Unland, Rainer

Working Paper

TOPAZ: A tool kit for the assembly of transaction managers for non-standard applications

Arbeitsberichte des Instituts für Wirtschaftsinformatik, No. 34

Provided in Cooperation with:
University of Münster, Department of Information Systems

Suggested Citation: Unland, Rainer (1994) : TOPAZ: A tool kit for the assembly of transaction managers for non-standard applications, Arbeitsberichte des Instituts für Wirtschaftsinformatik, No. 34, Westfälische Wilhelms-Universität Münster, Institut für Wirtschaftsinformatik, Münster

This Version is available at:
https://hdl.handle.net/10419/59343

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. 34

TOPAZ: A Tool Kit for the Assembly of Transaction Managers for Non-Standard Applications

Rainer Unland

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

November 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 in 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*. Unfortunately, the demands are not only sophisticated but also diversified, which means that different application areas might even place contradictory demands on transaction management. This paper deals with these problems and offers a solution by introducing a flexible and adaptable *tool kit approach for transaction management*. This tool kit enables a database implementor or applications designer to assemble application-specific transaction managers. Each such transaction manager is meant to provide a number of individualized, application-specific transaction types. Such transaction types can be constructed by selecting a meaningful subset from a "starter set" of basic constituents. Among the basic components provided by the starter set are those for concurrency control, recovery, and transaction processing control. In a first step these basic components are assembled and adapted to each other to form a kind of (non-executable) "skeleton" transaction. Skeleton transactions can be customized to make them a more meaningful basis for the construction of executable transaction type. Finally, executable transaction types can be constructed by equipping appropriate skeleton transaction with the specific semantics of the transaction model of choice. To be able to emulate each kind of application environment the different transaction types must be executable in any order within a nested transaction hierarchy. For this reason we propose a kind of "meta" (transaction) model. It specifies the constraints and rules which need to be obeyed by each transaction type. Particular emphasis is placed on the integration of flexible and powerful concepts for a comprehensive support of cooperative work.

1 Introduction

Conventional database systems have satisfied the requirements of applications for which they were designed, namely, business data processing applications such as inventory control, payroll, accounts, and so on. However, with the success of database systems in these areas new classes of applications have been identified which are also expected to benefit substantially from adequate database support. These application classes include: computer-aided design (CAD); engineering (CAE); software engineering (CASE), and manufacturing (CAM) systems; knowledge-based systems (expert systems and expert system shells); multimedia systems that manage images, graphics, voice, and textual documents; statistical and scientific applications, analysis programs, and so on (c. f., [Kim91]). However, these applications place demands on database systems that far exceed the capabilities of conventional (relational) systems. Such '*advanced database applications*' differ from traditional database applications in a variety of ways. Above all they require more powerful and flexible concepts for data modeling; for example, facilities are needed to model and manage complex nested entities, such as design and engineering objects or compound documents. Among others, such an advanced data model should provide:

- a rich set of data types which, moreover, should be extensible as with user-defined data types, and which should allow the application to model and manage specific 'shadings' of data types such as library or standard objects (objects which cannot be modified);
- facilities to define type specific operations and to integrate them into the database;
- powerful semantic concepts, such as generalization, association, or aggregation relationships;
- concepts for adequate support oftemporal evolution of data; for instance, concepts which permit the modeling of the temporal dimension of data and versioning of data.

However, the demands of non-standard applications on data management are not restricted to more flexible and powerful data modeling concepts. There is also a need for powerful and flexible storage structures and access methods in order to ensure the efficient execution of database operations. Moreover, non-standard applications often need to interact in a much more

sophisticated way with the database system. This results in considerably extended demands on transaction management.

With respect to the architecture of future database systems two major trends can be identified (see also [NeSt89]). Some groups favor the development of a *complete* DBMS comprising a parser, query optimizer, storage structures, access methods, transaction management, and so on. On the basis of this approach user extensions are satisfied within the context of a **full-function DBMS** (see Figure 1.1). Examples of this approach are AIM-P ([DKAB86]), ConceptBase ([JaJR88]), GemStone ([MSOP86]), OOPS ([UnSc89-2]), ORION ([WoKi87]), POSTGRES ([StRo87]), PROBE ([DaSm86]), and STARBURST ([SCFL86]).

Other groups favor a **database kernel system**. The underlying assumption of this approach is that a *general* storage system can serve as the *universal basis* for the realization of all 'flavors' of application-specific database systems; for instance, there exists an application-independent

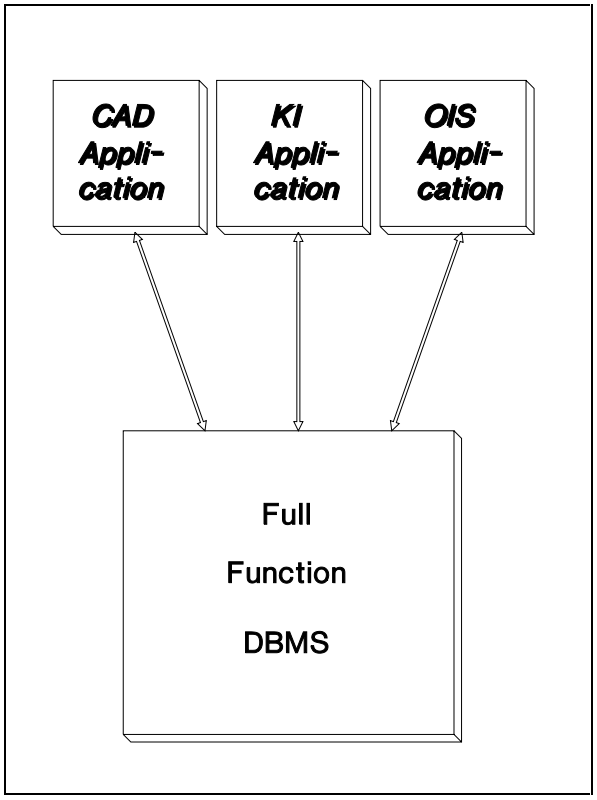


Fig. 1.1: Full function database management system

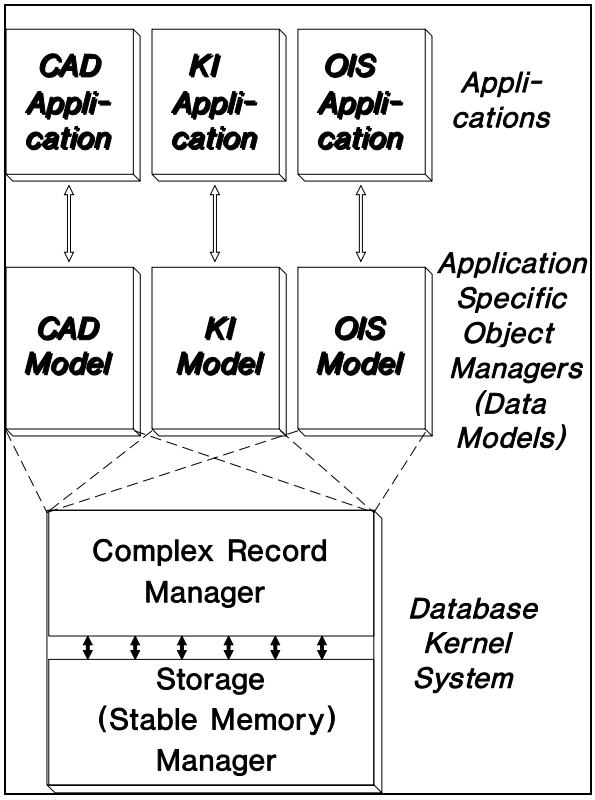


Fig. 1.2: Database kernel system and tool kit approach

kernel system on the basis of which the various application-specific data models have to be implemented (see Figure 1.2). This approach is investigated, for example, in DASDBS ([PSSW87]), OMS ([FrBo89]), and PRIMA ([HMMS87]).

A very similar approach is the **tool kit approach** where, in addition to a kernel, an **"erector set" of modules** is provided to allow a sophisticated applications designer or database implementor (DBI) to construct a customized system. This approach is being taken, for example, in EXODUS ([CaDe87]), and GENESIS ([BBGS88]). Advocates of the tool kit approach argue that requirements from the various application areas differ so much that a *single* interface is not appropriate for all of them. The consequence of this understanding is to provide a common storage system (kernel) and, in addition, to offer a tool kit as the basis for the development of application-specific front ends. The assumption is that this architecture not only supports applications in a more appropriate and natural way, but also makes the specifically tailored database systems more efficient.

However, it is not sufficient to provide rich and flexible data modeling facilities together with efficient and adaptable storage structures and access methods. New application areas also make sophisticated demands on transaction management (c. f., [ELLR90], [Katz84], [KaWe83], [Kelt88], [KoKB88], [KSUW85], [LoPl83], [NoZd90], [UnSc89-1], [UnSc91/2]). Without question, the transaction model realized in traditional database systems is powerful when applied to conventional applications. However, it is found *lacking* in *efficiency* and *functionality* when used for new applications. Efficiency is of particular importance in view of the throughput demands placed on the new generation of database systems. In terms of functionality, traditional transactions were assumed to be of simple structure and short-lived. Furthermore, they were targeted for competitive environments. Activities in non-standard application areas tend to access many objects, involve lengthy computations, and are interactive, that is, they pause for input from the user or the application. Although long-duration and collaborating activities are often found in these environments they cannot be represented by traditional transactions owing to their inability to meet the correctness requirement of serializability. Even in those cases where only some of the above characteristics are required and the corresponding activities can be modelled as traditional transactions, they degrade system performance due to increased data contention, thus failing to meet high throughput demands ([ChRa90]). The need for the next generation of database systems to capture the reactive, long-lived, and interactive

activities found in most new application areas demands the development of more suitable transaction models. These transaction models have to support *long-duration, interactive activities, application-* or *user-supported consistency*, and *synergistic cooperative work*. Whether a system is characterized as competitive or cooperative depends on how interactions among activities in the system are viewed: in *competitive environments*, interactions are more or less hindered whereas they are promoted in *cooperative environments*.

In order to fulfill the needs for more flexible transaction models, various extensions to the traditional model have been proposed. A first and rather promising approach was an extension of the traditional concept of transactions so that the otherwise flat transaction model permitted transactions to be executed within other transactions. This is called a **nested transaction**. In the commonly known (but not original) approach of Moss ([Moss81]) a nested transaction (recursively) consists of a set of child transactions that execute atomically with respect to their parent transactions and their siblings. A long-duration transaction may exhibit a rich and complex internal structure which can be exploited to distribute the work within the transaction, to execute it in parallel, and to roll back unsuccessful parts without affecting others.

Brief overview of the approach and points of novelty

To the best of our knowledge the nested transaction model is the fundamental basis of all advanced transaction models which have been proposed in literature. The differences between the various models consist in the number and meaning of the constraints and the rules which they place on the way (nested) transactions have to be formed and how they have to interact with each other.

Prominent examples of these constraints and rules are the following:

- Most approaches provide different transaction types. However, these types can only be *nested* in a *special, predefined order*. For example,
  - [KoKB85] present a model in which a design transaction consists of a number of project transactions each of which consists of a set of cooperating transactions. Each cooperating transaction, in turn, is a hierarchy of client-subcontractor transactions, each of which is a set of designer's transactions.

- ↳ [KSUW85] define a database transaction to be the basis for a set of user transactions. [UnSc89-1] add to this model group transactions.

- ☞ Some proposals require *all* transactions to run a *strict* two-phase lock protocol (locks cannot be released before end-of-transaction (EOT)), for example, Moss ([Moss81]) or Katz ([Katz84]).

- ☞ Almost all proposals restrict child transactions to *commit objects* (or locks) to their *parent transactions* only, for example, [HäRo87-1+2], [KLMP84], [KoKB85], [KSUW85], [Moss81], [UnSc89-1].

- ☞ Some approaches, for example, [KLMP84], [KoKB85] or [UnSc89-1], only allow *leaf transactions* to *perform operations* on data. *Non-leaf transactions* serve only as a kind of *database* for their child transactions.

Another observation is that a large number of approaches to advanced transaction modeling concentrate first of all on modularity, failure handling, and concurrent execution of subtasks while support of cooperative work is subordinated to serializability. Mostly, the assumption is still made that (sub)transactions are competitors rather than partners. Appropriate support of cooperative work, however, can only be achieved if the still predominant rigid concurrency control measures (strict isolation) are weakened, for instance, by moving some responsibility for the integrity of the data from the database management system to the application. Of course, this has to be done in a controlled way and the database management system has to offer as much help as possible.

Furthermore, advanced application areas are highly diverse. Applications may differ substantially in their demands and even in their understanding of consistent operations on data. This has led to a number of different, sometimes even contradictory demands on transaction management. Therefore, it is not surprising that most of the above rules and constraints reflect the individual views of the authors concerning the requirements and conditions of the application area which they claim as the target class for their transaction model. However, irrespective of how successful these extended transaction models are in supporting the systems that they were intended for, they merely represent points within the range of interactions required within the

spectrum from competitive to cooperative environments. Based on this observation it may look rather promising to choose a more general transaction model and try to extend it in a way that it covers (most of) the remaining points in the spectrum. While it is tempting to follow this approach any such work will by necessity be ad hoc and not general. It is our strong opinion that each 'hard-wired' transaction model will only capture a subset of the broadly diversified spectrum of demands to be found in complex information systems. This already follows from the fact that competitive environments usually place contrary demands on concurrency control and transaction management than cooperative environments. Consequently, a hard-wired transaction model may be suitable for a number of application areas, whereas it will be inappropriate for others (see also [ChRa90], page 1, or [NeSt89], page 5).

At the time when we started our project we had several basic requirements in mind which we wanted to be fulfilled by our approach. The most important are the following:

- ***Flexibility***
  Our approach should not be suited to a special application area but should be application independent.

- ***Cooperative work***
  Our approach should be capable of intensively supporting synergistic cooperative work.

- ***Adaptability***
  Our approach should permit an easy adaptation of the transaction manager to changing requirements of application areas.

- ***Extensibility***
  Our approach should be extensible in a way that special or new demands which cannot be meet yet can be satisfied after the integration/realization of new components, concepts, or features.

However, as the discussion at the beginning of this section clearly indicates, there are serious arguments which strongly indicate that *one* given transaction manager can only be a more or less satisfactory compromise. From our point of view it looks much more promising to follow an approach similar to the tool kit approach of database systems: a *tool kit for transaction*

management. We want this tool kit to be part of the general tool kit (or erector set of modules) of the database (kernel) system; for instance, it is meant to serve a sophisticated applications designer or database implementor (DBI) to model *his* application-specific transaction manager in an appropriate and natural way. Therefore, the tool kit is not yet another transaction model but is intended to provide a general framework for the realization of application-specific transaction managers.

For a first discussion the tool kit can be seen as to provide a number of fundamental transaction types at its interface. In fact, as will be shown later, the tool kit provides at its basis a set of basic components. These basic components can be combined and suited to each other to form a kind of "skeleton" transaction. Skeleton transactions, in turn, serve as a basis for the construction of the (core) transaction types which are provided at the interface of the tool kit.

An *application-specific transaction manager* first of all has to supply the application with a comprehensive set of application-specific transaction types; for instance, transaction types which are especially tailored to the individual requirements of the application. This may require an adaptation of some of the general transaction types of the tool kit to the individual semantics of an application, for example, by adapting concurrency control or recovery measures. This corresponds to a **local** or **intra-transaction adaptation**.

In our model, transaction types may differ in their *structure* as well as in their *behavior*. Different transaction types may use different compatibility matrices, may install different concurrency control measures, may rely on different recovery techniques, or may or may not support special kinds of operations, like checkout/checkin, suspend-transaction/resume-transaction, or split-transaction/joint-transaction ([PuKH88]), to mention some of the characteristics which make up different transaction types. Note, that the same operation (= same operation name) may be equipped with different semantics or may rely on a different implementation if it belongs to a different transaction type or if it is defined in a different transaction manager. For example, the checkin operation of a given transaction type may be allowed to make use of an extended set of lock modes while the checkin operation of another transaction type is not allowed to do so (for a discussion of possible lock modes see later).

Since the tool kit does not predetermine an order in which transaction types have to be executed within nested transactions (like design transaction consists of a number of project transactions each of which consists of ...) each transaction type can be executed as subtransaction of any other transaction type. By this all shades of heterogeneously structured transaction trees can be constructed. This makes it possible to define different types of working environments within the same nested transaction hierarchy, for example, competitive environments which require strict isolation of (sub)transactions (in the sense of serializability) as well as different shades of cooperative environments (in which a non-serializable kind of work is supported; see Figure 1.3). Of course, most other transaction model do also support this feature. However, they exactly dictate the type of working environment (transaction type) which has to be installed at a given level of the nested transaction hierarchy. Of course, to make sure that an unrestricted nesting of transaction types does not cause problems each transaction type must observe a small set of fundamental rules which can be regarded as a kind of "meta" transaction model. Again, the generality of the tool kit may require the determination of some restrictions on the order in which transaction are allowed to run with respect to a specific application environment, for example, by establishing some constraints on the type of child transactions a parent transaction is allowed to run. This corresponds to a **global** or **inter-transaction adaptation**.

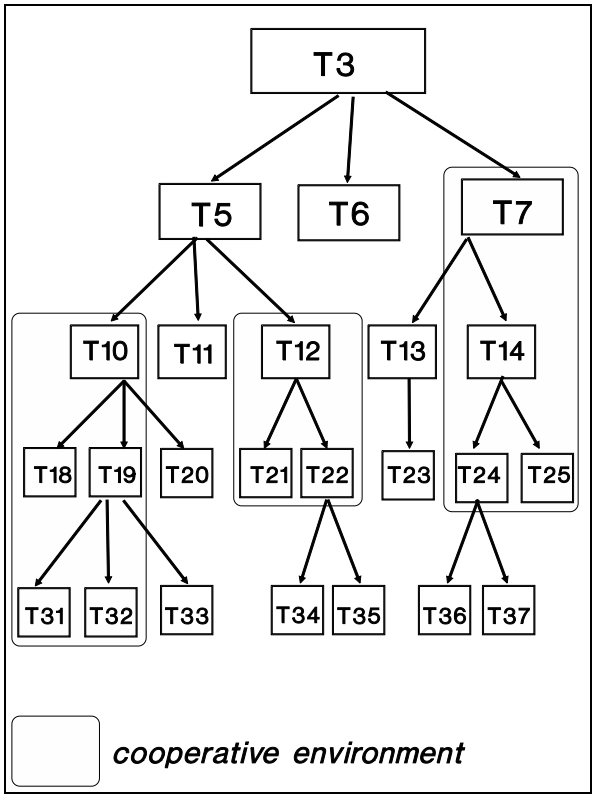


Fig. 1.3: *Competitive* and *cooperative* environments within a nested transaction

Structure of the paper

The remainder of this paper is organized as follows. Section 2 summarizes the basic terminology used in this paper. The fundamental set of rules of our approach is presented in section 3. The characteristics which make up different transaction types are discussed in section 4.

Section 5 concentrates on lock modes. First, a basic set of lock modes will be introduced. Then it will be shown how a subdivision of a lock into a part which describes the effect for the owner of the lock and another part which describes the effect for competitors can offer a sufficient fine granule for the definition of application-specific locks. This subdivision of a lock can especially be exploited within nested transactions to intensively support cooperative work. Furthermore, we will motivate and introduce lock modes which can be linked permanently to objects or temporarily to subjects (instead of temporarily to transactions). By this different version models as well as different object types (for example standard or library objects) can be supported. Section 6 summarizes the general rules of our transaction model. Section 7 briefly discusses first thoughts on a constraints, rules, and triggers mechanism which we want to integrate into the tool kit. Section 8 gives a brief overview of the structure of the tool kit while section 9 makes a few remarks on implementation issues. Section 10 surveys related work and compares it with the tool kit approach. Finally, section 11 concludes the paper and gives hints for further investigations.

2 Comparison of conventional and nested transactions

2.1 Conventional transaction management

A **transaction** is a sequence of reads and writes against a common and shared set of data - the database. Transaction management is concerned with supervising the way in which application programs and/or users share the database. A transaction is usually viewed as both the unit of concurrency and the unit of recovery. More specifically, the notion of transaction is commonly associated with four properties ([Gray79], [HäRe83]), namely atomicity, consistency, isolated execution, and durability (persistence), often referred to as the **ACID principle** (or **ACIDity properties** ([Leu91])).

☞ *Atomicity*

The **atomicity** property means that the sequence of reads and writes in a transaction is regarded as a single atomic action against the database. Consequently, atomicity defines a transaction as the unit of recovery; a transaction either completes successfully or it has no

effect on the database at all. Furthermore, the system can always recover to the boundaries of a transaction, such that results of partially completed transactions will not be visible.

☞ *Consistency*

Consistency assures that a successfully committed transaction always preserves the consistency of the database.

☞ *Isolation*

Isolated execution guarantees that the steps of several transactions can be interleaved so that they will not interfere with each other; for instance, it ensures that a transaction is completely shielded from the effects of any other concurrently executing transactions. The commonly agreed criterion for correct isolation is the **serializability property**. It requires that the effect of concurrent execution of more than one transaction is the same as that of executing the same set of transactions in a serial order (see Figure 2.1). To guarantee serializability a transaction manager employs a concurrency control scheme. The most common scheme is **two-phase locking**. It requires a transaction to be well-formed and two-phase. A transaction is **well-formed** if it always locks an entity in an appropriate lock mode before it works with the entity. It is **(simple) two-phase** if it consists of a **growing-phase** in which objects can only be acquired and a following **shrinking phase** in which objects can only be released ([Gray79]). If all objects are acquired at once at begin-of-transaction (BOT) the lock protocol is called **two-phase with predeclaring**.

