Skarra, A.: *Concurrency Control and Object-Oriented Databases*; Proc. 9th Int. Conf. on Data Engineering; Vienna, Austria; Apr. 1993

[SkZd89] Skarra, A.; Zdonik, S.: *Concurrency Control and Object-Oriented Databases*; in: 'Object-Oriented Concepts, Databases, and Applications'; Kim, W., Lochovsky, F. (Editors); Addison-Wesley Publishing Company; 1989

[SpSc84] Schwarz, P.; Spector, A: *Synchronizing Shared Abstract Types*; ACM Transactions on Computer Systems; Vol. 2, No. 3; August 1984

[Unla90] Unland, R.: *A Flexible and Adaptable Tool Kit Approach for Concurrency Control in Non Standard Database Systems*; Proc. 3rd Int. Conf. on Database Theory (ICDT); Paris, France; Dec. 1990

[Unla91] Unland, R.: *TOPAZ: A Tool Kit for the Construction of Application Specific Transaction Managers*; Research-Report MIP-9113; University of Passau; Department of Computer Science; Oct. 1991

[UnSc92] Unland, R., Schlageter, G.: *A Transaction Manager Development Facility for Non-Standard Database Systems*; in: [Elma92]

[WäRe92] Wächter, H.; Reuter, A.: *The ConTract Model*; in: [Elma92]

[Weih88] Weihl, W.: *Commutativity-Based Concurrency Control for Abstract Data Types*; Proc. IEEE 21th Annual Hawaii Int. Conf. on System Sciences (HICSS); Hawaii; Jan. 1988

Arbeitsberichte des Instituts für Wirtschaftsinformatik

Nr. 1 Bolte, Ch., Kurbel, K., Moazzami, M., Pietsch, W.: Erfahrungen bei der Entwicklung eines Informationssystems auf RDBMS- und 4GL-Basis; Februar 1991.

Nr. 2 Kurbel, K.: Das technologische Umfeld der Informationsverarbeitung - Ein subjektiver 'State of the Art'-Report über Hardware, Software und Paradigmen; März 1991.

Nr. 3 Kurbel, K.: CA-Techniken und CIM; Mai 1991.

Nr. 4 Nietsch, M., Nietsch, T., Rautenstrauch, C., Rinschede, M., Siedentopf, J.: Anforderungen mittelständischer Industriebetriebe an einen elektronischen Leitstand - Ergebnisse einer Untersuchung bei zwölf Unternehmen; Juli 1991.

Nr. 5 Becker, J., Prischmann, M.: Konnektionistische Modelle - Grundlagen und Konzepte; September 1991.

Nr. 6 Grob, H.L.: Ein produktivitätsorientierter Ansatz zur Evaluierung von Beratungserfolgen; September 1991.

Nr. 7 Becker, J.: CIM und Logistik; Oktober 1991.

Nr. 8 Burgholz, M., Kurbel, K., Nietsch, Th., Rautenstrauch, C.: Erfahrungen bei der Entwicklung und Portierung eines elektronischen Leitstands; Januar 1992.

Nr. 9 Becker, J., Prischmann, M.: Anwendung konnektionistischer Systeme; Februar 1992.

Nr. 10 Becker, J.: Computer Integrated Manufacturing aus Sicht der Betriebswirtschaftslehre und der Wirtschaftsinformatik; April 1992.

Nr. 11 Kurbel, K., Dornhoff, P.: A System for Case-Based Effort Estimation for Software-Development Projects; Juli 1992.

Nr. 12 Dornhoff, P.: Aufwandsplanung zur Unterstützung des Managements von Softwareentwicklungsprojekten; August 1992.

Nr. 13 Eicker, S., Schnieder, T.: Reengineering; August 1992.

Nr. 14 Erkelenz, F.: KVD2 - Ein integriertes wissensbasiertes Modul zur Bemessung von Krankenhausverweildauern - Problemstellung, Konzeption und Realisierung; Dezember 1992.

Nr. 15 Horster, B., Schneider, B., Siedentopf, J.: Kriterien zur Auswahl konnektionistischer Verfahren für betriebliche Probleme; März 1993.

Nr. 16 Jung, R.: Wirtschaftlichkeitsfaktoren beim integrationsorientierten Reengineering: Verteilungsarchitektur und Integrationsschritte aus ökonomischer Sicht; Juli 1993.

Nr. 17 Miller, C., Weiland, R.: Der Übergang von proprietären zu offenen Systemen aus Sicht der Transaktionskostentheorie; Juli 1993.

Nr. 18 Becker, J., Rosemann, M.: Design for Logistics - Ein Beispiel für die logistikgerechte Gestaltung des Computer Integrated Manufacturing; Juli 1993.

Nr. 19 Becker, J., Rosemann, M.: Informationswirtschaftliche Integrationsschwerpunkte innerhalb der logistischen Subsysteme - Ein Beitrag zu einem produktionsübergreifenden Verständnis von CIM; Juli 1993.

Nr. 20 Becker, J.: Neue Verfahren der entwurfs- und konstruktionsbegleitenden Kalkulation und ihre Grenzen in der praktischen Anwendung; Juli 1993.

Nr. 21 Becker, K., Prischmann, M.: VESKONN - Prototypische Umsetzung eines modularen Konzepts zur Konstruktionsunterstützung mit konnektionistischen Methoden; November 1993

Nr. 22 Schneider, B.: Neuronale Netze für betriebliche Anwendungen: Anwendungspotentiale und existierende Systeme; November 1993.

Nr. 23 Nietsch, T., Rautenstrauch, C., Rehfeldt, M., Rosemann, M., Turowski, K.: Ansätze für die Verbesserung von PPS-Systemen durch Fuzzy-Logik; Dezember 1993.

Nr. 24 Nietsch, M., Rinschede, M., Rautenstrauch, C.: Werkzeuggestützte Individualisierung des objektorientierten Leitstands ooL, Dezember 1993.

Nr. 25 Meckenstock, A., Unland, R., Zimmer, D.: Flexible Unterstützung kooperativer Entwurfsumgebungen durch einen Transaktions-Baukasten, Dezember 1993.

Nr. 26 Grob, H. L.: Computer Assisted Learning (CAL) durch Berechnungsexperimente, Januar 1994.

Nr. 27 Kirn, St., Unland, R. (Hrsg.): Tagungsband zum Workshop "Unterstützung Organisatorischer Prozesse durch CSCW". In Kooperation mit GI-Fachausschuß 5.5 "Betriebliche Kommunikations- und Informationssysteme" und Arbeitskreis 5.5.1 "Computer Supported Cooperative Work", Westfälische Wilhelms-Universität Münster, 4.-5. November 1993

Nr. 28 Kirn, St., Unland, R.: Zur Verbundintelligenz integrierter Mensch-Computer-Teams: Ein organisationstheoretischer Ansatz, März 1994.

Nr. 29 Kirn, St., Unland, R.: Workflow Management mit kooperativen Softwaresystemen: State of the Art und Problemabriß, März 1994.

Nr. 30 Unland, R.: Optimistic Concurrency Control Revisited, März 1994.

Nr. 31 Unland, R.: Semantics-Based Locking: From Isolation to Cooperation, März 1994.