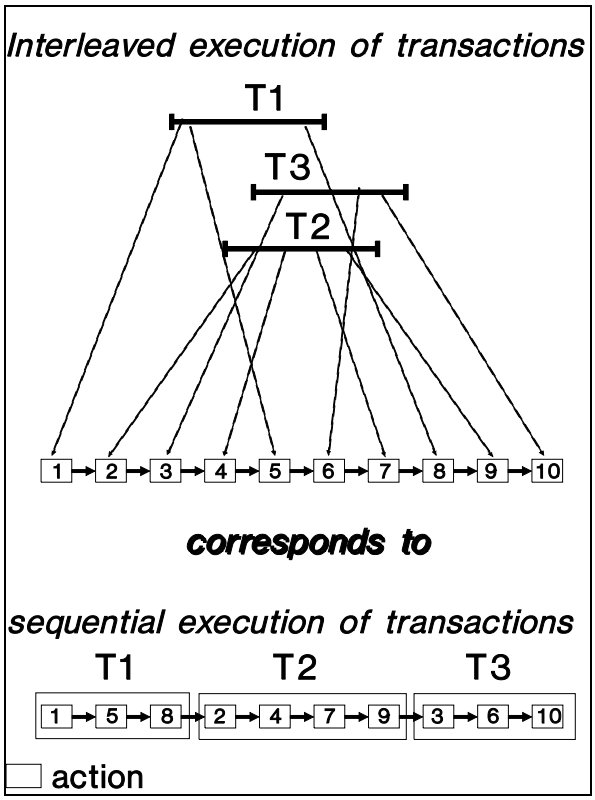


Fig. 2.1: Serializability property

Similarly, if all objects are released at once at end-of-transaction (EOT) the lock protocol is called **strict two-phase**. Finally, the lock protocol is called **non two-phase** or **conversational** if it is well-formed but not two-phase; for instance, objects can be acquired and released in arbitrary order. This protocol is especially important if cooperative work is to be supported**.**

☞ *Durability*

Finally, **durability** guarantees that once a transaction was committed successfully its results will survive any subsequent malfunction.

2.2 The concept of nested transactions

Traditional transaction management was mainly developed for business data processing/administrative applications. In such environments transactions are supposed to be rather short (typically at most a few seconds) and simple. Due to the simplicity of the application areas it is sufficient for efficient transaction management to provide exactly one type of transaction which is of flat structure and works directly on the data of the database. Moreover, due to the short run-time of transactions strict isolation is an acceptable feature since it usually does not slow down the execution of concurrent transactions significantly. However, this relatively simple kind of transaction management has some rather undesirable consequences for long-duration transactions. **Long-duration transactions** are transactions which usually cover a complex task. They last for much longer periods of time than conventional transactions (hours to days or even to weeks). To be efficient, a large number of personnel usually have to participate in the process of solving the problem who, of course, must share or exchange information. The conventional understanding of transactions as failure recovery and serialization units seems to be unacceptable in such situations. In particular, recovering from failures by rolling back to the beginning of a long-duration transaction and starting again is rarely appropriate for users who would then have to redo all of their work. The serializability property would result in the fact that if a long-duration transaction holds a lock on an object, any other long-duration transaction that must access the same object in a conflicting mode is blocked until the first long-duration transaction completes; for instance, other transactions are kept from executing their task for a long period of time. Furthermore, since the restrictive conditions of the seri-

alizability criterion prevent the exchange of data between transactions any kind of cooperative work is not within the scope of conventional transaction management.

The inability of traditional transaction management to support non-conventional application areas in an appropriate way led to an extension of the traditional concept of transactions in that the originally flat transaction model is allowed to include transactions within transactions. This kind of transaction is called **nested transaction**. The idea of nested transactions seems to have originated with Davies ([Davi73]) some time ago. He describes a nested transaction hierarchy as a *collection of nested spheres of control* where the outermost sphere is formed by the top-level transaction which incorporates the interface to the outside world (Figure 2.3 summarizes the 'tree terminology' that will be used in this paper). Reed ([Reed78]) presented the first comprehensive design of a nested transaction system. This design uses timestamps for synchronization. It was not before 1981 and the work of Moss ([Moss81]) that nested transaction attracted greater attention in the database community. Moss defines a **nested transaction** structure to recursively consist of a set of child transactions that execute atomically with respect to their parent transaction and their siblings. The nesting of transactions results either in hierarchical but *serial computations* similar to the ones that are achieved by procedure calls or in possibly *concurrent executions* if the invocation of a child transaction does not block the execution of its parent transaction. The first type of processing model is sometimes referred to as **synchronous invocation mechanism** while the second corresponds to an **asynchronous invocation mechanism** (c. f., [HäRo87-2]).

Moss' commonly known approach to nested transactions ([Moss81]) requires the results a parent transaction sees from its child transactions to be as though the child transactions execute serially. Main achievements of this basic approach are:

- ☞ *Modularity and failure handling*
  The nesting allows the user to structure and delegate his work and to define more graceful units of recovery (namely subtransactions).

- ☞ *Concurrent execution of work (subtasks)*
  Of course, the longer a transaction is, the greater the chance that parts of the transaction can be executed concurrently. However, to take advantage of this potential concurrency

in the application, suitable lock modes and granules of concurrency control have to be offered. The nested transaction concept embodies an appropriate control structure to support supervised and, therefore, safe intra-transaction concurrency. Of course, intra-transaction concurrency results in increased efficiency and decreased response time.

As long as one concentrates on these features the **operational integrity** (prevention of integrity violation which may result from the concurrent access of users to objects) can still be ensured entirely by the database system since modularity and failure handling as well as a concurrent execution of subtransactions still permit transactions to be strictly isolated from each other (see Figure 2.2). However, especially in design environments this type of nesting is still too restrictive. In such application areas the support of cooperative work within groups of users is mandatory. Contemporary concepts for transaction management assume that transactions are competitors rather than partners. Therefore, they place constraints on the manner in which transactions are allowed to work on the data that prevent any kind of cooperative work. In cooperative environments, transactions cooperate by, for example, accessing (reading) objects which are being modified by another transaction, by releasing objects before end-of-transactions, by exchanging or delegating objects, or by notifying each other of their behavior or state. This listing strongly indicates that a support of cooperative work can only be achieved if the rigid measures of conventional transaction management can be weakened; for instance, if some responsibility for the integrity of the data can be moved from the database system to the application environment (see Figure 2.2). Of course, this has to be done in a controlled way and the database system has to offer as much help as possible.

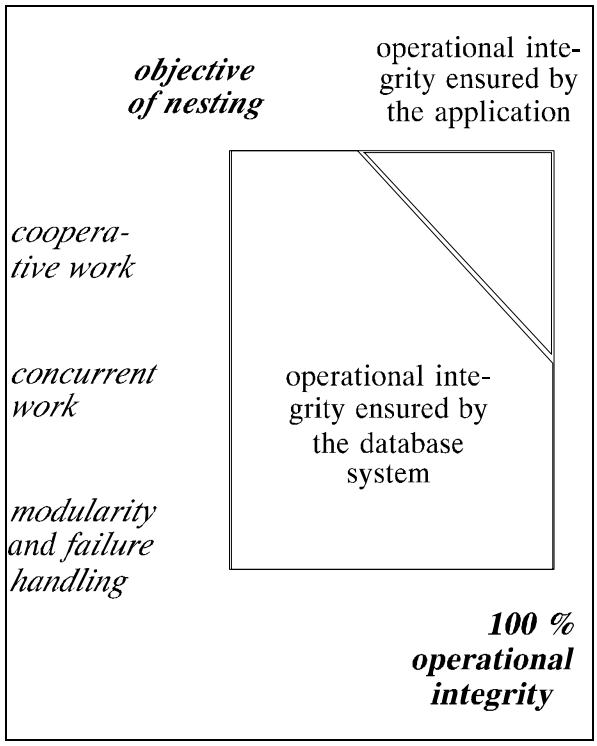


Fig. 2.2: Motives for the nesting of transactions

Our tool kit approach enables the DBI to assemble application-specific transaction managers which can individually be suited to the varying requirements of different application areas. Special emphasis was placed on the support of different forms of cooperative work. But, on the other hand, if these concepts are extensively used some burden is placed on the DBI. He has to ensure that the application is 'safe', which means that he has to provide means which allow the application to take on the responsibility for that part of the integrity preservation which was removed from the database system.

Figure 2.3 summarizes the basic 'tree terminology' that will be used in the remainder:

root or **top level transaction**: root of the transaction hierarchy; here: T5

superior, ancestor: each transaction on the path of a given transaction to the database (ancestor: including the given transaction itself); for example, T19, T10, T5 form the set of superiors of T31 (ancestor: plus T31 itself)

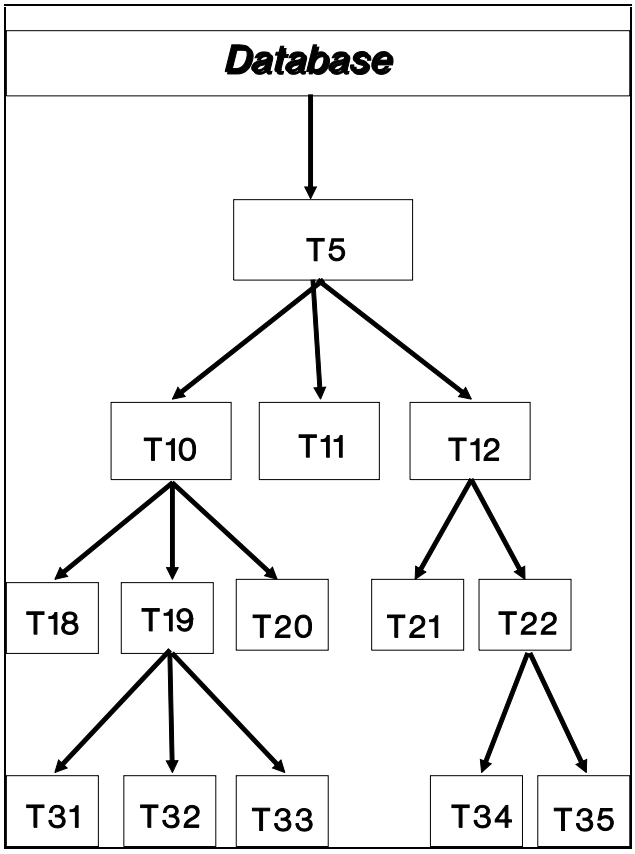
inferior, descendant: each transaction which is part of the subtransaction hierarchy spanned by a given transaction (descendant: including the given transaction itself); for example, T21, T22, T34, and T35 form the set of inferiors of T12 (descendant: plus T12 itself)

parent transaction: immediate superior of a transaction; for example, T5 is the parent transaction of T10 (T11, T12)

child transaction: immediate inferior of a transaction; for example, T10, T11, T12 are child transactions of T5

sibling: any other child transaction of the parent transaction of a given transaction; for example, T19, T20 are siblings of T18

leaf transaction: transaction which has no inferior; here: T11, T18, T20, T21, T31 - T35



non-leaf (or **inner**) **transaction:** transaction which has at least one inferior; here: T5, T10, T12, T19, T22

ancestor chain: the set of all ancestors of a given transaction (including the database); for example, the ancestor chain of T34 consists of T34, T22, T12, T5 and the database.

descendant tree (sometimes also called **sphere** of a transaction)**:** the set of all descendants of a given transaction; for example, the descendant tree of T12 consists of T12, T21, T22, T34, and T35

Fig. 2.3: Basic tree terminology

2.3 ACIDity properties revisited

The above discussion clearly indicates that the ACID principle has to be interpreted in a different way in the context of nested transactions.

☞ *Atomicity*

One reason for the introduction of nested transactions has been that they provide an appropriate control structure to achieve more graceful and flexible units of recovery, namely subtransactions. From the parent transaction's point of view a child transaction achieves the all-or-nothing type of execution; for instance, if subtransactions abort they must not affect the outcome of their parent transaction (or any other transaction). Child transactions correspond to actions on the level of their parent transaction. Therefore, child transactions are natural units of **in-transaction recovery** ([HäRo87-1+2]). Thus, a parent transaction may skip an (unsuccessful) (subtrans)action and replace it by a different one

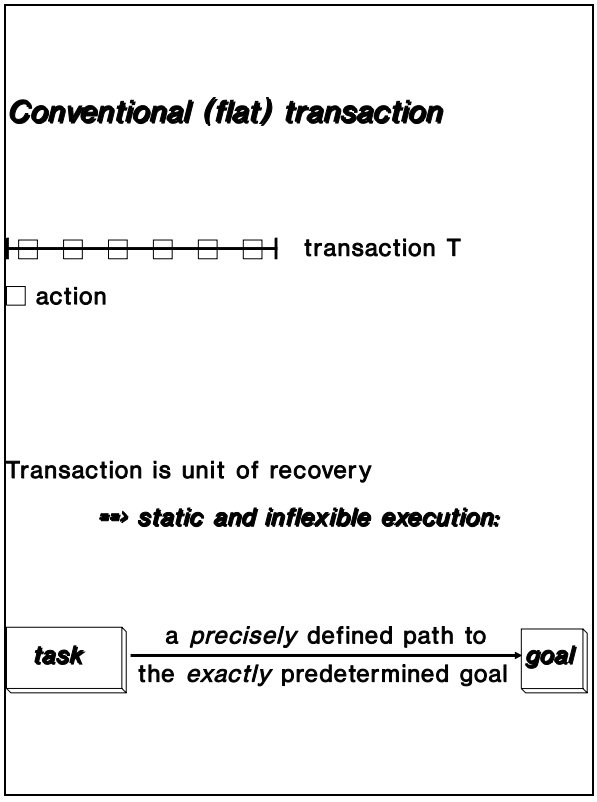


Fig. 2.4: Static course of flat transactions

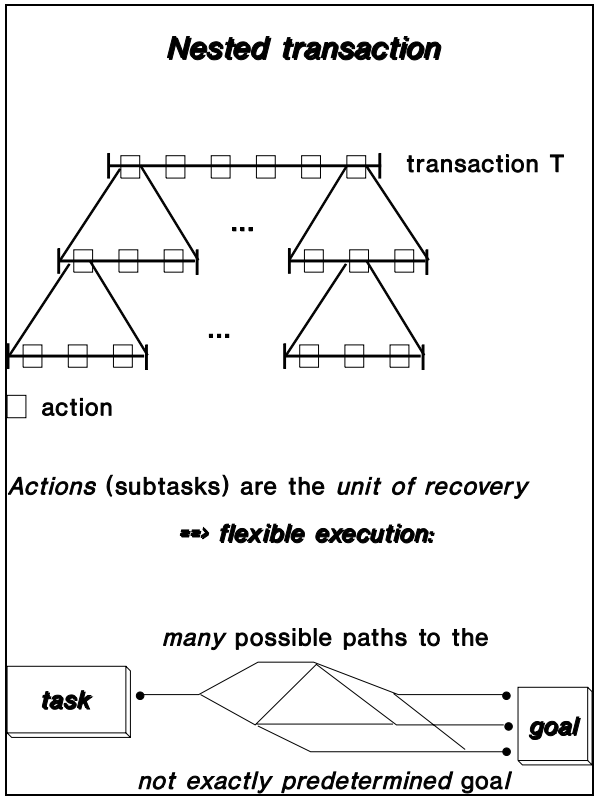


Fig. 2.5: *Flexible* course of nested transactions

or it may perform (subtrans)actions with a different (modified) task. Altogether, we gain the freedom to change the course of a transaction dynamically (see Figure 2.5). Conventional (flat) transactions are described by a static (predefined) list of actions and, therefore, can only commit successfully if all of their actions are executed correctly (see Figure 2.4). In contrast, the concept of nested transactions not only allows a (sub)transaction to dynamically determine how its goal can be achieved but even allows it to redefine its goal (see Figure 2.5).

☞ *Consistency*

Due to the nature of the task of subtransactions it would be unnecessarily restrictive to require global consistency to be preserved by a subtransaction. For a subtransaction, running at a particular level of the hierarchy, it is sufficient to only observe the normally weaker consistency constraints specific for that level; for instance, subtransactions have to preserve level-specific consistency instead of global consistency. The respective constraints are usually specified by the subtransactions' parent for the subtransaction and its siblings.

☞ *Isolation*

The interpretation of isolation gives by far the most rise to controversial and heated debates in the context of nested transactions. Some people argue that strong isolation of subtransactions is a mandatory feature even within nested transactions (primarily for reasons of not provoking the danger of (unrestricted) rollback propagation). Others put forward the objection that the notion of cooperative transaction has been devised not only to provide the same intuition as conventional transactions but also to support the requirements of cooperative work. Cooperative work, however, may mean that transactions are associated with user groups, where the transactions within a group need to employ a different concurrency control policy among themselves than with respect to transactions in other groups. This results in the necessity to relax concurrency control measures within a group; for example, allowing simultaneous updates to multiple instances of an object or allowing reads to uncommitted updates. Among groups, however, a stricter policy such as serializability may still be worthwhile. A compromise seems to be to at least require

top level transactions to be strictly isolated and to possibly provide weaker properties for subtransactions.

Altogether it can be stated that strict isolation is still an important property in the context of nested transactions but has to be supplied by weaker criteria. However, the problem with weaker criteria is that a generally accepted equivalent to the consistency levels of traditional concurrency control ([Gray79]) is not within sight. A support of cooperative work must inevitably remove some of the responsibility for the correct and consistent processing of data from the database system to the user or application; for instance, the system has to rely on the help of the user or application to ensure consistency. And, of course, it has always been a heated debate how reliable users or applications are and whether a reliable system should depend on users' decisions at all.

With respect to the tool kit we do not need to contribute to this discussion. The tool kit provides a kind of framework for the realization of application-specific transaction managers. Therefore, it has to offer facilities for the support of cooperative work (as well as for the realization of strict isolation). However, it is left to the designers of the specific transaction managers to make use of special features and to equip the basic components with adequate semantics. In this sense, the tool kit is on a semantically lower level.

- *Durability*

  Commit of a subtransaction and persistence of its results are conditional subject to the fate of its superiors (see atomicity). Even if a subtransaction commits, an abort of one of its superiors will undo its effects. Like consistency durability is level-specific. All updates of a subtransaction become permanent at the latest when the enclosing top level transaction commits.

The flexibility of a given transaction model depends on the way these four properties are interpreted (or realized). Thus, the tool kit has to offer as much flexibility and generality as possible and leave it to designers of the application-specific transaction managers to restrict this generality to an adequate level.

2.4 Fundamental rules of Moss' approach

According to Moss' model ([Moss81]) a nested transaction consists of a number of subtransactions. Every subtransaction again may be composed of a number of subtransactions and so on. Subtransactions are the unit of recovery; for instance, they are the unit for an all-or-nothing type of execution. Consequently they do not force the parent transaction to abort if they abort. An aborted child transaction does not leave any marks on the level of its parent transaction; for instance, it leaves the parent transaction as if it had never existed at all. A child transaction must commit before its parent transaction commits. Commit of a child transaction and durability of its results are conditional subject to the fate of its superiors. Even if a child transaction commits, aborting of its superiors will undo its effects. All updates of a subtransaction become permanent only when the enclosing top level transaction commits. These rules are also valid in our approach.

Moss distinguishes between two types of transactions, non-leaf transactions and leaf transactions. Only leaf transactions are allowed to perform operations on objects while non-leaf transactions only serve as a kind of database for their inferiors. With this assumption in mind concurrency control for nested transactions has to obey the following rules:

1. *Meaning of locking*

   For a transaction to perform an operation, the transaction must hold the lock corresponding to that operation.

2. *Strict two-phase locking*

   To avoid rollback propagation a transaction has to be strict two-phase; for instance, no locks can be released until the transaction has either committed or aborted.

3. *Exclusion*

   If a transaction requests a lock, the request can be granted only if all holders of conflicting locks (if any) are ancestors of the requesting transaction.

4. *Release*

   When a transaction succeeds, all its locks are either inherited by its parent transaction or released in case the transaction is top level. When a transaction aborts, all its locks are

discarded. If any of its superiors hold a lock on the same object, they continue to do so.

5. *Uniformity*
   All transactions follow these rules.

3 Basic concepts and fundamental rules of the tool kit approach

Before we start to discuss our approach some simplifications are to be clarified first:

- ☞ The current version of the tool kit concentrates on locks as the means of concurrency control.
- ☞ The following discussion focuses on long-duration transactions. Other transaction types (for example, conventional (short) transactions) are not considered.

3.1 Basic concepts of the tool kit approach

A lot of approaches to nested transactions in literature concentrate on modularity, failure handling and concurrent execution of subtasks. Support of cooperative work is only second-rate. However, without question, this feature is a substantial requirement of most non conventional application areas. One essential objective of our approach is to provide such facilities. In this sense our approach is, among others, an extension and generalization of the concept of nested transactions in the direction of a support of cooperative work.

In this section we will explain the basic terms and concepts of our approach. Moreover, we will compare our approach with Moss' approach to nested transaction and will show how we extended and generalized the rules of Moss' model in order to provide more facilities for cooperative work.

The first general concepts of our approach are, that we treat a transaction (type) as an abstract data type, that we associate an owner with each long-duration transaction, and that we add a unique transaction identifier to each long-duration transaction (type). Moreover, for the follow-

ing discussion we assume that each long-duration transaction is equipped with an object pool (see below). However, it has to be clarified, that this is not a requirement of our approach but is simply assumed for reasons of simplicity.

Object pool

Since in our approach a long-duration parent transaction serves as a kind of database for its child transactions we make this explicit by associating with each transaction an object pool. An **object pool** is nothing else but a container for the collection of objects which are associated with the transaction at a given point of time. However, this does not mean that the object pool has to contain each object physically. Instead, it may only manage the object identifiers and the locks imposed on these objects.

The introduction of an object pool points to a substantial difference between Moss' approach and our approach (at least on the conceptual level). In Moss' approach the unit of transfer between transactions is the lock (see Figure 3.1). If a transaction acquires a lock it gets the right to perform the corresponding actions on the object (in the database). If transactions want to perform conflicting actions on an object these actions have to be executed one after the other in most approaches to nested transactions. In our approach, if a transaction T acquires a lock it not only gets the lock but also a copy of the object; for instance, the copy of the corresponding object is inserted (at least logically) into the object pool of T. The unit of exchange between transactions is the pair **copy of the object/lock on the object** (see Figure 3.2). The reason for this is twofold:

1. We want to support the coexistence of different states of the same object, for example, during a design process. Different levels within a transaction hierarchy often represent different states of a task. A copy of an object at a higher level of the hierarchy usually represents an older but, as such, "stable" state of the object while a copy at a lower level represents a younger but still "uncertain" state of the object.

2. In order to support cooperative work our approach has to permit a controlled concurrent execution of conflicting actions on different copies of the same object (we will discuss this topic later). Therefore, it can happen that different copies exist in the object pools of diffe-

rent transactions. Of course, these different copies need to be merged to a single, consistent copy at a later time.

In the following, if we use the term object or lock (in connection with our approach) we always mean the pair copy of object/(new) lock.

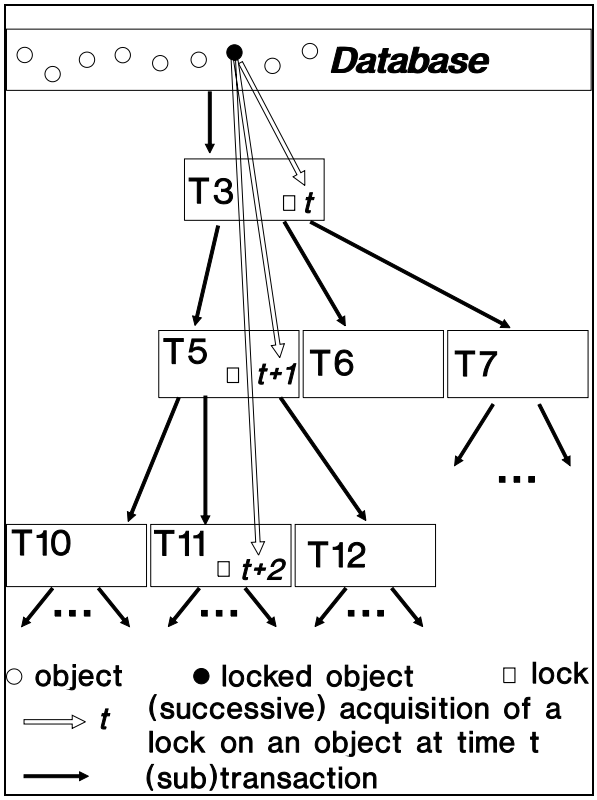


Fig. 3.1: Transfer unit (*lock*)

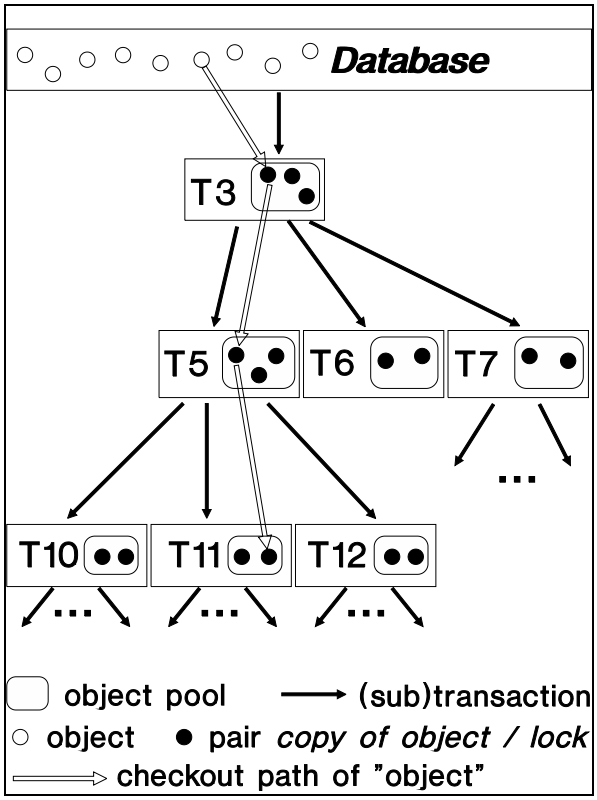


Fig. 3.2: Transfer unit (*copy of object/ lock*)

Abstract data type

In our approach a transaction type is seen as an abstract data type; for instance, it is characterized by the operations which come with the type. A number of operations are common to all types; for example, operations to suspend and continue the transaction, to acquire or release objects, or to commit it. However, these operations may be implemented in different ways (depending on the underlying transaction type). Other operations are only specific to some transaction type since they come with the features by which a transaction type differs from other transaction types.

Owner

Each transaction is associated with an owner who is nothing else but a user or a user group. Only members of the associated user group (in the following called **owners** of a transaction) are allowed to perform operations on the transaction. Moreover, every transaction T can only be active on behalf of one owner at a time. The owner who has control of T can perform any operation associated with T until he explicitly suspends the transaction. An owner may run several transactions at the same time. This enables him to work on different subjects simultaneously, and to use transactions alternatively. However, as an active user U of a transaction T^A, U does not have access to objects of a transaction T^B which is not an ancestor of T^A even if U is an owner of T^B (an exception to this rule is an explicit cooperation which will be discussed later on). For example, let us consider the transaction hierarchy of Figure 3.1. If U currently works with transaction T10 he is not allowed to acquire objects (in his function as a user of T10) from any transaction of the descendant tree of T7 even if he is also an owner of this transaction. User groups are allowed to dynamically change; for instance, new members can be added to or removed from the user group at any time.

Transaction identifier

Each transaction (type) is assigned a unique transaction (type) identifier by the system. Additionally, users may establish a unique transaction (type) name by which they can uniquely refer to the transaction (type).

3.2 Fundamental rules of the tool kit approach

In order to be able to exploit the fundamental semantics of advanced database applications the serializability-based transaction models need to be replaced by models which make it possible to express semantics beyond serializability. Moreover, especially in case of a support of cooperative work, a transaction manager should be able to carefully deal with human or applications involvements in order to ensure correctness of the system as a whole. With respect to the locking approach there are several ways to achieve these goals, among them the following:

1. *Weaker lock protocols*

   By the use of weaker lock protocols transactions may be allowed to exchange data or to release data before end-of-transaction (EOT).

2. *Weaker lock modes*

   By providing lock modes which explicitly facilitate a higher degree of concurrent work on data the concurrency control component may be able to exploit application-specific semantics.

The second alternative will be discussed in the section about lock modes. In this section we will concentrate on alternative 1. We will start with an example which indicates that an early release of data has to be handled with care.

To avoid rollback propagation most approaches to nested transactions require (sub)transactions to act strictly two-phase. Alternative 1 wants an application, in certain situations, to run (sub)transactions which employ weaker lock protocols, like simple two-phase locking, especially if the application is able to avoid or drastically restrict rollback propagation. However, an early release of objects has to be handled with care since it can violate the two-phase principle as the following discussion will show.

With respect to the acquisition of objects the usual proceeding within nested transactions is that a subtransaction T^S can only acquire objects from its ancestors. To not endanger serializability an object release is treated more restrictively. The commonly applied **object (lock) release rule** requires T^S to upward inherit all its objects (locks) to its parent transaction. While the more restrictive object release rule avoids any consistency problems the more liberal object acquisition rule may lead to consistency problems if subtransactions of a nested transaction are allowed to run other lock protocols than the strict two-phase lock protocol it should be mentioned that some approaches in literature do not treat this problem adequately).

In the given scenario (see Figure 3.3) several subtransactions of a nested transaction work, one after the other, on the same object O. First, transaction T5 acquires O from T3 in its growing phase (1), modifies O, and releases the modified object O'. This implies that T5 is now in its shrinking phase. According to the object release rule O must be inherited by the parent trans-

action of T5, here T3 (2). Next, transaction T6 acquires (3) (this results in the dependency T5 --> T6) and modifies O', and releases the modified object O" (4). Again, T3 inherits O". Finally, T12 acquires O" from T3 and modifies it (result: O'"). Since T12 is a child transaction of T5 O'" will be upward inherited to T5 (6). However, T5 is already in its shrinking phase and, therefore, no longer in a position to acquire any object. This situation does not only torpedo the principle of two-phase but it, moreover, violates the principle of strict isolation since T5 gains an insight into the modifications carried out by T6 (which corresponds to the dependency T6 --> T5 which, of course, conflicts with the dependency T5 --> T6)).

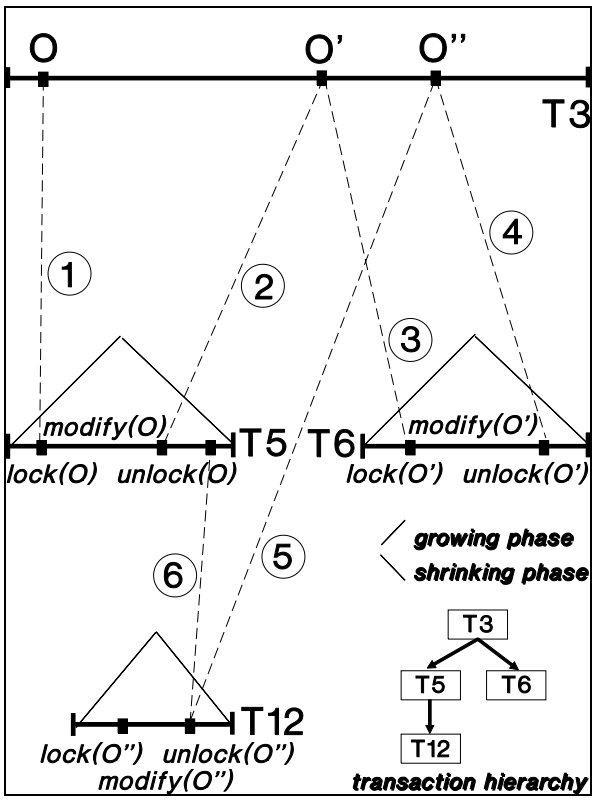


Fig. 3.3: Example scenario

Stepwise transfer

If we take a closer look at Figure 3.3 it becomes clear that the problem only arises because T12 acquires an object from a superior (different from its parent transaction) without considering the status of its parent transaction T5 (which is already in its shrinking phase). But since T5 inherits all objects from its child transactions it has to be ensured that such a situation cannot occur. The problem can be solved in several ways. One solution is to force each child transaction to consider the status of all transactions in its chain of ancestors up to that transaction from which it wants to acquire an object. Only if the status of each of these transactions permits the acquisition of the object the object can be granted. A more restrictive solution is to only allow a child transaction to acquire objects from its parent transaction. We chose a solution in between which combines the advantages of both approaches. The general principle of our approach is that a transaction T can only acquire objects from its parent transaction. However, if T needs an object O from some other ancestor T^A this is realized by a stepwise check-

out of O from the object pool of T^A via the transactions on the path to T (successions of downward check-outs). If, for example, T12 wants to acquire an object O from the parent transaction of T3 such a demand is satisfied by a stepwise check-out of O from the parent transaction of T3 to T3 to T5 to T12. On each level the concurrency control scheme of that level (transaction) is employed to safely transfer O to its destination. The set of objects which is accessible to a transaction is defined by its access view which will be discussed in more detail later on.

In a similar way we define the stepwise check-in of an object O. This means, that we do not require a transaction T to pass O onto its parent transaction. T may transmit O to some other superior T^S if the status (lock protocol) of each transaction on the path to T^S permits such a proceeding (more precisely, if T^S belongs to the release view of T. The release view will be discussed later on).

The installation of the concept of stepwise transfer has the consequence that our approach realizes *downward inheritance* while most other proposals realize *upward inheritance* (for example, [HäRo87-1+2], [KLMP84], [KoKB85], [Moss81]). Let us assume that a transaction T wants to acquire an object O from a superior T^S. With **downward inheritance**, O is inserted into the object pool of each transaction on the path from T^S to T *when it moves downward*. With **upward inheritance** the parent transaction inherits the lock on O only after the child transaction has committed; for instance, *when the object (lock) moves upwards*. In both cases O will be added to every object pool on the path from the database to the deepest transaction which has acquired O, with upward inheritance, however, at a later point of time. Downward inheritance has a rather advantageous feature. It guarantees that the object pool of a parent transaction contains all objects of its descendant tree (though, not inevitably the newest status of the object) with the exception of those objects which were newly created by an inferior. Therefore, if a transaction T searches for a special object O it can definitely be concluded that no inferior of the parent transaction of T possesses O if the parent transaction does not. Moreover, if we need to check a lock we only need to consider the ancestor chain up to the first transaction whose object pool contains the object. If, here, the object is not locked in an incompatible mode the lock request can be granted.

Besides the stepwise transfer another fundamental concept of our approach is the two-stage control-sphere.

Two-stage control-sphere

The underlying principle of the **two-stage control-sphere** is that a parent transaction is only responsible for the correct coordination and execution of the work (task) on *its* level. It may define subtasks and start child transactions to deal with these subtasks. Each child transaction, again, is by itself responsible for the correct coordination and execution of its task and, therefore, can decide autonomously, how this task can be executed best. In other words, the characteristics which were established on the level of the parent transaction are only valid for its child transactions. The child transactions, in turn, may establish a different environment for their child transactions (see Figure 3.4). The two-stage control-sphere establishes the foundation for the possibility of executing transactions of different types within one transaction hierarchy.

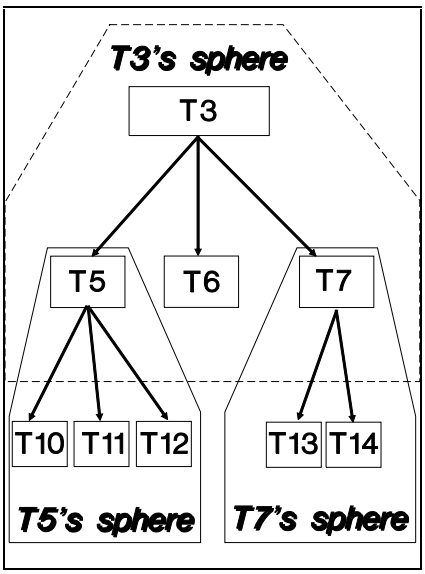


Fig. 3.4: *Two-stage* control-sphere

4 Characteristics of transaction types

The various characteristics which make up a transaction can be subdivided into two parts (see also [Unla91]): characteristics which describe the internal representation of a type and characteristics which describe its behavior. In this section we will concentrate on the second part. The structure of transaction types will be outlined in section 8.

4.1 Concurrency control scheme

The two stage control-sphere stands for the possibility that each transaction type T can establish its own concurrency control scheme for its object pool. This means, that each T can inde-

pendently determine according to which rules descendants of T can acquire objects from T. Such a free choice of concurrency control scheme is possible since the stepwise transfer of objects guarantees that each transaction which is involved in the stepwise transfer needs to use the concurrency control scheme which is required by its parent transaction. For example, in Figure 3.5, if transaction T3 wants locks to be used to synchronize access to its object pool its child transactions T5, T6, and T7 need to run a lock protocol if they want to acquire objects from T3. However, each of the child transactions may employ its own type of lock protocol, for example, T5 may run two-phase locking with predeclaring, T6 simple two-phase locking, and T7 strict two-phase locking (for a discussion of different types of lock protocols see later or, for example, [BeHG87]). On the other hand, each child transaction may run a different concurrency control scheme for its own object pool. T5, for instance, may run optimistic concurrency control (OCC) (c. f. [UnPS86]) with the consequence that the child transactions of T5 (T10, T11, T12) have to run an optimistic concurrency control scheme if they want to acquire objects from T5. For example, if T12 wants to acquire an object O from T3 the stepwise transfer ensures that O is first transferred from the object pool of T3 to the object pool of T5 by using a lock protocol and then from the object pool of T5 to the object pool of T12 by using OCC.

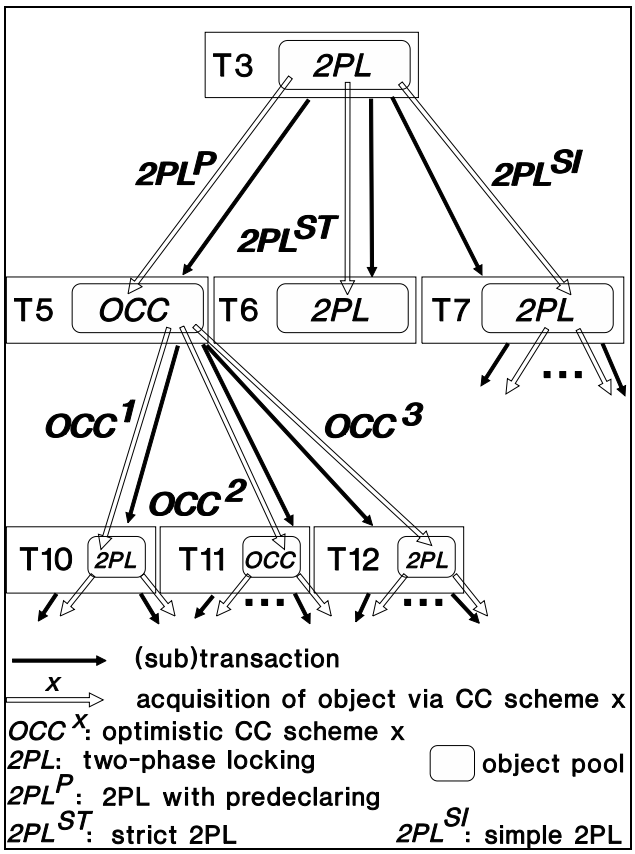


Fig. 3.5: Use of *different* concurrency control schemes within a nested transaction

To summarize, each transaction T needs to establish two concurrency control schemes. The first scheme lays down according to which rules T can acquire objects from its parent transaction. This scheme must be in accord with the concurrency control scheme which the parent transaction runs on its object pool. The second scheme lays down which concurrency control

measures are to be applied to T's own object pool. Here, T is given free hand to install any scheme which is supported by the tool kit.

In this paper, however, we will only concentrate on lock protocols. Currently, we support the following (well-known) types of lock protocols:

- ***two-phase***

  The two-phase locking protocol is the most common protocol in conventional database systems. It defines a transaction to consist of two phases, a growing phase and a following shrinking phase. One can identify several types of growing phases as well as shrinking phases. Each type of growing phase can be combined with each type of shrinking phase.

  - **growing phase**

    - *simple*

      The growing phase starts with the beginning of the transaction and ends as soon as the first object is released.

    - *predeclaring (preclaiming)*

      All objects which a transaction needs are acquired at the beginning of the transaction. With it the growing phase ends. No further objects can be acquired. Predeclaring prevents deadlock to occur.

    - *extended predeclaring*

      As before, but additional objects can be requested as long as no lock was released by the transaction (the transaction is not yet in its shrinking phase). If the objects are not locked in an incompatible mode they are granted. Otherwise, the lock request is rejected (a corresponding message is sent to the transaction) but the transaction is not blocked (no waiting for locked objects).

  - **shrinking phase**

    - *simple*

      Objects can be released one by one. During the shrinking phase no further objects can be acquired from the parent.

- *strict*

  All objects are released at once at the end of the transaction. A strict two-phase locking protocol prevents rollback propagation to occur.

- ***non two-phase***

  This protocol allows objects to be acquired and released in arbitrary order. Non two-phase is especially important if some kind of cooperative work is to be supported.

As already mentioned, we treat a transaction as an abstract data type. Therefore, the acquisition of an object of the parent is realized by a **check-out operation** while the transfer of an object from a child's object pool to the parent's object pool is realized by a **check-in operation**.

Conceptually, our approach requires a separate lock manager for each transaction. In practice, it is undesirable to install multiple lock managers, because of excessive performance overhead. Therefore, we implemented a mechanism similar to the hierarchical transaction naming scheme (see [KoKB88]) which allows a single lock manager to simulate the lock managers required by our approach.

The two-stage control-sphere requires that the acquisition of an object from a superior (other than the parent) is realized by a stepwise check-out. However, whether an object is accessible to a transaction depends on the states of the transactions which are affected by a stepwise check-out. In general, the objects which are accessible to a transaction are described by the access view of a transaction.

4.2 Object visibility (access view and release view)

An **access view** defines the collection of objects which are, in principal, accessible to a transaction T. "In principal" means that an object may not be accessible to T at a given point of time due to the fact that it is being locked by another transaction in an incompatible lock mode.

In the context of this paper we associate with the notion of view the default view provided by the system. No attention is paid to the fact that this view may be restricted by access control mechanisms.

Depending on its position in the transaction hierarchy each transaction T has an individual **access view**. It consists of all objects of the object pools of the chain of ancestors up to the first object pool which is check-out blocked (excluding this object pool) or, if there is none, up to the public database (including the public database). The object pool of the parent transaction T^P of a transaction T^A is **check-out blocked** if at least one of the following two restrictions is valid:

1. *implicit restriction*

   T^A runs a two-phase lock protocol and is already in its shrinking phase; for instance, T^A is no longer in a position to acquire objects from T^P.

2. *explicit restriction*

   T^P explicitly prohibits its inferiors to acquire objects from its object pool or any other object pool of the ancestor chain of T^P; for instance, T^P is allowed to explicitly install a check-out blockade with the consequence that the access views of its inferiors are restricted. If, nevertheless, a child transaction T^C of T^P needs some object O of an object pool of the ancestor chain of T^P this can only be achieved if T^P explicitly transfers O to its own object pool, therefore, making O directly accessible to T^C.

The implicit version of the access view has to be considered to prevent consistency violations as they were shown in Figure 3.3 from occurring. The explicit restriction allows a superior to restrict its subtransactions' effects on objects by limiting the number of objects accessible to them (at least for the time the explicit checkout blockade is valid).

The access view of a transaction T changes dynamically during the lifetime of T since, at any point of time, an ancestor of T may start its shrinking phase or install/remove a check-out blockade.

In Figure 4.4 the access view of T38 consists of the objects of the object pools of T38, T26, T15, T8, T4, T1, and the database if neither T4 nor T1 are in their shrinking phase nor an ancestor has established an explicit check-out blockade (T8 cannot be in its shrinking phase since it runs the strict two-phase lock protocol) (please do only concentrate on such features of Figure 4.4 which are relevant to this example; other features will be explained later on). As soon

as T4 starts its shrinking phase the object pool of T1 and the database are no longer accessible to T38. If T15 installs an explicit check-out blockade the access view of T38 is restricted to the objects of the object pools of T38 and T26.

If a transaction T acquires an object O from an object pool of its access view, O is (at least logically) added to all object pools on the path from the original object pool of O to T. This implies that the access view of T may contain different states of an object (since higher object pools may contain "older" versions of an object). Let us again consider transaction T38 of Figure 4.4. If T38 wants to acquire object O from transaction T15, O is also part of the object pools of T8, T4 and T1 (unless O was newly created by some descendant of T15). Since, for example, T8 may have modified O before it was transferred to T15, different states of O exist within the access view of T38. This leads to the problem of a "lost update" if T38 acquires O from the object pool of T4. Therefore, by definition, the access view of T38 only contains one instance of O, namely the instance of the first ancestor of T38 whose object pool contains O. In our example this is the instance of the object pool of T15 if O currently does not belong to the object pool of T26.

The notion of access view is similar to the notion of *view set* in [ChRa90]. However, the view set of a nested transaction in [ChRa90] is defined as always containing each object which was acquired by some ancestor and all objects of the database. No attention is paid to the fact that the view set must decrease in case an ancestor of a given transaction starts its shrinking phase.

Similar to the notion of access view we define the notion of **release view**. A release view determines the object pool up to which an object of a given transaction T can be released at most (if no other lock on the object prevents this). The release view consists of all object pools of the ancestor chain up to the first ancestor which is check-in blocked (excluding this object pool) or, if there is none, up to the public database (including the public database). An object pool of the parent transaction T^P of a transaction T^B is **check-in blocked**, if at least one of the following two restrictions is valid:

1. *implicit restriction*

   T^B runs a two-phase lock protocol and is still in its growing phase, for instance, T^B is not allowed to insert objects into the object pool of T^P.

2. *explicit restriction*

   T^B explicitly prohibits the insertion of objects into the object pool of its parent transaction; for instance, T^B is allowed to explicitly install a check-in blockade. T^B may install a check-in blockade to prevent its inferiors from releasing objects to a superior of it. Due to the nature of check-in and check-out, a check-out blockade works on the level it was defined while a check-in blockade only becomes valid on the level of the parent transaction (since, otherwise, a child transaction would not be able to release any object at all); for instance, in case T^B installs an explicit check-in blockade a child transaction of T^B can still insert an object into the object pool of T^B but not in the object pool of any superior of T^B.

In Figure 4.4 the object pool of T1 is check-in blocked for T4 and any inferior of T4 as long as T4 is in its growing phase. The object pool of T8 is check-in blocked for its inferiors since T8 runs the strict two-phase lock protocol. In general, an object pool of a transaction T which obeys a two-phase lock protocol is either check-out blocked (if T is in its shrinking phase) or check-in blocked (if T is in its growing phase).

A transaction can change any time its parameter value for check-in or check-out blocked; for instance, an explicit check-in or check-out blockade can dynamically be established or released.

The implicit version of check-out blocking always propagates from higher levels of the transaction hierarchy to lower levels (an access view can never increase, only decrease) while the implicit version of check-in blocking always propagates in reverse order (since an ancestor can only start its shrinking phase, never a new growing phase). This feature makes it possible to determine, at any point of time, up to which level incorrect objects may have propagated at most; for instance, if an already released object needs to be corrected it is definitely determinable up to which object pool this object may have propagated at most (up to the first object pool which is implicitly check-in blocked (not check-in blocked by an explicit restriction).

4.3 Task

Some approaches to nested transactions require work on objects to be exclusively performed in the leaf transactions of the transaction tree. Non-leaf transactions only serve as a kind of database for their child transactions. However, in many applications it is desirable that a non-leaf transactions T can also perform operations on its objects. Of course, in such a case the

work of T on its objects has to be synchronized with the work of T's child transactions on these objects. In literature, at least two solutions for this problem are proposed:

1. In [KLMP84], [UnSc89-1] the object pool of a transaction T is subdivided into a part which is only accessible to T (called **private database**) and a part which is accessible to inferiors (called **(semi-)public database**). When T acquires an object it is, by definition, copied into the private object pool. Objects can be made accessible to inferiors by explicitly moving them from the private to the semi-public object pool (see Figure 4.1).

2. Härder and Rothermel propose a concept which is based on a releasing (weakening) of lock modes for descendants (they call it **downgrading** a lock) ([HäRo87-1]). A transaction T, holding a lock in mode M, can downgrade the lock to a mode M'. By this, descendants are allowed to access the objects in a mode which is compatible to M'. Transactions which do not belong to the descendant tree of T still see the more restrictive lock mode M. Downgrading of locks makes it possible for T to put objects at the disposal of its descendant tree while still protecting them from incompatible access by other transactions (see Figure 4.2).

Although these approaches clearly indicate a solution to the problem we decided to do it in a way which makes it possible to solve the problem by means of concepts already introduced (instead of integrating new concepts). As already mentioned, our approach relies on the concept of abstract data type. Therefore, a transaction type is defined by its behavior. With respect to access operations we strictly separate between two groups of operations. Checkout/-in realize the (only) connection to superiors since they allow a transaction T to *acquire objects from* or *to release objects to* a *superior*. If T wants to manipulate objects it has to use operations like read, modify, delete, insert. Since T is only allowed to manipulate objects of its *own* object pool these operations are only defined on T's object pool. In our terminology the term **service transaction** denotes a transaction which only serves as a database for its child transactions. Therefore, it only provides the checkout/-in operations.

Operational transactions allow their users to furthermore *operate* on the objects of their object pools. Therefore, they additionally provide the set of operations for object manipulations. However, the tool kit does not realize these operations on the level of the operational transac-

tion T itself. Instead, they are associated with a special, implicit subtransaction T' of T (see Figure 4.3). Since T' is treated as a child transaction of T operational integrity is ensured by the usual mechanism (concurrency control algorithm of T). If T runs a lock protocol, T', by default, obeys the non two-phase lock protocol. Therefore, a lock can be acquired or released any time. In case of a service transaction the implicit subtransaction is simply not installed.

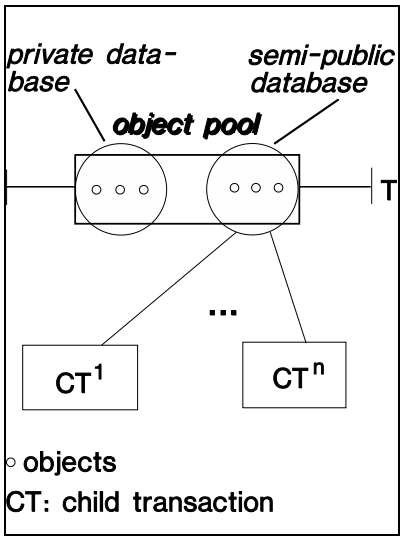


Fig. 4.1: *Private/semi-public* database

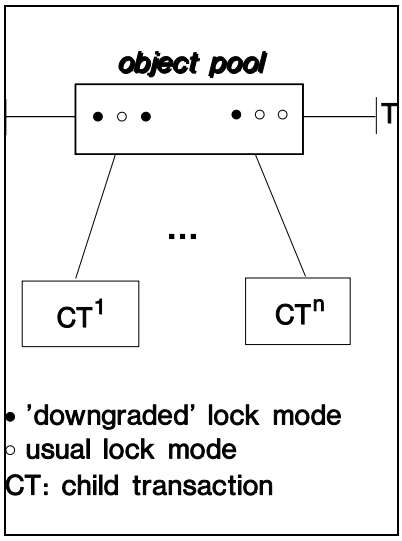


Fig. 4.2: Downgrading of locks

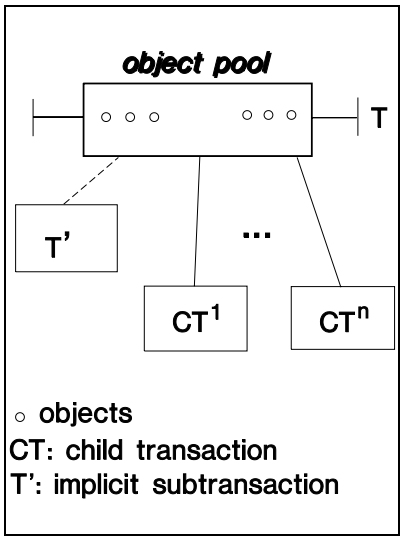


Fig. 4.3: *Implicit* subtransaction T'

Whenever a user of an operational transaction T wants to directly apply an operation to an object of the object pool of T this request is directed to T' (of course, transparent to the user or the application). T', per default, will try to locked the requested objects and perform the manipulations. Afterwards, the objects are released immediately. This, of course, gives the user the perfect illusion that he directly performs his operations on the object pool of T.

4.4 Concurrent execution of (child) transactions

The decomposition of a unit of work (complex task) into subtasks and the possibility to execute subtasks as subtransactions strongly suggest to execute (sub)transactions concurrently whenever possible. With respect to a parent transaction and its child transactions two kinds of intra-transaction parallelism can be identified, parent/child parallelism and sibling parallelism. With **parent/child parallelism** a parent transaction is allowed to run in parallel to its child transactions while in case of sibling parallelism siblings are allowed to run concurrently. The

combination of these both kinds of parallelism leads to four levels of intra-transaction parallelism (see also [HäRo87-1]):

1. *no intra-transaction parallelism at all*

   If neither parent/child parallelism nor sibling parallelism is allowed then, at any time, at most one transaction can be active. This corresponds to *serial computation*s similar to the ones that are achieved by procedure calls.

2. *only sibling parallelism*

   Such a situation is given if the parent transaction is a service transaction, therefore only provides its child transactions with data.

3. *only parent/child parallelism*

   This combination does not make much sense and, therefore, is unlikely to occur in practice.

4. *unrestricted intra-transaction parallelism*

   This combination provides the highest degree of concurrent work since it permits arbitrary intra-transaction parallelism. It means that the parent transaction as well as (at least some) child transactions are operational transactions.

While combinations 2 to 4 can be modelled within the current framework of our tool kit combination 1 cannot. However, it represents an important special case. If both parent/child parallelism and sibling parallelism are prohibited no concurrency control measures need to be established on the level of the object pool of the parent transaction (since no concurrency is possible). This, of course, frees transaction processing from some burden and makes it much more efficient. To be able to exploit this increase in efficiency, we added a new characteristic which lays down whether a transaction prohibits any kind of intra-transaction parallelism. Again, this characteristic is only valid on the level of the parent transaction and its child transactions. A child transaction may stipulate that its child transactions can execute concurrently.

4.5 Explicit cooperation (collaboration)

Our approach supports the possibility of explicitly installing a **direct cooperation** between two or more transactions from different branches of the transaction tree. More specifically, we want to support a controlled lending, transfer, or exchange of objects. A direct cooperation can only be permitted if some conditions are fulfilled. Let us assume that T^A and T^B are two transactions which want to cooperate directly. Let T^R be that transaction of the transaction hierarchy which constitutes the root of the smallest subtree which comprises T^A as well as T^B. A cooperation between T^A and T^B can only be established if the paths from T^A to T^R and from T^B to T^R are neither check-out nor check-in blocked. This means that both transactions, T^A and T^B, are allowed to stepwise check-in objects in the object pool of T^R and, moreover, to stepwise check-out objects from the object pool of T^R to their own object pools (for instance, the object pool of T^R is part of the access and release views of T^A as well as T^B). This situation can only occur if all transactions on the paths from T^B to T^R and T^R to T^B (T^R excluded) support the non two-phase lock protocol. Only in case of a transfer of objects, for example, from T^A to T^B, can this requirement be diminished to the requirement that the path from T^A to T^R is not check-in blocked and the path from T^R to T^B is not check-out blocked. As long as a cooperation exists none of the involved transactions is allowed to establish a check-out or check-in blockade.

For example, in Figure 4.4 a cooperation can be installed between T39 and T42 since all transactions on the paths from T39 to T17 and from T17 to T42 run the non two-phase lock protocol. On the other hand, a general cooperation between any of the descendants of T8 and T9 cannot be installed since T8 as well as T9 run a (different type of) two-phase lock protocol. However, since T9 is always in its shrinking phase (since it runs the simple two-phase lock protocol with predeclaring) while T8 is always in its growing phase (since it runs the strict two-phase lock protocol) a transfer of objects, for example, from T42 to T38 is possible (all other transactions on the path from T42 to T4 and from T4 to T38 run the non two-phase lock protocol).

4.6 Serializability revisited

The common criterion for correct synchronization of flat transactions is serializability. In nested transactions at least the transactions along an ancestor chain may share some objects. This, of course, raises the question whether the criterion of serializability can also be applied to nested transactions. For the following discussion we will assume, that each (sub)transaction obeys the ACID principle. However, to be honest, it should be clarified here that the tool kit, in order to support cooperative work, provides weaker lock protocols (like non-two phase) as well as weaker lock modes (see later). Therefore, the following discussion is only applicable to our tool kit if the respective application-specific transaction manager does not exploit these weaker features of the tool kit.

Let us start with a relatively simple scenario. Given a parent transaction T^P of type service transaction and a bunch of child transactions T^{Ci} to T^{Cj} (nested transaction of depth 2). This situation corresponds to a conventional database systems where a number of transactions (T^{Ci} to T^{Cj}) work on a common database (T^P) (remember that the feature of stepwise transfer requires each transaction to only acquire objects from or to release objects to its parent transaction). Since T^{Ci} to T^{Cj} obey the ACID principle their execution is serializable; for instance, they appear on the level of T^P as isolated, atomic actions. For example, if all transactions run the strict two-phase lock protocol they are serializable at least in commit order. Now that we have defined a serialization order for the child transaction we have to insert T^P into this schedule. Since T^P is a service transaction it has not accessed any object (neither read (in the sense that some information was released to the outside world) nor modified any data). Therefore, it is legal to place T^P at the end of the serial schedule gained so far. This results in the following legal serial schedule:

S_1: [(T^{Ci}, ..., T^{Cj})T^P] (if T^{Ci} to T^{Cj} is the commit order of the child transactions).

Moreover, since T^P shielded its child transactions from any other transaction, therefore, guaranteeing that no other transaction was able to access any of the objects used by its child transactions in an incompatible way it can be assumed, without restricting generality, that no other transaction T must be placed somewhere within the transactions of S_1 in order to ensure serializability (or, the other way round: if there exists a serial schedule in which T is placed some-

where within S_1 then there also exists a serial schedule in which T is either placed before or after S_1). This, of course, means that we can simply replace the schedule S_1 by T^P. This, in turn, means that we can reduce our problem to find a serial schedule for a nested transaction to the problem of stepwise identifying the serial schedule for a parent transaction and all of its child transactions. Or, more specifically, the following algorithm works perfectly:

> Start at the level of the top-level transaction T^{TL}. Compute the serial schedule spanned by T^{TL}'s child transactions and add T^{TL} at the end of this schedule (as explained above). Now (recursively) take each of the child transactions and consider them as the top of the next (two-stage) nested transaction. Resume until each inner node of the nested transaction is expanded in the way described. All in all this corresponds to a (breadth first or depth first) expansion of the top-level transaction.

Of course, it is also possible to derive the serial schedule in opposite direction (bottom-up instead of top-down).

Now let us consider the more general case, that the parent transaction can be an operational transaction. While the child transactions represent transactions on the level on which they are performed they correspond to actions on the level of their parent transaction T^P; for instance, T^P is allowed to read or modify any of its objects any time (before an action (child transaction) was performed as well as afterwards). What does that mean? On the level of the child transactions we are able to identify a serial schedule (as was demonstrated above). However, on the level of T^P we cannot simply expand this schedule by adding T^P at the end of this schedule. In reality the set of actions which were performed by child transactions enriched by the set of actions which were not executed by child transactions altogether constitute T^P; for instance, T^P is represented by this extended set and, therefore, cannot be placed somewhere within the serial schedule of the child transactions. However, the argumentation of the simpler case above is still valid. T^P is an isolated unit which corresponds to an action on the level of its parent transaction. Therefore, we can identify for each inner transaction a serial schedule for its child transactions. But the next level is a higher level of abstraction and cannot be described in terms of the lower levels. Or, in other words, we can describe for each level of a nested transaction the serial schedule(s) of this level but we cannot identify one global schedule. This does not mean that a nested transaction is not serializable. It just means that each level of the nested

transactions represents an own level of abstraction. Therefore, it is, in general, impossible to identify a view which comprises all levels of abstraction.

4.7 Recovery

To exploit the advantages of nested transactions, recovery has to be refined and adjusted to the demands of the control structure of nested transactions. Moreover, it must be adapted to the requirements of the various applications. In application concurrency control semantic information about the objects and their operations is used in designing specific concurrency control measures to enhance concurrency within objects. Application-specific recovery can be designed along the same lines to exploit the semantics of the application in order to minimize the effects of transaction failures. Application-specific recovery may reduce the cost of recovery, for example, by tolerating partial failures or by supporting not only backward recovery but also forward recovery (c. f., [ChRa90], [ReWä91]). In the event of failure of transaction components, the failed portions can be isolated, allowing the rest of the transaction to proceed. Failed portions of transaction can be retried, compensated by attempting another alternative, or even ignored (as is indicated in Figure 2.5). Nested transactions naturally support user-controlled in-transaction checkpointing since the boundaries of child transactions may act as restart points. Even more flexibility for partial rollback of in-progress transaction can be gained by explicitly establishing savepoints ([GMBL81], [KSUW85]). Savepoints allow the transaction to decide which restart point is the best choice in a given situation. However, especially online transaction processing requires a more sophisticated interpretation of savepoints. Users often do not want the complete amount of "new" work to be rolled back to an earlier point of time but only "useless", "faulty", or "unpleasant" portions. Especially, they often want their actual working environment to be left intact if some of their work is to be rolled back. For example, let us consider a travel planning activity. Let us assume that we already did a reservation for a flight and a hotel and just have loaded the information about car rental companies. It happens that one of the car rental companies offers a special discount in case a certain airline is used. Unfortunately, we booked the wrong one and now want to roll back the reservation. In this case we just want the reservation for the flight to be rolled back but not the reservation of the hotel or the loading of the information about the car rental companies. Of course, in this simple case the rollback can easily be done by a compensating action. However, in more complex situations

compensation may be a bad idea because it is too complicated, for example, if too many compensating actions have to be performed till the desired state is recovered.

Another problem is, that the system has to clearly advise the user about the state of his application after a rollback was performed so that the user is exactly informed which of his work was undone. For example, if the user is editing a document he needs to know which of his corrections were undone. This requires the recovery component, among others, to be able to support the notion of context and context description.

As a supply to traditional recovery techniques (like before and after images ([Gray79])) compensating transactions (will) play a major role in the field of advanced transaction models. The intention of semantics-based concurrency control is to capture more application-specific semantics. Compensating (trans)actions were designed along the same lines. They are a flexible and powerful means to exploit the semantics of the application in order to minimize the effects of transaction failures. They especially relax the all-or-nothing principle since the compensation of an action must not necessarily result in an undo of *all* of the action's effects but may result in an application or even situation-specific repair action; for instance, a compensating transaction undoes, according to the *semantics* of the *application*, the effects of the transaction it compensates for. Quite a few advanced transaction models already rely on this technique, like *Sagas* ([GaSa87]), *ConTracts* ([WäRe91]), *multi-level* and *open nested transactions* ([WeSc91]), or a bunch of transaction models for multidatabase systems, like *Flexible transactions* ([ELLR90]), *Polytransactions* ([RuSh91]) or *S-transactions* ([VeEl91]).

Without doubt, compensating (trans)actions are a powerful, flexible, and indispensable recovery concept and, therefore, have to be integrated in our tool kit, too. Since they allow, for example, child transactions to commit and generally release their locks on data (no upward inheritance of locks) data can be released much earlier to the public than by the usual procedure applied within nested transactions. Especially, compensating transactions seem to be a mandatory feature for the relaxation of the nested (hierarchical) transaction model to more general models. On the other hand, compensating transactions introduce a new dimension of complexity. Especially, their inherent complex and hard to identify influence on other recovery techniques (than compensation), concurrency control measures, and transaction processing control

substantially impede a smooth integration of this technique into a more general environment like the tool kit.

Since in our project work on recovery aspects is still under way we have to refer the interested reader to forthcoming results [MeUn9x].

4.8 Example of a heterogeneously structured transaction tree

Figure 4.4 gives a simplified example of a heterogeneously structured transaction tree (especially, we focus on two characteristics of transaction types only, namely concurrency control scheme and task). It is assumed that each transaction provides full intra transaction parallelism (see section 4.4) and that each transaction runs a lock protocol on its object pool (no OCC or other concurrency control scheme is considered in this example).

As was explained in section 4.1 each children transaction can install its own type of lock protocol which regulates access to the data of the object pool of its father transaction (or, vice versa, the father transaction can restrict its child transactions to use a predefined type of lock protocol (or to use a lock protocol from a predefined *set* of lock protocols)).

In the example, the higher levels of the transaction hierarchy employ stricter types of lock protocols (since they run two-phase lock protocols) while on lower levels transactions partly join together to constitute a cooperative environment (since they run the non two-phase lock protocol as, for example, T17, T28, T30, T39, T40, and T41). This allows the installation of each kind of working environment.

For example, let us assume that the task of T1 is the design of a new vacation village. This task is subdivided into the design of the infrastructure (T3) and the design of the buildings (T4). The task of T4 can be further subdivided into the design of the administration buildings (T8) and the design of the vacation buildings (T9). Now let us assume, that the task of T17 is to design different types of (vacation) lodges and a central indoor swimming pool. The design of the indoor swimming pool is a relatively isolated task. Therefore it is delegated to a transaction which is strongly isolated from its environment (T29). The design of the different shapes of lodges needs much communication and exchange of data and is, therefore, performed within a cooperative environment (T17, T28, T30, ...).

In the Figure, the boxes indicate different tasks. Especially on higher level, transactions (for example, T1, T3) often only serve as service transactions while the leaf transactions, of course, are always of type operational.

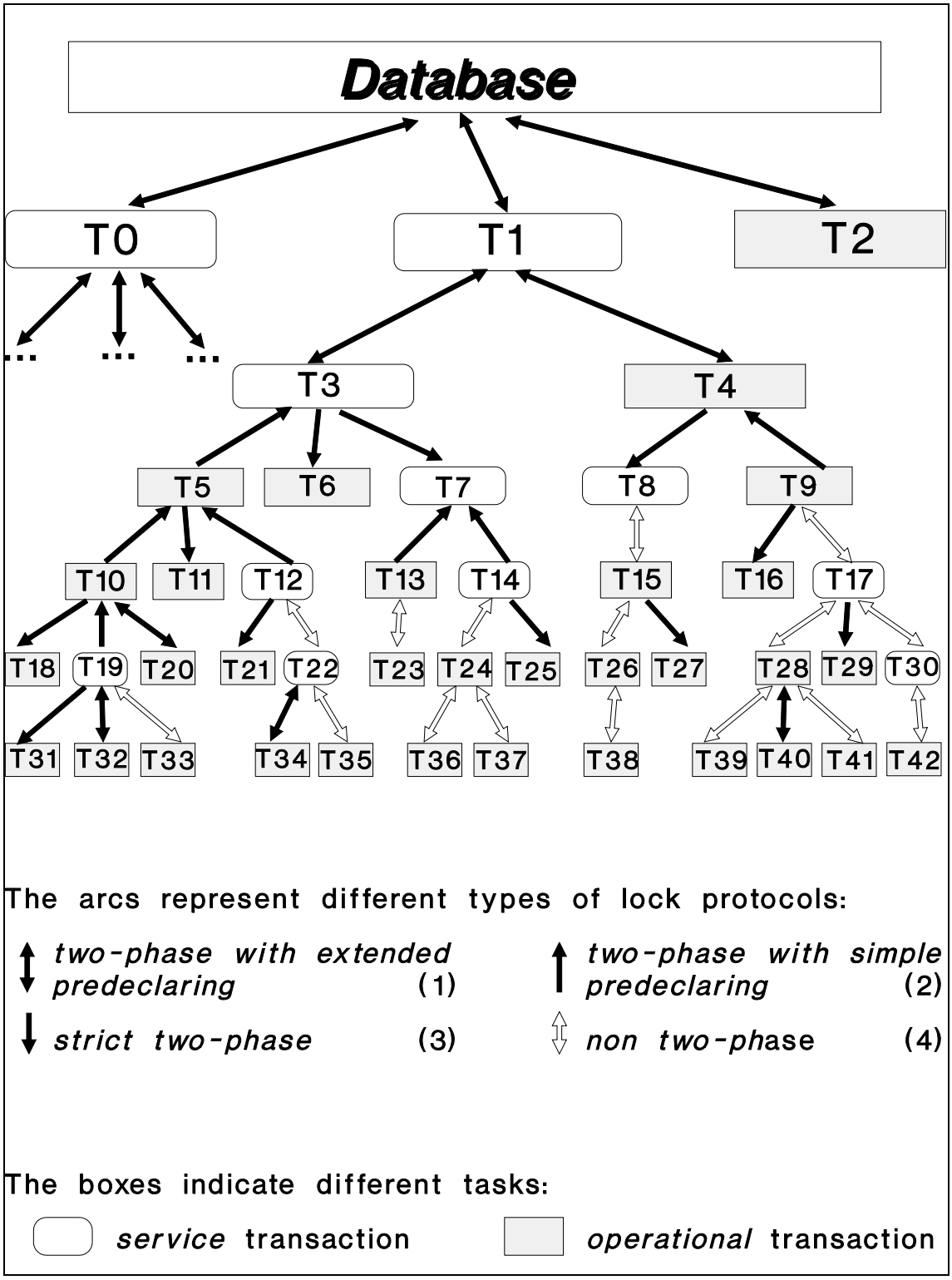


Fig. 4.4: *Heterogeneously* structured transaction hierarchy

5 Lock modes

5.1 Motivation of our approach

The data modeling facilities of most non-standard database systems allow the programmer to define complex objects and a rich set of operations on those objects. Since these operations are typically on a semantically higher level than *reads* and *writes* concurrency control mechanisms are needed which allow the programmer to exploit the inherent (application-specific) semantics of such operations. A common approach to the integration of application-specific semantics into concurrency control is type-specific or semantics-based concurrency control (c. f., [BaRa90], [BaKa91], [ChRR91], [HeWe88], [SkZd89], [SpSc84], [Weih88]). This kind of concurrency control aims at databases modelled in terms of objects; for instance, objects are treated as instances of abstract data types. An abstract data type is characterized by a set of specified operations which describe the only way in which a user is allowed to 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 non-serializable behavior in specific environments. With semantics-based concurrency control transactions can only invoke operations which are explicitly defined on the objects. Therefore, controlling the concurrent execution of the transactions involves the control of execution of the operations invoked on the objects. Whether 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 [ChRR91]). 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 ([SkZd89]). In other words, whatever the semantics of operations of application programs on objects is they are, as far as concurrency control is concerned, 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. Of course, this mapping entails that some of the semantics of operations may be

lost. 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 it decides compatibility on the level of database operations and not on the level of application operations and a write operation conflicts with another write operation on the same object.

Semantics-based concurrency control or **type-specific concurrency control** takes advantage of the operations semantics, which means that concurrency control is performed on the level of application operations. Since application operations are typically on a semantically higher level than reading and writing semantics-based concurrency control allows the definition of new weaker notions of conflicts among operations not possible with the information available only about objects and their types. 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 an obvious 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 the performance in a complex information system.

Note, however, that the applicability of commutativity and compatibility depends on the way in which concurrency control measures were implemented. If two update operations are allowed to run concurrently (or, as a weaker and more realistic criterion, in concurrent transactions) it has to be guaranteed that the modifications are performed on the same physical (not logical) piece of data. For example, assume two transactions running concurrently in a publication environment. One transaction applies a spelling checker to a document while another transactions inserts a new, but already checked, chapter into the same document. Compatibility allows both actions to be executed concurrently in different transactions. However, if each transaction works internally on a copy of the document (for example, created by a checkout operation) a

lost update will inevitably take place although, on a more abstract level, both operations are compatible.

Let us assume that semantics-based concurrency control can, in the average, increase concurrency. However, there are some drawbacks associated with this approach. The most important is that it eliminates the common and neutral basis of conventional concurrency control schemes. 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 type *Account* need to be considered to decide what kind of concurrency control measures are to be implemented (see Figure 5.1). In other words, conventional concurrency control provides a common and static basis (a number of lock modes) on which each operation of an application need to be mapped (see Figure 5.2).

In semantics-based concurrency control such a common basis is no longer existent. Instead, the compatibility of operations has to be defined on a semantically higher level, namely on the level

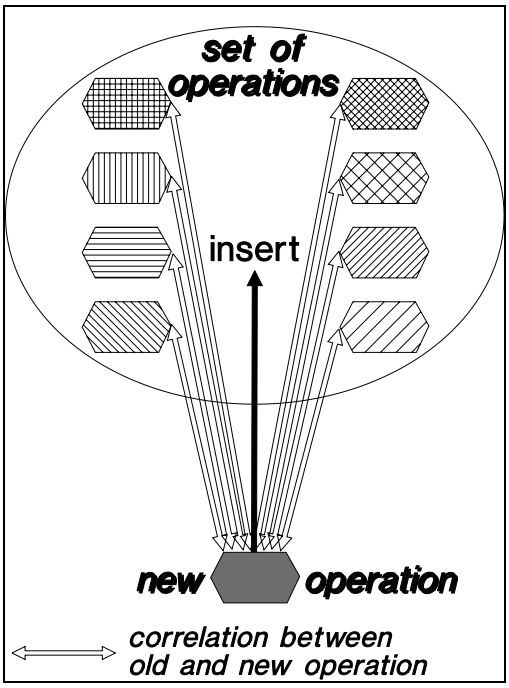


Fig. 5.1: *Semantics-based* Concurrency Control

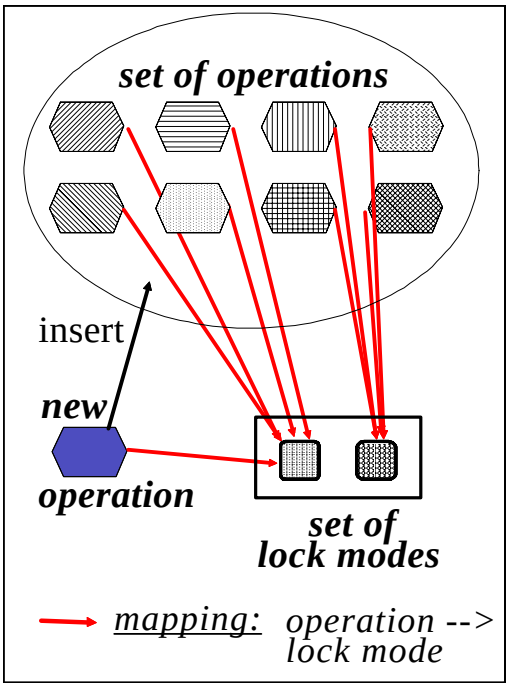


Fig. 5.2: Traditional Concurrency Control

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 of the given type. Therefore, an insertion of a new operation is highly sophisticated 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). Without any doubt, semantics-based concurrency control is useful, however, it is hard to design and implement. For the time being, semantics-based concurrency control may be applicable in special environments, but it is not sufficiently understood to be applied as a general approach to concurrency control.

The tool kit approach for concurrency control provides another solution for the inclusion of application-specific semantics into concurrency control (see also [Unla90]). It provides a number of elementary concepts from which the relevant can be chosen and put together to properly meet the demands of an application-specific concurrency control mechanism. For this reason the tool kit provides a rich 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. 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.

Barghouti and Kaiser ([BaKa91]) give a nice and comprehensive overview of work on concurrency control in advanced database applications.

5.2 Transaction related locks

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 5.1:

> 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 development. 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 in which the operation is supposed to be executed. In the next sections we will discuss in what way our approach can consider the various demands of different application areas. The first sections will concentrate on locks which can be linked temporarily to transactions (as in the conventional case) These locks will be called **transaction related locks** in the following. First, we will identify a set of useful lock

modes. Thereafter, it will be shown how these lock modes can be provided with proper effects on concurrent transactions.

However, the tool kit does not only provide locks which can be linked temporarily to transactions but also locks which can be temporarily bound to subjects (applications, user) (called **subject related locks**) or permanently to objects (called **object related locks**). The necessity of these concepts will be motivated later.

5.2.1 Basic lock modes of the tool kit approach

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 of 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.

5.2.2 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 database 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.

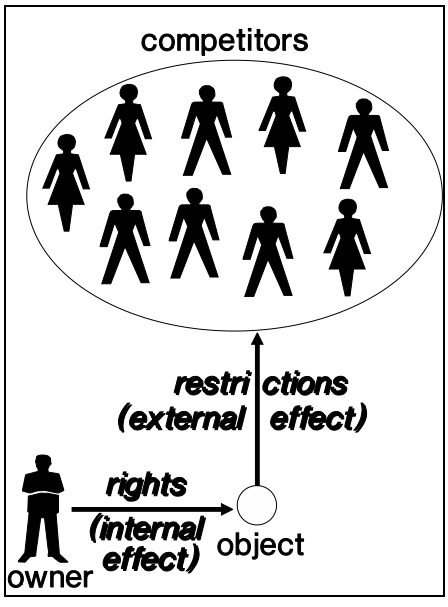


Fig. 5.3: *Two* effects of a lock mode

If we analyze the semantics of a lock, it becomes clear

that a lock on an object O has always two effects (see Figure 5.3):

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 5.2:

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.

internal effect \ *external effect*	**B**	**S**	**D**	**U**	**X**
B	👍	👍	👍	👍	👍
S	👍	👍	😐	😐	👎
D	👍	👍	😐	☝	👎
U	👍	😐	☝	☝	👎
X	👍	👎	👎	👎	👎

👍 ***permitted***
😐 ***possible***
👎 ***prohibited***
☝ ***should be used with care***

Tab. 5.1: Compatibility matrix *internal/external effects*

A general mechanism for the definition of lock modes should make it possible to individually combine a given internal effect with each 'meaningful' external effect. Table 5.1 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 permission of such a combination depends on the needs and abilities of the application. If the application is able to accept some responsibility for the consistent processing of data a 😐 may be replaced by a 👍 (for example, in cooperative environments). However, the sequence of 👍 in a row must be continuous; for example, if an S/U lock is permitted this implies that an S/D is permitted too. A ☝ corresponds to a 😐, but indicates that this combination should be used with care since an uncontrolled use will inevitably lead to inconsistent data. Since the rows for X and B do not contain a 😐, these internal effects can only be combined with one external effect (to form the X/B- and B/X-lock).

5.2.3 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 ac-

quires several B-locks for different objects there is no guarantee that the validity intervals of these objects do overlap.

Note, however, that the minimal demand on every operation on data is that the pure read or write process is realized as an atomic unit (short lock on page level).

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.

5.2.4 Upgrading a lock mode

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 5.1:

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 5.2:

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.

Of course, since the external effect defines what lock mode can be granted at most, a competitor can also acquire any weaker lock; for example, if an S/U lock is granted for an object then competitors may not only request a U/S lock but also weaker locks, for example, a D/S lock.

In the previous section it was said that a 👍 in a row of Table 5.1 indicates that the appropriate external effect can generally be permitted. An external effect, on the other hand, describes which operations can be granted to a competitor at best. In other words, an external effect of a lock describes the strongest internal effect of a lock which can be granted to a competitor. Nothing is said about the external effect which can be combined with the internal effect of a lock of an individual competitor. Unfortunately, it is no good idea to allow the competitor to combine a possible internal effect with any possible external effect as the following example shows.

Example 5.3:

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.

Example 5.3 shows, that a competitor cannot choose any external effect which can (theoretically) be combined with the given internal effect. Instead, competitors can only acquire locks with an external effect which are compatible to the internal effects of the already granted locks; for instance, the new lock must be compatible to already granted locks.

Definition 5.3:

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 5.4:

Let us assume that an S/U-lock is already granted to a user P. If a competitor C wants to acquire a D/S-lock, such a request can be granted since the external effect of P's lock permits the internal effect of C's lock (since D is weaker than U) and the external effect of C's lock does not prohibit the internal effect of P's lock (since both effects are the same). Since the internal effect D is weaker than the internal effect U and the external effects are the same the D/S lock is weaker than the U/S lock.

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 5.4:

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.

For the following discussion we will assume that each ☝ in Table 5.1 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 5.1), 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 (S is stronger than D since competitors are only allowed to read the object) but a weaker internal effect (S is weaker than D since D additionally permits the derivation of a new version). If an owner of an S/S lock wants to change the lock mode to a D/D lock (or vice versa) he needs to acquire the weakest stronger lock (D/S-lock).

5.2.5 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.

5.2.6 Dynamic assignment of an external effect

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**.

5.3 Transaction related 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 5.4. 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 child transactions 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:

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 5.4). 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 5.4, 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).

To summarize, in a nested environment a child transaction T^S may acquire an object O in lock mode X/Y from its parent transaction. On the level of the parent transaction O will be locked in mode X/Y. On the level of T^S this mode will be weakened to the mode X/*. T^S may replace * with any permissible external effect. Of course, if T31 wants to acquire an object O in mode U/B from transaction T5 (see Figure 5.5) the rule of stepwise transfer requires that the object/lock pair O (U/B) is taken over by each transaction on the path from T5 to T31.

A more detailed discussion of the semantics of lock modes and how they can be upgraded or downgraded is given in [Unla89].

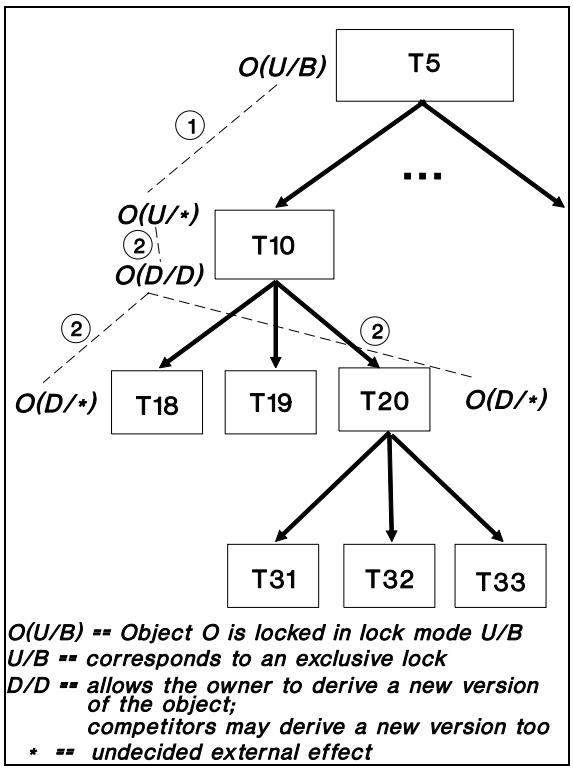


Fig. 5.4: Acquisition of an object from the *parent transaction*

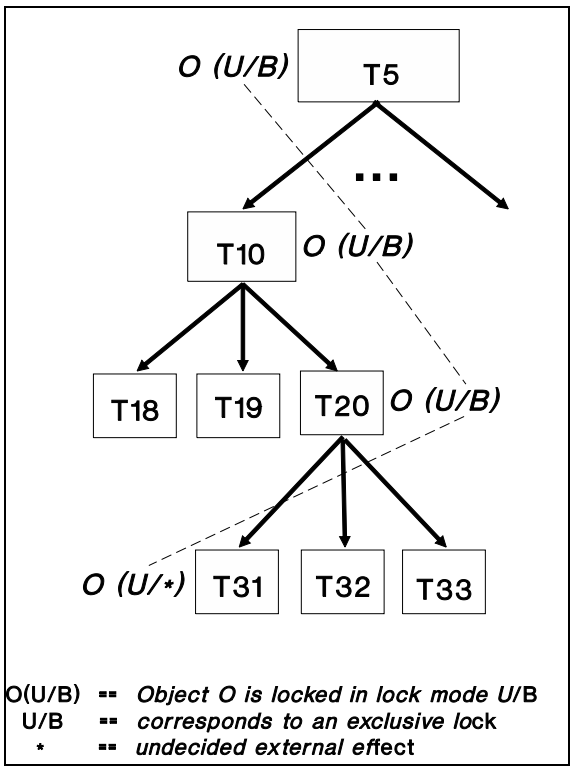


Fig. 5.5: Acquisition of an object from a *superior* (≠ parent)

5.4 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. 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 to them. However, if concurrency control relied on pure locks, it would be too inflexible (U/U).

Example: 5.5

Let us consider an abstract data type *PRICE-PRODUCT* on which the following four operations are defined (see Figure 5.6):

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 5.6) 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).

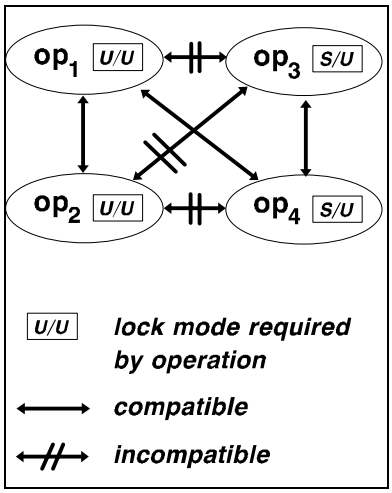


Fig. 5.6: Relationship between operations

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]).

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: 5.6

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 5.5

The scenario of Figure 5.6 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]).

5.5 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 5.7).

Example: 5.7

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*.

The common feature of these three examples is that they define restrictions on the access to data (for example, updating is prohibited). In principal, these restrictions can be ensured by three mechanisms:

1. *access control*
2. *integrity constraints*
3. *concurrency control*

Of course, none of the alternatives is exactly destined for the desired purpose. However, all of them could be extended in a way that they meet the requirements. We decided to assign the task to the concurrency control component. One reason for this decision is that it is not only elegant but also efficient since the concurrency control mechanism can exploit the additional knowledge about restrictions on the handling of data to increase efficiency and performance. A more detailed argumentation is given in [Unla89].

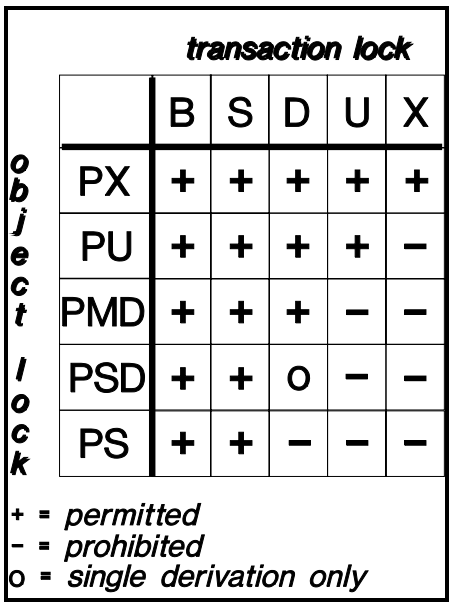
Our idea is to link locks permanently to objects. This kind of lock will be called **object-related lock** (**OR-lock** for short). An OR-lock, once imposed on an object, can neither be weakened nor released, it can only be upgraded. The lock remains valid as long as the object exists. OR-locks behave 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.



Tab. 5.2: Compatibility matrix object/transaction related locks

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 (if a modification or deletion of an object is prohibited every read is automatically a consistent read).

Table 5.2 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 5.7, 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.

In [Kelt87] prevention locks were introduced for similar reasons. But these locks were especially designed for CAD databases and versions.

In the appendix an example is presented which demonstrates the different facets of our approach.

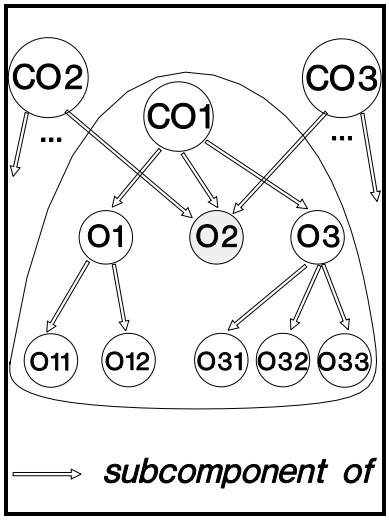


Fig. 5.7: Standard object O2 as subcomponent of several complex objects

5.6 Subject related lock

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.

Moreover, as already mentioned, a complex task often consist of several subtasks each of which works with a few common objects and a number of objects only specific for that subtask. If a task corresponds to a transaction the phase-specific objects need to be locked for an unnecessary long period of time, therefore, possibly blocking concurrent transactions. To avoid such unfavorable situations specific transaction types, like, for example, split-transactions were introduced ([PuKH88]). A split-transaction is a long-duration transaction which is split into two or more parts in order to be able to release some objects, which were requested by a concurrent transaction T^C earlier. Again, to safely transfer the objects to T^C we need some shelter which ensures that the released objects cannot be locked by another competitor before they can be acquired by T^C. To support a safe transfer of objects between transactions the tool kit provides 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 trans-

action/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 used in one transaction at a time. Only after the TR-lock on O was 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. However, the use of O in a transaction T substantially restricts O's further employment since it causes a dependency from locality. More specifically, the use of O in T1 corresponds to an explicit check-out of O from the database (or some object pool from the ancestor chain) by T1; for instance, O is added to all object pools on the path from the database to T1. Consequently, O can only be used in some other transaction T2 if O can be transferred from T1 to T2 according to the object transfer rule (see the chapter about explicit cooperations).

To differentiate a TR-lock from an SR-lock the latter will be written in italic letters in the following.

Example 5.8:

Let us assume that a user P has acquired the SR-lock *U/S* on O. Since the external effect of the lock is *S*, O can at most be read by concurrent transactions. If P wants to work on O in transaction T he (the transaction) can directly redeem the SR-lock by any TR-lock which is not stronger than U/S (for example, S/S or U/S). If P has acquired some lock on O in T, for example, a U/S-lock, he cannot acquire an incompatible lock mode on O, for example, again a U/S-lock, in any other transaction. This is prevented by the TR-lock which was acquired by T.

6 General rules of the tool kit approach

This section summarizes the "meta" rules which have to be obeyed by each transaction (type):

1. Dependency

Each transaction strictly depends on a unique parent transaction. The parent transaction can either be the database or another transaction. In the first case the transaction is either the top level transaction of a nested transaction or a flat transaction. In the second case the transaction is a child transaction of another transaction, the parent transaction. Dependency on a parent transaction entails the following effects:

- ☞ A child transaction must commit before its parent transaction commits.
- ☞ Commit of a child transaction and durability of its results are conditional subject to the fate of its superiors. Even if a child transaction commits, aborting of its superiors will undo its effects.
- ☞ A child transaction can only acquire objects from its parent transaction. If it needs an object from another superior the rule of stepwise transfer must be observed.
- ☞ A child transaction has to observe the relevant rules and constraints which were defined by its parent transaction with respect to the child transaction (according to the two-stage control-sphere).

2. Object acquisition and release

Each transaction can only acquire objects which belong to its access view. Moreover, it can only insert objects into an object pool of its release view. Only in case of an explicit cooperation may a transaction additionally acquire objects from or release objects into the object pool of another transaction. When a transaction T^C acquires an object from an ancestor T^A the object is duplicated and the copy is inserted into the object pool of T^C. If T^A is not the parent transaction of T^C the object/lock pair is, at least logically, inserted into the object pool of each transaction on the path from T^A to T^C (stepwise transfer). In any case T^C gets the object with an undecided external effect; for instance, T^C is allowed to grant any lock which is compatible with the acquired internal effect. All work of a transaction has to be performed on the objects of its own object pool. If a transaction T^C modifies an object and wants these modifications to become valid T^C has to insert the modified object into the object pool of a superior before EOT (end of transaction). By this, the object version of the superior is replaced by the "new" version of T^C. If the superior is not the parent transaction the object is removed from the object pools of all transactions on the path to the superior (of course, excluding the superior itself).

3. Concurrency control

If child transactions are allowed to concurrently acquire objects from the object pool of their parent transaction the parent transaction needs to run a concurrency control scheme for its object pool. For an operational transaction T to perform an operation on an object of its own object pool, T must observe the concurrency control scheme installed on its object pool.

4. Exclusion

If a transaction requests an object, the request can only be granted if the object is not locked in an incompatible mode (in general, if the appropriate concurrency control scheme agrees to such a request).

5. ***Transaction abort or commit***

   When a transaction fails (aborts), its objects are discarded. However, the object versions of its ancestors are left intact. In either case, when a transaction commits or aborts, its locks on the objects in the object-pool of the parent transaction are released.

6. ***Uniformity***

   All transactions follow these rules.

7 Constraints/rules/triggers

Constraints, rules, and triggers are rather powerful concepts to supervise, control, and organize the dynamic behavior (activities) of a computer system. Therefore, it is no surprise that these mechanisms also have a long tradition as integral parts of database systems (c. f., [ChCh82], [DaBM88], [DeZo81], [Eswa76], [KoDM88]). They are used to make database systems active (for example, [DaHL90]), to ensure integrity (for example, [KoDM88]), to control and supervise the execution of (sub)transactions (for example, [ChRa90], [ELLR90], [WäRe91]), or to support the modeling of cooperative environments (for example, [NoZd90]). The inherent power of these concepts and their general applicability are an excellent reason to also integrate such a mechanism into the tool kit. However, on the other hand, their integration into database systems (and not only into database systems) involves some tricky problems, especially with respect to consistency preservation (c. f., [StRH90], [StHP89]). The need to realize them on the basis of a sound formal foundation seems to be mandatory. Therefore, since our work on this subject is not yet finished, we just want to discuss briefly the intention we follow up with the integration of these mechanisms into the tool kit.

Our objectives are two-stage. First, we want to allow the programmer to define constraints on the use of the concepts of the tool kit and second we want to further improve one of the main goals of the tool kit - an extensive support of cooperative work. The management and supervision of control flow and execution order of (sub)transactions is, on the other hand, at least at the moment no hot topic within our project.

Constraints and rules as a means to specialize concepts

Since the intention of every tool kit must be, to provide (basic) components on the most general level we need mechanisms to restrict this generality if considerably more specialized, application-specific transaction managers are to be implemented. Some of such restriction mechanisms are already built-in. For example, the tool kit provides facilities to restrict the access view as well as the release view, to prohibit the use of operations, or to restrict the compatibility of lock modes (the last two facilities are briefly discussed within section 8). Other aspects, however, are not yet covered. For example, the tool kit permits the construction of arbitrarily structured (heterogeneous) transaction trees. However, such a liberalness is sometimes not wanted by an application. Especially in design environments groups of designers require a comprehensive support of cooperative support within their group, however, want their to be shielded from outside world. For this reason most advanced transaction models for design environments provide enclose a cooperative environment by more restrictive (ACID) transactions. To meet such demands we want to provide the programmer constraints which allow him to define how subtransactions of a given parent transaction type have to look like; for instance, we want to give the programmer the opportunity to predefine the structure of a transaction tree. In addition to such a static predefinition we want to provide rules by which a programmer can lay down under which circumstances which restrictions should come into effect.

Constraints and rules as a means to further support cooperative work

As was already mentioned before, cooperative work can only intensively be supported if the user or the application is involved in the preservation of the consistency of the database. This, however, means that the tool kit has to provide as much support to user as possible. To do so it has to be ensured that the degrees of freedom which are offered by the tool kit cannot only be used but also controlled. Among others, the tool kit allows lock modes to be compatible which are per se not compatible. For example, a U/S lock allows the owner of the lock to modify the locked object while other transactions are still allowed to read the object. Depending on the application it could essential that an access to the object by other transactions can be controlled; for instance, that other transactions are only allowed to access the object of certain conditions are fulfilled. For this reason we want permit that rules can be bound to locks. In section 5.2.6 we already discussed open locks. Open locks are locks whose external effect is

left undecided. Compatibility with other requests is each time individually be decided and may depend on the profile of the requesting process or the current state of the object. While one possibility is that the owner of locks makes the decision by himself the other possibility is that the decision is automatically made by the system on the basis of predefined rules. This frees the owner from being (frequently) disturbed in his work by other processes. In addition to the rule mechanism we want to offer a *notification mechanism* by which the owner or the requesting process can be informed about certain facts or can be asked to do something. The underlying rule mechanism is similar to ECA rules (event - condition - action rules, c. f., [DaHL90]). The basics of this concept were already presented in more detail in section 5.4.

8 Brief overview of the structure of the tool kit

The tool kit is meant to provide a wide range of different transaction types. This set needs to include conventional (short duration) transactions as well as different types of long-duration transactions. This requires that transaction types are made up of different components. For example, while a basic long-duration transaction type may hold its own object pool and lock table a more specific transaction type may, additionally, hold its own compatibility matrix for its object pool. By this, access to objects of the object pool can be individually adapted to the requirements of a specific environment. For example, in a more cooperative environment, the compatibility matrix may define lock modes to be compatible which should not be compatible in a more competitive environment.

In this section we will concisely outline how different transaction types can be constructed.

The tool kit can be regarded as a kind of **object-oriented transaction manager development facility** for the following reasons:

- ☞ Each component of the tool kit belongs to a **class**; each class represents a different type of component.
- ☞ Components are realized as **abstract data types**. This means in particular that transaction types are characterized by the operations which come with them. A number of op-

erations are common to all (long-duration) transaction types; for example, operations to suspend, continue and commit the transaction and to acquire or release objects. However, these operations may be implemented differently for different transaction types (with different semantics or with different implementation part (data structure)). Other operations are only specific to some transaction types since they come with the features by which these transaction types differ from other transaction types. Note, that the concept of abstract data type makes it possible to easily react to changing requirements. If, for example, in a workstation/server environment the maintenance of an object pool is to be moved from the server to the workstation this can easily be realized due to the locality of such changes. Moreover, this concept leaves some leeway for the implementation of logical structures; for example, while the logical structure of a transaction type may require a local object pool the actual implementation may lean on a global object pool (a common pool for all transactions).

☞ The assembly and refinement of transaction types is, among others, realized via (**multiple**) **inheritance**.

A transaction type is developed in the following way: First, the basic constituents are chosen from the set of **basic components** of the tool kit. These constituents are combined and suited to each other (either via multiple inheritance or via aggregation) to form a kind of **skeleton** of a first basic transaction type. For example, in Figure 8.1 the skeleton for LONG-DURATION TRANSACTION (2) is assembled from the basic components OBJECT POOL and TRANSACTION (1). At this basic level TRANSACTION is nothing more than a frame within which operations can be executed; for instance, no rules or constraints are laid down, no concurrency control or recovery mechanisms are established, etc. A skeleton transaction (or skeleton for short) can be specialized to more specific skeletons by adding more components and/or specializing (refining/extending) existing components.

In Figure 8.1, LONG-DURATION TRANSACTION is specialized to LONG-DURATION PESSIMISTIC TRANSACTION (transaction which runs a lock protocol; (4)) by adding the constituents COMPATIBILITY MATRIX and LOCK TABLE (3). A skeleton corresponds to a fundamental transaction type which is not yet executable since it does not obey the specific semantics of some underlying transaction model. For example, a skeleton may already provide a *check-out* operation. However, the rule that objects can only be acquired from an ancestor is not yet laid down; for instance, an instance T of LONG-DURATION PESSIMISTIC TRANSACTION can theoretically acquire an object from any other transaction and any other transaction can acquire on object from T (only theoretically since a skeleton transaction is defined to be not executable at all; see above). The rules and constraints of a specific transaction model need to be added in a subsequent phase (this is done within the model layer). Skeletons which are equipped with specific semantics (5) constitute **executable transaction types** (6). Of course, by adding further rules or constraints executable transaction types can be further specialized (7); for example, a transaction type which lays down that its instances need to run a simple two-phase lock

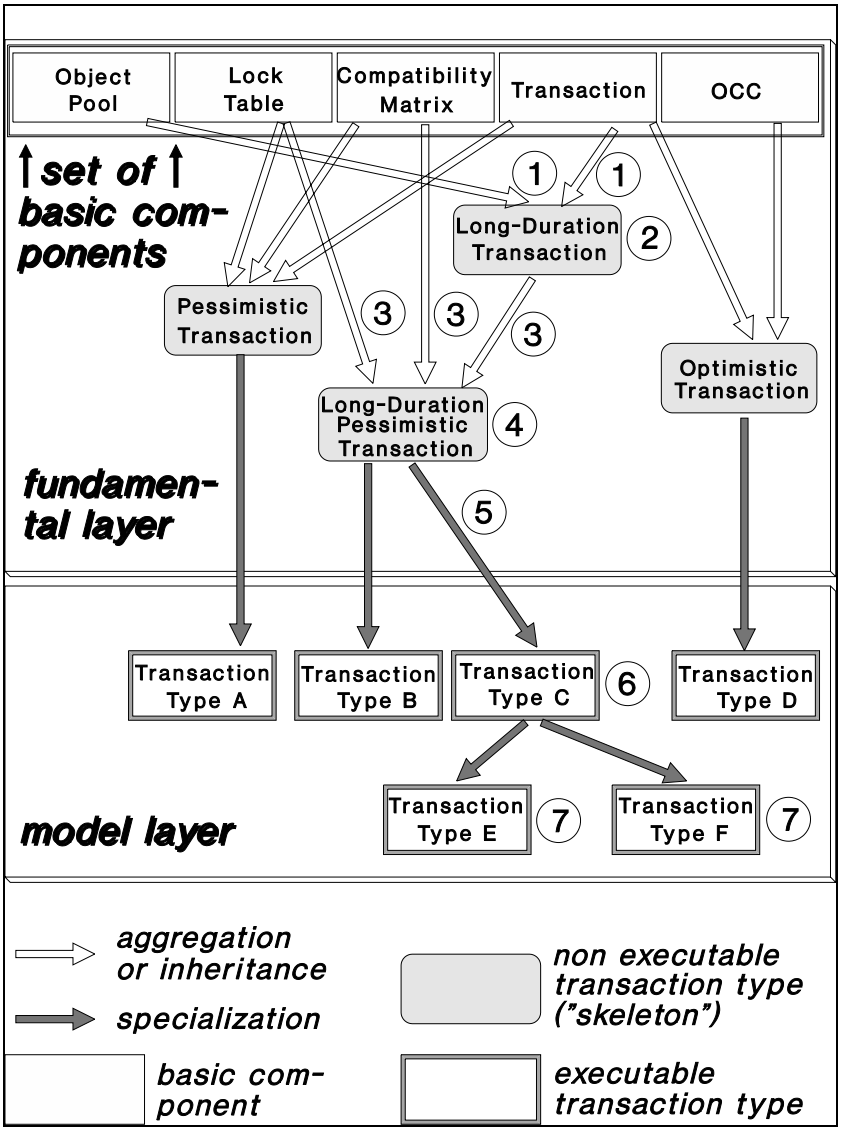


Fig. 8.1: *Structure* of the tool kit

protocol may be specialized to a transaction type which requires its instances to run the strict two-phase lock protocol with predeclaring.

Altogether the tool kit provides three sets, a set of basic components, a set of skeletons and a set of executable transaction types. Roughly speaking, the basic components are general objects which do not provide any specific semantics, skeletons are constructed by recursively putting things (basic components, skeletons) together while executable transaction types are defined by specializing operations and adding constraints and rules, etc. The basic components and the skeletons make up the **fundamental layer** of the tool kit while the executable transaction types constitute the **model layer**.

In Figure 8.2 the **create-transaction-class** command creates a new executable transaction type T^{Tnew} with name <**transaction-class-name**:> as subclass of an already existing transaction type specified in **subclass-of:**. T^{Tnew} inherits all properties of the superclass. These properties can be strengthened; for instance, a stronger lock protocol can be defined or some more operations can be prohibited in the **constraints** clause, etc. For example, if the inherited lock protocol relies on a simple growing phase these lock protocol can be increased to a lock protocol which requires preclaiming.

The **prohibit-operation** *clause* determines which operations are applicable to the object pool of T^{Tnew}. If the *all* option is chosen T^{Tnew} represents a service transaction since T cannot operate on its object pool. The other way round, if the no option is chosen or only some operations are defined as to be prohibited T^{Tnew} represents an operational transaction.

The **prohibit-lock-mode**: *clause* allows the programmer to specify whether the status of some internal/external effect pairs are to be changed from *permitted* to *prohibited*. Consider Table 5.1. A 😐 (and ☝) indicates that the compatibility of the respective internal/external effect pair is conditional subject to the fate of the respective transaction type. Each transaction type maintains its own compatibility matrix. In this matrix all 😐‘s and ☝‘s must have been replaced either by a 👍 or a 👎 (according to the rules for replacement, see section 5.2.2). T^{Tnew} inherits the compatibility matrix of its superclass, however, can strengthen this matrix by defining some further pairs to be incompatible.

The **prohibit-object-related-lock-mode**: (**prohibit-subject-related-lock-mode**:) clause serves the same purpose as the second, however, with regard to object (subject) related lock modes.

Besides the constraints mentioned above the constraints section allows the programmer to define other constraints, like constraints on the type of subtransactions (as discussed in section 8).

In the **operation:** *clause* new operations can be added or old operations can be replaced by new ones. For example, if we want to realize split transactions ([PuKH88]) we have to add the *join-, accept-join-, split-, and split-commit-transaction* operations.

Finally, in the **recovery-mechanisms:** clause the necessary recovery techniques can be chosen.

create-transaction-class

transaction-class-name:

transaction-class-id:

subclass-of:

lock-protocol: [
non-two-phase /
two-phase (*growing-phase*: [*simple* / *extended preclaiming* / *preclaiming*],
shrinking-phase: [*simple* / *strict*])]

constraints
...
prohibit-operation: [*no* / *all* / {[*read* / *update* / *derive-version* / *insert* / *delete*]}]
prohibit-lock-mode:
prohibit-object-related-lock-mode:
prohibit-subject-related-lock-mode:

operations:

recovery-mechanisms:

Fig. 8.2: Construction of an *executable* transaction type

Figure 8.2 only shows the principal way of how transaction types can be constructed. The original interface of the tool kit is window oriented. Therefore, each operation (like the **create-transaction-class** operation) provides its own command menu biased to the individual content of the window; for instance, options which cannot be chosen are not presented to the user.

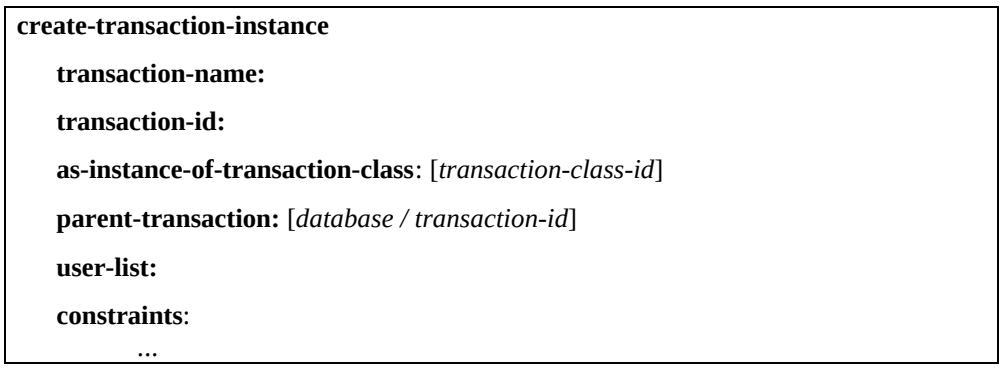


Fig. 8.3: Generation of a *real* transaction

While the **create-transaction-class** operation allows the programmer to define a new transaction type the **create-transaction-instance** operation (see Figure 8.3) creates a real transaction T^{new}. The type of T^{new} is specified in the **as-instance-of-transaction-class**: clause. The **parent-transaction:** clause specifies the parent transaction of T^{new} in a given transaction hierarchy. If the parent transaction is the database T^{new} is a top level transaction. The **user-list:** clause specifies the owner of the transaction while the **constraints:** clause allows the user to specify transaction-specific constraints (which are only valid for this transaction but not for other transactions of this type).

Finally, the transaction can be started by the **begin-transaction:** [*transaction-id*].

An **application-specific transaction manager** is defined by choosing the relevant (application-specific) transaction types from the set of executable transaction types. At best, all transaction types of interest are already existent and need only to be selected. In some cases, however, the tool kit may not provide all of them. In some other cases, a kind of general transaction type may already exist but need to be augmented by some special semantics. For example, if the tool-kit does not already provide *split-transactions* ([PuKH88]) the corresponding transaction types need to be defined (for example, by adding the corresponding operations) if this type of transaction is to be supported. Situations like this make it necessary to extend the tool kit by the missing transaction types. Dependent on the requirements an extension may affect several layers. In the simplest case an already existent executable transaction type need only to be supplemented by some additional semantics. In case the model layer does not provide an executable transaction type which can be used as the basis for a further specialization the missing type must be constructed from a skeleton. If the fundamental layer does not already

provide an appropriate skeleton it must be derived from another skeleton or it must be constructed from the basic set of components. In the worst case, the set of basic components has to be augmented (for example, by a new concurrency control scheme). Finally, if a different global transaction model is to be installed (for example, a model which allows a transaction "hierarchy" to be netlike instead of treelike) a new model layer must be developed; for instance, the transaction types of the model layer must be equipped with different semantics. Of course, such a task is rather ambitious and should, therefore, only be performed by a specialist.

9 A few comments on implementation issues

In a certain sense our approach is conceptually very similar to the database kernel system approach respective tool kit approach for database systems (see Figure 1.4) which especially means that the tool kit can be seen as to stay on the same level as the core data model of a database kernel system. The interface of a database kernel system provides a *core* data model only (or not even a data model but only the basis for the design a data model). Such an interface has to offer as many basic building blocks as necessary to put the DBI in a position to design and realize his application-specific data model. This implies that the core model should 'hard-wire' as few restrictions as possible to not endanger or even prevent the definition of adequate application-specific data models or, the other way round, each concept should be realized as general as possible at the core level. Application-specific data models are obtained by choosing the appropriate concepts of the core model and equipping them with specific semantics by *adding constraints and rules* (among others). This corresponds to a specialization. Applied to the tool kit this means that the "meta" transaction model has to lean on as few rules and constraints as possible. This is one of the reasons why the tool kit permits, for example, an arbitrary nesting of transactions of different types, whereas "full-function" transaction managers usually lay down some order. The tool kit provides a core set of general transaction types together with a small set of fundamental rules ("meta" transaction model) and it is left to the DBI or applications designer to enrich the fundamental set of rules and the core transaction types by further constraints and rules (inter- and intra-transaction constraints) to capture the specific semantics of a given application (see also the discussion in chapter 2.3).

As indicated in Figure 9.1 the tool kit can be regarded as to be logically subdivided into four parts, a tool kit for *transaction types*, for *concurrency control*, for *recovery*, and for *constraints, rules, and triggers* (as already mentioned, up to now, only the first two parts are entirely designed and implemented).

While the first prototype of the tool kit is simply implemented on top of a relational database management system we currently work on a *deep* integration of the tool kit into a non-standard database management system (NDBMS) (see Figure 9.1). All functionality of the interface of the tool kit will be mapped on lower level operations of the database system which especially means that all downward functionality (operations which are realized in terms of lower level operations) is hard-wired wherever useful. While this, of course, makes extensibility on lower levels of the system more complicated it, hopefully, makes our tool kit much more efficient.

Within the database system we want to rely on multi-level concurrency control ([Weik87]) which means that the transactions of our tool kit will internally be realized as multi-level transactions. **Multi-level** or **layered transactions** are a special case of nested transactions. Each transaction (tree) has the same nesting depth. The nodes of a given level of a transaction tree always correspond to operations at a particular level of abstraction in a layered system. The edges in a transaction tree express the fact that the operations of a particular level are realized by a sequence of operations at the next lower level. Since each level of a transaction tree represents its own level of abstraction each transaction on this level has to release its objects

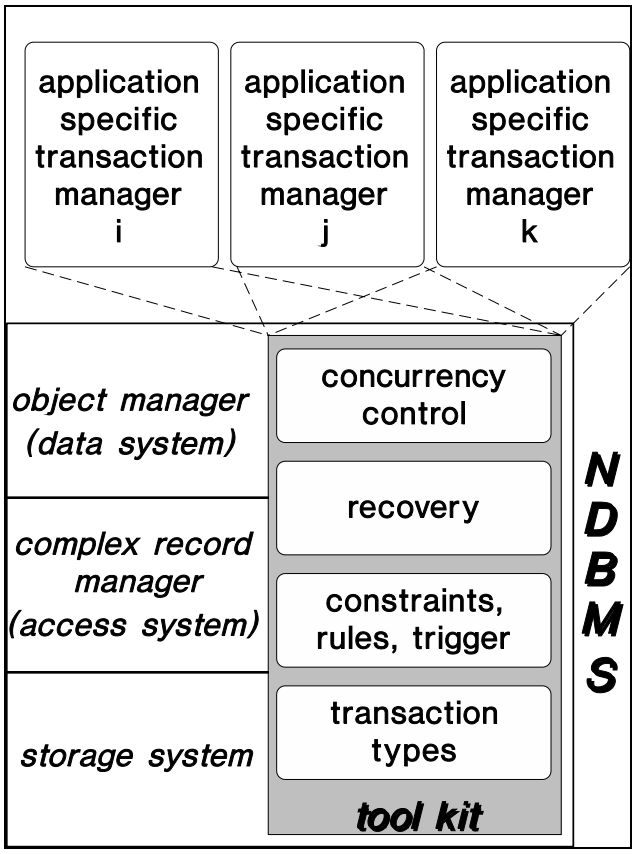


Fig. 9.1: Deep integration of the tool kit into the NDBMS

after commit. In case of a transaction failure at a higher level rollback has to be carried out by compensating transactions. The key ideas of multi-level concurrency control are to increase efficiency by releasing data much earlier (at least on lower levels of the system) and to make use of semantics-based concurrency control; for instance, to exploit the semantics of operations in level-specific conflict relations that reflect the commutativity or compatibility of operations ([WeSc91]).

Concurrency control for a layered architecture can be implemented on several levels. In case of the tool kit we are thinking about a quasi three level architecture where the lowest level is the page level, the medium level the complex record (storage object) level, and the top level the object level (*quasi* because the object level is a kind of logical level). On page and complex record level we want to implement a combination of semantics-based concurrency control and locking which means that we want to realize semantics-based concurrency control by means of the extended set of lock modes provided by the tool kit.

10 Overview of related work

Especially in the recent past a number of transaction models for advanced database applications were proposed in literature. As already mentioned, most of these approaches are either geared to special application areas or concentrate on the support of special conditions (like distribution of data) or are adapted to special types of database systems (like active database systems).

10.1 Special purpose transaction models

10.1.1 Design applications, especially CAD/CAM/VLSI

Design applications, especially CAD and VLSI applications, were among the first areas which strongly insisted on the development of more powerful transaction models. Therefore, most earlier proposals concentrate on the support of these application areas; c. f., [HaLo81], [LoPl83], [KLMP84], [KoKB85], [KSUW85], [HäRo87-1]. Basically, these approaches con-

centrate 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.

[HäRo87-1] discuss various issues of concurrency control for nested transactions. A model of nested transactions is introduced allowing for synchronous and asynchronous transaction invocation as well as single call and conversational interfaces. Their approach is among the first which enables not only parallelism among siblings but also among parent and children; for instance, they permit non-leaf transactions to perform operations on data. Their concept of *downward inheritance of locks* makes data manipulated by a parent available for its children. Moreover, the authors refined their original concept to *controlled downward inheritance* to enable a transaction to supervise and restrict the access mode to its inferiors for an object (see also section *4.3 Task*).

Since the above approaches concentrate on modularity, failure handling, and concurrent execution of subtasks (even though in different ways) we strongly assume that the current version of our tool kit is able to emulate (and implement) these approaches more or less directly. Appropriate investigations to substantiate this assertion are under way.

The **Cooperative Transaction Hierarchy** or **Transaction Groups** concept ([SkZd89], [NoZd90]) defines a nested framework for cooperating transactions in a design environment. Traditional approaches to transaction processing require conflicting operations to be executed in a strictly sequential way. The Transaction Groups model tries to overcome this problem by structuring 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.

Like the tool kit approach the model of cooperative transaction hierarchies places particular emphasis on the support of cooperative work. However, the tool kit approach provides substantially more flexibility in the composition of transaction hierarchies since it permits an arbitrary nesting of transaction types while the model of cooperative transaction hierarchies just provides one predefined, static hierarchy. On the other hand, the model of cooperative transaction hierarchies supports the specification of sequences of operations, a concept which is not volunteered by the tool kit approach. The low level lock modes are a subset of the lock modes provided by the tool kit while the concept of communication mode is covered by the rules and constraints concept.

10.1.2 Design applications, especially CASE

Recently, some papers were published which deal with **transaction models for software engineering**.

The **Split Transaction Model** ([PuKH88]) provides some facilities for *restructuring* in-progress transaction, such as the split-transaction and join-transaction operations. The split-transaction operation allows one on-going transaction to be split into two or more transactions as if they had always been independent, separating the data items accessed by the original

transaction among the new transactions in a serializable manner. The join-transaction operation allows two or more on-going transaction to be joined into one, combining the data items accessed by the originally separate transactions as if they had always been part of the same transactions, so that the changes are released together. In either case, the new operations are means to sustain the serializability criterion, even in environments like this which require some support of cooperative work. The main contribution of this approach is that it permits transactions to be split or joined. The tool kit does not provide such operations directly. However, the concept of subject related locks provides a basis on which these operations seem to be realizable. In fact, this model as well as its successor (see below) seem to be an excellent measure of the applicability of the tool kit since it requires standard transaction types to be equipped with further application-specific semantics. Again, investigations are under way.

[Kais90] extends the above work by adding and integrating the concept of programmer interaction. The novelty of this approach is the support of groups of individuals who are cooperating to achieve common goals in the context of design environments based on object management systems.

Cooperative transactions have been designed to support cooperation among the members of a group whilst groups must be isolated from other groups or individuals. As the author states, nearly all proposed cooperative transaction schemes have a particular shortcoming, namely they do not permit overlapping groups. For example, they usually do not allow a manager to see up-to-the-minute work of his developer groups. The proposed participant transaction model, however, introduces a new formalism, called participation domain, for relaxing the classical intent of serializability. It allows certain users to be designated as participants in a specific set of transactions. These transactions need not appear to have been performed in some serial order with respect to these participants; for instance, participants may view uncommitted updates. All other users, however, must observe a serial order.

The primary incentive behind participation domains as the basis for the **Participant Transaction Model** is that serialization conflicts can arise only across domains, never within domains, by definition. The intent is that most conflicts would normally arise between transactions associated with the same domain provided that domains are carefully designed to represent appropriate software design activities. The users controlling these transactions are cooperating on

the same group task and thus very likely need to share objects. Users working on unrelated tasks are assumed to very rarely share objects, and in these few cases the cost of creating parallel versions, with the later merging problem is acceptable. In this respect this approach is very close to our assumption that competitive and cooperative environments will coexist within a nested transaction (see Figure 1.3). Our concept of owner is similar to the concept of groups since it does not restrict the composition of a group. Therefore, overlapping groups can be established. The support of cooperation in the sense of the proposed participant transaction model exactly meets the intention of the tool kit approach and, therefore, must be realizable by the tool kit.

In a succeeding article ([KaPe91]) the authors report that in some situations their proposal is still unacceptable since it is symmetric. For example, it allows developers to inspect whatever the manager is doing. Therefore, the article presents a more general solution to this problem.

10.1.3 Other approaches

The **Nested Transaction Model with Predicates and Versions**, **NT/PV Model** for short ([KoSp88]), enhances the nested transaction model by explicit predicates and multiple versions. Each subtransaction has a precondition and a postcondition. The precondition describes the database state which is required for the transaction to execute correctly, while the postcondition describes the state of the database after the transaction commits. The concurrency control component ensures that interleaved executions of transactions do not violate any of these conditions. As is demonstrated in [KoSp90], explicit predicates are another concept to implement cooperating transactions.

With multiple versions a new version of a data item is created containing the new value each time the data item is updated. The old value is saved and can be exploited by the concurrency control component to allow greater concurrency. The tool kit offers several alternatives to exploit the existence of versions to increase concurrency and efficiency (object-related locks, special lock modes).

[LeMS91] propose the **Interactive Transaction (ITX) Model** to address requirements for transaction processing in a *multimedia telecommunication environment.* The authors propose a feedback mechanism to make the model interactive. This provides the ability to build applications as a set of cooperative tasks. They define an interactive transaction (ITX for short) as a feedback control process that interacts with the environment iteratively to satisfy (possibly user defined) objectives. Cooperative objectives are defined in terms of the observations on the objects shared by the cooperating ITXs. The system appears to be in a stable state to ITX when its observations remain unchanged. Such stable states are characterized by the correctness criterion of fixed point.

To achieve the cooperative objectives, multiple ITXs indirectly interact with each other by issuing transactions (TXs). TXs are atomic, but need not be serializable. An ITX remains active until some predefined termination condition is met. While an ITX is active, it monitors and reacts to the change in the states of the shared objects, possibly by repeated execution of its TXs. The execution of the TXs is controlled by the correctness criteria of the ITX. Both the termination condition and the correctness criteria are specified by the application logic and the users.

The contributions of the ITX model are:

- ☞ a high level feedback control framework supporting several correctness criteria
- ☞ a new correctness criterion for the definition, monitoring, and control of the distributed cooperative tasks
- ☞ an execution control criterion supporting active database features such as triggers, constraints, active views, and snapshot

The last approaches, like some succeeding proposals, rely on rules, constraints, or triggers. Since we have not finished work on this subject we will not compare these facets of the respective approaches with the abilities of the tool kit approach.

10.2 Transaction models for special types of database systems

Dayal, Hsu, and Ladin ([DaHL90], ([DaHL91]) propose a **generalized transaction model for long-running activities and active databases**. Their model is based on *different types of nested transactions*, namely concurrent subtransactions, deferred subtransactions, and decoupled (causally dependent and independent) transactions, and *triggers*. **Concurrent subtransactions** are subtransactions of a nested transaction in the original sense. If the execution of a subtransaction is explicitly delayed until the parent transaction has finished its task the subtransaction is called **deferred**. Like a regular subtransaction, the commit of a deferred subtransaction is conditional subject to the fate of its superiors. A **decoupled transaction** consists of a set of actions which were broken off from a more general transaction and executed within the decoupled transaction. Decoupling some actions permits transactions to finish more quickly, thereby releasing system resources earlier, and improving transaction response times. A decoupled transaction can execute concurrently with the transaction from which it was spawned. If it has to be ensured that the decoupled transaction must be serialized after the transaction from which it was spanned, the decoupled transaction is called to be **causally dependent**: it can commit only if the initiating transaction commits and only after the initiating transaction has committed. If no such dependencies exist the decoupled transaction is called **causally independent**. In contrast to regular concurrent subtransactions an abort of a decoupled transaction does not effect the initiating transaction (for example, the initiating transaction need not to be informed about the failure of the decoupled transaction).

The proposed transaction model is especially tailored to the needs of an active database system in which *event-condition-action (ECA) rules* are used as a general formalism for modelling the functionality (the activity) of the database. In such an environment a transaction T may execute an operation that signals event E for rule R, which has condition C and action A. If the E-C coupling mode is immediate, then C is evaluated as soon as E is detected (within the corresponding subtransaction). If the mode is deferred, then C is evaluated within the shelter of T, but after the last operation in T (again, within the corresponding subtransaction). If the mode is decoupled, then C is evaluated in a separate transaction. The same options are available for the C-A coupling. All in all, the model shows a more flexible way for expressing control flow for long-running activities in the context of an active database system. The use of the ECA rules allows the control flow to be dynamically modified based on the database state or the history of

events that have occurred. Exception handlers as well as compensating actions (see below) can be associated with each activity; these are invoked automatically by the system using a fixed policy as in the Saga model (see later) or they can be dynamically invoked by rules.

10.3 Transaction models based on compensating transactions

The use of compensating transaction provides much more flexibility with respect to the release of data. In this section we will discuss some approaches which substantially rely on compensation.

[GGKK91] introduce **Nested Sagas** as a means to allow for composition of long-running activities into Sagas. A **Saga** was defined as a collection of atomic transactions ([GaSa87]). By grouping transactions into a Saga, an application gains the ability to not only abort an individual transaction but a collection of transactions, namely a Saga. A rollback process of a Saga relies on the following two rules:

- Active transactions of the Saga are simply aborted and rolled back.
- Committed transactions of the Saga are compensated by application programs (compensating transactions) that were coded and supplied when the Saga was created.

One shortcoming of the original concept of Sagas is that it limits nesting to two levels. Nested Sagas remove this deficiency by generalizing the two-stage model to an arbitrarily nested model. Nested Sagas can recursively be defined as follows:

- A single, atomic transaction is a **primitive Saga**.
- A collection of Sagas is a **composite Saga**.
- A Saga is composed of an arbitrary number of primitive Sagas (transactions) and composite Sagas.

A Saga can be requested to abort at any time. Primitive Sagas are aborted by rolling back their effects, since they are atomic transactions. An abort of a composite Saga is applied to each of its component Sagas. Those that have committed are compensated for, while those that are

running are recursively aborted. The compensating actions used to abort a Saga must be specified when a Saga is defined.

As nested transactions nested Sagas support the decomposition of a long-running activity into a collection of related, simpler steps. The main achievement of this approach is its ability to abort or commit activities (subtransactions) independently by exploiting the concept of compensating transactions. Transactions, however, still have to observe the ACIDity property; for instance, a support of cooperative work is not within the scope of this approach.

[VeEl91] recommend the **S(emantic)-Transaction Model** which can be seen as an adaptation of the Saga model to a highly autonomous multidatabase environment; for instance, the S-transaction protocol supports the distributed and autonomous execution of long-lived, dynamically generated, tree structured transactions. The model was introduced to support cooperation of the international banking system. Since autonomy and confidentiality is a major concern of this approach, a component system issuing an S-transaction cannot dictate to other systems which further system should participate in the execution of the S-transaction. A component system is entitled to execute a subtransaction of an S-transaction in any way it wishes. In case of a failure, a component can issue an equivalent request to an alternative system. Consequently, the exact execution tree of an S-transaction cannot predefined. It may vary from one execution to another. For reasons of recovery, S-transactions use compensating transactions. Since subtransactions always commit when they complete their task the isolation of S-transactions is only supported at the level of subtransactions. Therefore, no concurrency control or commitment control is required on the level of S-transactions. The success or failure of a subtransaction depends on the success of failures of its children, and is described as an algebraic expression called a semantic constraint. S-transactions are meant to preserve both local and global consistency but, of course, only if the involved component systems can guarantee consistency.

[MRKN91] present a **transaction model for an open distributed publication environment**. The main properties of their model are that the system can guarantee (global) ACIDity under certain conditions, that a higher degree of concurrency can be achieved (compared to conventional transaction management), and that partial transaction undo is supported by aborting subtransaction (as is usual within nested transactions). The transaction model is based on open nested transactions ([BeBG89], [WeSc91]) which are a generalization of multi-level transac-

tions (see below). The main achievement of open nested transactions is that the changes of a subtransaction can already be made visible at the end of the subtransaction, however, not to every other transaction. In order to avoid inconsistent use of the results of committed subtransactions only those (sub)transactions which commute with the committed one are allowed to use the results. That is, the result of the execution of commuting transactions must be independent of their execution order. Early release of data requires the existence of inverse operations; for instance, the transaction model relies on compensating transactions and semantics-based concurrency control.

The concepts of semantics-based concurrency control and compensating transactions are closely related to each other. Semantics-based concurrency control allows the system to exploit commutativity or even compatibility of operations (for a definition, see section 5.1). Both commutativity as well as compatibility means that from the transactions' point of view the order in which compatible operations are performed is irrelevant. Consequently, the order in which two compatible operations op_1 and op_2 will be performed is accidental, either op_1 before op_2 or vice versa. And of course, such an open situation has great influence on recovery. Traditional techniques, like before-images, can no longer be applied since its use would mean that rollback propagation may occur (or, even worse, that durability can no longer be ensured). The only serious alternative is to combine semantics-based concurrency control and compensating transactions.

The current version of the tool kit neither supports semantics-based concurrency control nor compensating transactions. However, we believe that our concept of lock distinction in combination with the rule/constraints/trigger mechanism of the tool kit can serve as a solid basis for an integration and implementation of various facets of semantics-based concurrency control (see section 5.4).

10.4 Multi-level and open nested transactions

Multi-Level or **Layered Transactions** (c. f., [BeSW88], [Weik91]) and their generalization, the **Open-Nested Transactions** ([BeBG89], [WeSc91]) are a variant of nested transactions. Nested transactions are distinguishable from layered transactions first in that their internal

structure is explicit and provided as a user facility, and second in that their component transactions are not necessarily atomic. Multi-level transactions have an *implicit hierarchical internal structure* which is a result of transactions invoking operations on complex objects. Thus, the operations are decomposable into sub-operations. Both operations and sub-operations are considered atomic. That is the user sees a multi-level transaction as a set of atomic operations similar to a traditional transaction. The nesting is provided as a system facility and is not visible to the user. This means that layered transaction are first of all designed to provide a higher degree of concurrency (by finer grained lock granules) and efficiency (by exploiting operations' semantics on different levels of the system). As was already mentioned in section 9 the internal levels of the tool kit rely on multi-level transactions.

10.5 Multidatabase transaction models

Multidatabase transaction models extend the concept of nested transactions to be applicable in multidatabase environments. A **multidatabase system** provide facilities to access data stored in multiple *autonomous* and possibly *heterogeneous* database systems.

Polytransactions ([RuSh91]) concentrate on the maintenance of interdependent data in multidatabase environments. **Interdependent data** is defined to be two or more data items stored in different databases that are related to each other through an integrity constraint. The integrity constraint specifies the data dependency and the consistency requirements between the data items. For a given transaction the polytransaction model does not require all of the subtransaction dependencies and execution dependencies between them to be known in advance. Instead, it is assumed that there exists an interdatabase dependency schema (IDS) which contains a set of data dependency descriptors (D^3) that specify the interdatabase dependencies. The D^3s contain information indicating the conditions when the data dependency is to be considered violated, and the repair action to recover the data dependency. A polytransaction T^+ is the transitive closure of a transaction T with respect to the IDS. That is, when a transaction T is executed, the D^3s are checked to see if actions need to be taken to maintain the consistency of the database. The appropriate actions are taken, which may in turn cause other actions to be taken, and so on. In effect, the transaction T is considered the root transaction of polytransaction T^+ and the actions generated by T are considered to be the subtransactions of T. The ac-

tions generated by actions are considered to be the subtransactions of subtransactions. The main achievement of the polytransaction model is that it frees the programmer from having to worry about maintaining global data dependencies.

The main intention of the **Flex Transaction Model** ([ELLR90], [Leu91], [RELL90]) is to provide more flexibility in transaction processing, especially in multidatabase systems. As usual, they assume a long-duration transaction to be decomposable into a set of subtasks. The Flex Transaction model allows the user to specify for each subtask a set of functionally equivalent subtransactions, each of which when completed will accomplish the task. The execution of a Flex Transaction succeeds if all of its subtasks are accomplished; for instance, for each subtask one of the specified functionally equivalent subtransactions must commit. The approach is resilient to failures in the sense that it may proceed and commit even if some of its subtransactions fail. An important contribution of the model is that it allows to define dependencies - realized by predicates - on the subtransactions of a Flex Transaction, like failure-dependencies, success-dependencies, and external-dependencies. Failure-dependencies and success-dependencies define the execution order on the subtransactions while the external-dependencies define the dependencies of the subtransaction execution on events that do not belong to the transaction. Moreover, Flex transactions allow the user to control the isolation granularity of a transaction through the use of compensating transactions. Regarding to Figure 2.5 the approach allows the definition of many possible paths to achieve a (predefined) goal. In contrast to the tool kit this approach has placed his main emphasis on *predefining* possible paths.

The tool kit, in its current version, does not consider aspects of multidatabase environments. An extension of the tool kit in this direction is subject to future work.

10.6 Higher level approaches

ACTA ([ChRa90]) is a common transaction framework within which one can specify and reason about the nature of interactions between transactions in a particular model. It can be used to formally specify atomic transactions, Sagas, and a number of variations to the original model of Sagas, each variation coming out of changes to the formal characterization of Sagas. More specifically, ACTA allows the user to specify the effects of transactions on other transactions,

via the definition of *inter-transaction dependencies*, and the effects on objects, via the specification of the *view* of a transaction, the *conflict set* of a transaction and via the concept of *delegation*. The inter-transaction dependencies are similar to the ones which can be defined within the Flex Transaction model (commit, abort, termination, exclusion, begin, serial dependencies, etc.). The notion of **view** is similar to the notion of access view in the tool kit approach (it defines the state of objects visible to a transaction at a given moment). The **conflict set** of a transaction contains those in-progress *operations* with respect to which conflicts have to be determined; for instance, it allows the user to specify that conflicting operations in the usual sense are not be considered as conflicting in a given environment. The conflict relation allows ACTA to capture different types of semantics-based concurrency control, and to tailor them for cooperative environments. Therefore, the intention of conflict set is similar to the intention of the more relaxed interpretation of lock modes of the tool kit approach. Both approaches allow the user to implement different facets of semantics-based concurrency control (for a discussion of this topic with respect to the tool kit see above) and to install different kinds of cooperative environments. However, since the tool kit does not require the user to define compatibility on the level of operations it can be regarded as being a little bit more flexible. If required, it allows the user to define an operation to be compatible to another one in a given situation whilst being incompatible in another situation (by acquiring different lock modes, for example, U/U in the first case and U/S in the other). Finally, **delegation** allows a transaction to delegate the responsibility for committing or aborting an operation on an object to another transaction. This concept is similar to the concept of explicit cooperation (see section 4.5) of the tool kit approach. However, in the tool kit approach the granularity of delegation is the object and not the operation on the object. Moreover, in order to not undermine the constraints of the lock protocol 'on duty' the tool kit permits delegation only if special conditions are satisfied. To summarize, the ACTA framework allows the user to control and supervise (sub)transaction executions which is not an intention of the tool kit approach. With respect to the modeling of specific transaction managers both approaches have the same intention. However, we assume that the tool kit provides a superset of the facilities of the ACTA framework. A more precise statement to this assumption is subject of intensive studies which are currently performed at our department.

The **ConTract model** ([Reut89], [WäRe91]) concentrates on the definition and controlling of long-lived computations in non-standard applications. A **ConTract** is defined as a set of predefined actions (called **steps**) with an explicit specification of control flow among them. A step must not necessarily correspond to a (sub)transaction. It can also be a set of actions not performed under the shelter of a transaction. Therefore, this approach is on a semantically higher level than the other approaches presented in this section (and, of course, as the tool kit).

The main emphasis of the ConTract model is that the execution of a ConTract must be forward recoverable. This means when an execution of a ConTract is interrupted by failures, it must be re-instantiated and continued from where it was interrupted after the system has recovered. For this reason, all state information including database state, program variables of each step, context information, and the global state of the ConTract must be made recoverable. As Sagas, a ConTract is allowed to externalize partial results before the whole ConTract has committed. Compensating transactions are used to obliterate the results of committed steps which are identified as to be useless or integrity violating in retrospect. Similar to the NT/PV model, the ConTract model allows to define invariants (for example, preconditions, postconditions) on (data of) the database to support and relax concurrency control. As most other approaches which cover the aspect of supervising and controlling transaction executions the ConTract model allows the user to resolve conflicts in a more flexible way by specifying what to do when conflicts occur.

The main purpose of the tool kit is to provide concepts for the construction of transaction types and application-specific transaction managers. Control and supervision of (sub)transaction executions or predefining possible execution orders of (sub)transactions are not within the scope of the current version of the tool kit; for instance, the tool kit can be regarded as to offer services on a semantically lower level than, for example, the ConTract model. Nevertheless, to be in accord with its intention the tool kit has to provide some basic concepts which can be taken as a framework to realize higher level concepts. For this reason, the tool kit provides ingredients like *subject related locks* which, among others, can be used to realize a controlled transfer of objects from one transaction to another, or *constraints and rules*, which can be exploited by higher levels concepts, for example, to realize control flow. In fact, within a large, nationwide research program ('*Object bases for Experts*', sponsored by the

German Research Community (DFG)) we intend to integrate the tool kit approach and the ConTract model in order to bring together this two levels of abstraction.

10.7 Tabulated overview

Figure 10.1 summarizes the features of the most relevant models covered in this survey in terms of whether the respective transaction model

- provides *different* transaction types,
- allows an *arbitrary* nesting of transactions,
- supports *ACIDity* or *user-defined correctness criteria* (however, ACIDity must be considered as discussed in section 2.3),
- offers a *rule*, *constraint*, or *trigger mechanism*,
- supports *compensation* of actions, and
- allows the user to *control or supervise the execution* of (sub)transactions.

Model	*Transaction Types*	*Subtrans. Hierarchy*	*Correctness Criteria*	*Rules*	*Compen-sation*	*Control Flow*
Traditional	no	no	ACID	no	no	no
Nested [Moss81]	no	yes	ACID	no	no	no
CAD/CAM [KLMP84], [KoKB85] [KSUW85]	few	yes	ACID[2]	no	no	no
Cooperative trans. [NoZd90]	2	yes	user-defined	yes	no	limited[3]
Split [PuKH88]	no	no, dynamic restructuring	ACID	no	no	no
Participant trans. [Kais90]	2	yes	user-defined	no	no	no
Active DBS [DaHL90]	3	yes	ACID	yes	yes	no
Sagas [GaSa87]	no	no	ACID[1]	no	yes	no
Nested Sagas [GGKK91]	no	yes	ACID[1]	no	yes	no
S-transaction [VeEl91]	no	yes	ACID[1]	no	yes	no
Multi-Level [WeSc91]	no	yes, balanced, static level	ACID	no	yes	no
Open-Nested trans. [MRKN91]	no	yes	ACID[1]	no	yes	no
Polytransaction [RuSh91]	no	yes	user-defined	yes	no[4]	no
Flex transaction [ELLR90]	no	yes	user-defined	yes	yes	yes
ACTA [ChRa90]	numerous, can be constructed	yes	ACID - user-defined	yes	yes	yes
ConTract [WäRe91]	limited[1]	no	ACID[1]	yes	yes	yes
Tool Kit	numerous, can be constructed	yes,hete-rogeneous	ACID user-defined	not yet	not yet	no

1. depends on the involved component systems
2. [KSUW85] allow a restricted cooperation within group transactions (isolation is weakened)
3. by the concept of *patterns* (allows to specify operation sequences that must occur in a history
4. however, they provide *consistency restoration procedures* which restore consistency when the modification of some data makes other data inconsistent

Fig. 10.1: Characteristics of advanced transaction models

11 Concluding remarks

We have presented a flexible and adaptable tool kit approach for transaction management. The tool kit allows a sophisticated applications designer or database implementor to develop individual, application-specific transaction managers. 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. We have presented a basic set of characteristics 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 any 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.

3. With respect to concurrency control issues the tool kit provides concepts on two (orthogonal) levels. First, the tool kit provides a set of lock modes in the original sense. However, to be able to react to the needs of cooperative environments locks can be much finer grained. Each lock mode is described by an *internal effect/external effect pair*. This subdivision permits an exact adaptation of locks to the requirements of different application area; for instance, application-specific semantics can be exploited for reasons of concurrency control. Moreover, it was shown that the subdivision of lock modes can be exploited to comprehensively support cooperative work within a nested transaction. Second, in addition to 'conventional' transaction related locks *object related* and *subject related locks* were introduced. Object related locks are bound to objects and can be used to, for example, deal with different types of version models (time versions, version graphs) or library (standard)

objects. Subject related locks are bound to subjects (user, application, etc.) and can be used to, for example, supervise or direct the transfer of objects between transactions.

A first prototype of the tool kit was implemented in Smalltalk on top of the relational DBMS ORACLE and is currently in its test phase. It serves as a testbed for the examination of the weakness and strong points of our approach. We intend, as a second step, to integrate a revised version into the database kernel system OMS ([FrBo89]).

Further investigations

An urgent task is the definition of a formal notion of consistency in the context of advanced database applications. Further work has to be invested to integrate concepts which allow a child transaction to become even more independent of its parent transaction, for instance, to support the concept of open nested transactions ([WeSc91]). Moreover, the need that a transaction hierarchy has to form a tree might be too restrictive in same cases (although cooperation makes it possible to evade this principle). We want to support transaction graphs as well.

Another crucial point is the development of an appropriate intelligent interface for the tool kit. Even the current version of our tool kit is rather complex and requires deep knowledge about the characteristics and concepts of transaction management. Therefore, we want the tool kit to provide an intelligent interface which supports a DBI in his task to choose the right components and to combine them in an appropriate way.

At some later time we want to extend the tool kit in a way that it also permits the construction of transaction managers for homogeneous/heterogeneous distributed database systems (for example, multidatabase systems as described in [ElDe90] and [ELLR90]).

Acknowledgement

I wish to thank Prof. Dr. Gunter Schlageter for his support and his much (not many, because uncountable) constructive and helpful comments (not only on this topic), and Harm Knolle, and Erhard Welker for their helpful comments and valuable suggestions on a former version of this paper.

Literature

[BaKa91] Barghouti, N.; Kaiser, G.: *Concurrency Control in Advanced Database Applications*; ACM Computing Surveys; to appear 1991

[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

[BBGS88] Batory, D.; et. al.: *GENESIS: An Extensible Database Management System*; IEEE Transactions on Software Engineering; Vol. 14, No. 11; Nov. 1988

[BeBG89] Beeri, C.; Bernstein, P.; Goodman, N.: *A Model for Concurrency Control in Nested Transaction Systems*; Journal of the ACM; Vol. 36, No. 2; 1989

[BeHG87] Bernstein, P.; Hadzilacos, V.; Goodman, N.: *Concurrency Control and Recovery in Database Systems*; Addison-Wesley Publishing Company; 1987

[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]

[CaDe87] Carey, M.J.; DeWitt, D.J.: *An Overview of the EXODUS Project*; IEEE Database Engineering; Vol. 10, No. 2; June 1987

[ChCh82] Chang, J.-M.; Chang, S. K.: *Database Alerting Techniques for Office Activities Management*; IEEE Transactions on Communications; Vol. COM-30, No. 1; Jan. 1982

[ChRa90] Chrysanthis, P.; Ramamrithan, K.: *ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior*; Proc. ACM-SIGMOD Int. Conf. on Management of Data; Atlantic City, NJ; May 1990

[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

[DaHL91] Dayal, U.; Hsu, M.; Ladin, R.: *A Generalized Transaction Model for Long-Running Activities and Active Databases* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

[DaSm86] Dayal, U.; Smith, J: *PROBE: A Knowledge-Oriented Database Management System*; in: 'On Knowledge Base Management Systems';, M. Brodie, J. Mylopoulos (Editors);, Springer Publishing Company; 1986

[Davi73] Davies, C.: *Recovery Semantics for a DB/DC System*; Proc. ACM National Conference 28; 1973

[DeZo81] De Antonellis, V.; Zonta, B.: *Supporting Semantic Rules by a Generalized Event/Trigger Mechanism*;Proc. 6th Int. Conf. on Very Large Data Bases (VLDB); 1981

[DKAB86] Dadam, P.; Kuespert, K.; Andersen, F.; Blanken, H.; Erbe, R.; Guenauer, J.; Lum, V.; Pistor, P.; Walch, G.: *A DBMS Prototype to Support NF^2-Relations: An Integrated View on Flat Tables and Hierarchies*; Proc. ACM-SIGMOD Int. Conf. on Management of Data; Washington, D. C.; 1986

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

[ElDe90] Elmagarmid, A.; Du W.: *A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems*; Proc. IEEE 6th Int. Conf. on Data Engineering; Los Angeles, California; Feb. 1990

[ELLR90] Elmagarmid, A.; Leu Y.; Litwin, W.; Rusinkiewicz, M.: *A Multidatabase Transaction Model for InterBase*; Proc. 15th Int. Conf. on Very Large Data Bases (VLDB); Brisbane, Australia; Aug. 1990

[Eswa76] Eswaran, K. P.: *Specifications, Implementations, and Interactions of a Trigger Subsystem in an Integrated Data Base System*; IBM Research Report RJ1820; Aug. 1976

[FrBo89] Freitag, J.; Bode, T.: *A General Manager for Storage Objects as the Basis for the Implementation of Complex Objects in an Object Management System*; (in German); Proc. GI Conference on Database Systems for Office Automation, Engineering, and Scientific Applications; Zurich, Switzerland; March 1989

[GaSa87] Garcia-Molina, H.; Salem, K.: *Sagas*; Proc. ACM-SIGMOD Int. Conf. on Management of Data; San Francisco, California; 1987

[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

[GGKK91] Garcia-Molina, H.; Gawlick, D.; Klein, J.; Kleissner, K.; Salem, K.: *Modeling Long-Running Activities as Nested Sagas* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

[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

[GMBL81] Gray, J.; McJones, P.; Blasgen, M.; Lindsay, B.; Lorie, R.; Price, T.; Putzolu, F.; Traiger, I.: *The Recovery Manager of the System R Database Manager;* ACM Computing Surveys; Vol. 13, No. 2; 1981

[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

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

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

[HäRo87-1] Härder, Th.; Rothermel, K.: *Concurrency Control Issues in Nested Transactions*; IBM Almaden Research Report RJ5803, San Jose; Aug. 1987

[HäRo87-2] Härder, Th.; Rothermel, K.: *Concepts for Transaction Recovery in Nested Transactions*; Proc. ACM-SIGMOD Int. Conf. on Management of Data; San Francisco, California; 1987

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

[HMMS87] Härder, T.; Meyer-Wegener, K.; Mitschang, B.; Sikeler, A.: *PRIMA - a DBMS Prototype Supporting Engineering Applications*; Proc. 12th Int. Conf. on Very Large Data Bases (VLDB); Brighton, England; 1987

[JaJR88] Jarke, M.; Jeusfeld, M.; Rose, T.: *A Global KBMS for Database Software Evolution: Documentation of first ConceptBase Prototype*; Research Report MIP-8819; University of Passau, O. Box 2540, D-8390 Passau, Germany; 1988

[Kais90] Kaiser, G.: *A Flexible Transaction Model for Software Engineering*; Proc. IEEE 6th Int. Conf. on Data Engineering; Los Angeles, California; Feb. 1990

[KaPe91] Kaiser, G.; Perry, D.: *Making Progress in Cooperative Transaction Models* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

[Katz84] Katz, R.H.: *Transaction Management in the Design Environment*; in: 'New Applications of Data Bases'; G. Gardarin, E. Gelenbe (Editors); Academic Press; 1984

[KaWe83] Katz, R. H.; Weiss, S.: "*Transaction Management for Design Databases*"; Computer Sciences Technical Report #496, University of Wisconsin Madison; February 1983

[Kelt87] Kelter, U.: *Concurrency Control for Design Objects with Versions in CAD Databases*; Information Systems; Pergamon Journals Ltd.; Vol. 12, No. 2; 1987

[Kelt88] Kelter, U.: *Concepts for Transactions in Non Standard Database Systems*; (in German); Informationstechnik it; R. Oldenbourg Verlag; Vol. 30, No. 1; 1988

[Kim91] Kim, W.: *Object-Oriented Database Systems: Strengths and Weaknesses*; Journal of Object-Oriented Programming; SIGS Publications, New York; Vol. 4, No. 4; July/August 1991

[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

[KoDM88] Kotz, A.; Dittrich, K.; Mueller, J.: *Supporting Semantic Rules by a Generalized Event/Trigger Mechanism*; Proc. Conf on Extending Data Base Technology (EDBT); Venice, Italy; 1988

[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

[KoKB88] Korth, H.F.; Kim, W.; Bancilhon, F.: O*n Long-duration CAD Transactions*; Information Sciences 46, pp 73 - 107; 1988

[KoSp88] Korth, H.F.; Speegle, G.: *Formal model of correctness without serializability*; Proc. ACM-SIGMOD Int. Conf. on Management of Data; Chicago, Illinois; June 1988

[KoSp90] Korth, H.F.; Speegle, G.: *Long-Duration Transactions in Software Design Projects;* Proc. IEEE 6th Int. Conf. on Data Engineering; Los Angeles, California; Feb. 1990

[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

[LeMS91] Lee, M.; Mansfield, W.; Sheth, A.: *An Interactive Transaction Model for Distributed Cooperative Tasks* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

[Leu91] Leu, Y.: *Composing Multidatabase Applications using Flexible Transctions* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

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

[MeUn9x] Meckenstock, A.; Unland, R.: *Recovery Concepts for Non-Standard Database Systems*; Research-Report (in preparation); University of Münster; 199x

[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

[MRKN91] Muth, P.; Rakow, T.; Klas, W.; Neuhold, E.: *A Transaction Model for an Open Publication Environment* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

[MSOP86] Maier, D.; Stein, J; Otis, A.; Purdy, A.: *Development of an Object-Oriented DBMS*; Proc. ACM 1st Int. Conf. on Object-Oriented Programming Languages, Systems and Applications (OOPSLA); Portland, Oregon; Sept. 1986

[NeSt89] Neuhold, E.; Stonebraker, M.: *Future Directions in DBMS Research*; ACM SIGMOD Record; Vol. 18: No. 2; July 1989

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

[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

[PSSW87] Paul, H.-B.; Schek, H.J.; Scholl, M; Weikum, G.: *Architecture and Implementation of the Darmstadt Database Kernel System*; Proc. ACM-SIGMOD Int. Conf. on Management of Data; San Francisco, California; 1987

[PuKH88] Pu, C. ; Kaiser, G.; Hutchinson, N.: *Split-Transactions for Open-Ended Activities*; Proc. 14th Int. Conf. on Very Large Data Bases (VLDB); Los Angeles, CA; Aug. 1988

[Reed78] Reed, D.: *Naming and Synchronization in a Decentralized Computer System*; Ph.D. Thesis; M.I.T. Dept. of Elec. Eng. and Comp. Sci.; Technical Report 205; Sept. 1978

[Reut89] Reuter, A.: *ConTracts: A Means for Extending Control Beyond Transaction Boundaries*; Proc. 2nd Int. Workshop on High Performance Transaction Systems; Asilomar; Sept. 1989

[ReWä91] Reuter, A.; Wächter, H.: *The ConTract Model* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

[RELL90] Rusinkiewicz, M.; Elmagarmid, A.; Leu Y.; Litwin, W.: *Extending the Transaction Model to Capture more Meaning*; ACM SIGMOD Record; Vol. 19, No. 1; March 1990

[RuSh91] Rusinkiewicz, M.; Sheth, A.: *Polytransactions for Managing Interdependent Data* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

[SCFL86] Schwarz, P.; Chang, W.; Freytag, J.; Lohman, G.; McPherson, J.; Mohan, C.; Pirahesh, H.: *Extensibility in the Starburst Database System*; Proc. of the ACM-IEEE International Workshop on Object-Oriented Database Systems; Pacific Grove, California; IEEE Computer Society Press No. 734; 1986

[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

[StHP89] Stonebraker, M.; Hearst, M.; Potamianos, S.: *A Commentary on the POSTGRES Rules System*; SIGMOD Record, Vol. 18, Nr. 3; September 1989

[StRH90] Stonebraker, M.: Rowe, LA.; Hirohama, M.: *The Implementation of POSTGRES*; IEEE Transactions on Data and Knowledege Engineering; Vol. 2, No. 1; March 1990

[StRo87] Stonebraker, M.; Rowe, L:A.: *The POSTGRES Papers*: (*The Design of POSTGRES; The POSTGRES Data Model; A Rule Manager for Relational Database Systems; The Design of the POSTGRES Storage System; A Shared Object Hierarchy*); Electronics Research Laboratory; College of Engineering, University of California Berkeley, Memoran-

dum No. UCB/ERL M86/85; June 1987; additionally Proc. ACM-SIGMOD; Washington, D.C.; May 1986; Proc. ACM SIGMOD; San Francisco, California; May 1987, Proc. 13th Int. Conf. on Very Large Data Bases (VLDB); Brighton, England; 1987; Proc. IEEE 3th Int. Conf. on Data Engineering; Los Angeles, California; Feb. 1987; Proc. ACM-IEEE Int. Workshop on Object-Oriented Database Systems; Pacific Grove, California; Sept. 1986

[Unla89] Unland, R.: *A General Model for Locking in Non-Standard Database Systems* (in German); Research-Report; University of Hagen, Department of Computer Science; 1988; and (extended abstract) Proc. GI 3rd Conf. on Database Systems for Office Automation, Engineering, and Scientific Applications; Zurich, Switzerland; 1989

[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

[UnPS86] Unland, R.; Prädel, U., Schlageter, G.: *Redesign of optimistic methods: improving performance and applicability*; Proc. IEEE 2nd Int. Conf. on Data Engineering; Los Angeles, California; Feb. 1986

[UnSc89-1] Unland, R.; Schlageter, G.: *A Multi-Level Transaction Model for Engineering Applications*; Proc. 1st Int. Symposium on Database Systems for Advanced Applications; Seoul, South-Korea; April 1989

[UnSc89-2] Unland, R.; Schlageter, G.: *An Object-Oriented Programming Environment for Advanced Database Applications*; Journal of Object-Oriented Programming; SIGS Publications, New York; Vol. 2, No. 3; May/June 1989

[UnSc91] Unland, R.; Schlageter, G.: *A Flexible and Adaptable Tool Kit Approach for Transaction Management in Non Standard Database Systems* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

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

[VeEl91] Veijalainen, J.; Eliassen, F.: *The S-transaction Model* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

[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

[Weik87] Weikum, G.: *Enhancing Concurrency in Layered Systems*; 2nd Int. Workshop on High Performance Transaction Systems; Pacific Grove, California; Lecture Notes in Computer Science 359; D. Gawlick, M. Haynie, A. Reuter (Editors); Springer Publishing Company; 1987

[Weik91] Weikum, G.: *Principles and Realization Strategies of Multilevel Transaction Management*; ACM Transactions on Database Systems (TODS); Vol. 16, No. 1; March 1991

[WeSc91] Weikum, G.; Schek, H.-J.: *Multi-Level Transactions and Open Nested Transactions* (extended abstract); IEEE Data Engineering Bulletin; Vol. 14, No. 1; Special Issue on "Unconventional Transaction Management"; March 1991

[WoKi87] Woelk, D.; Kim, W.: *Multimedia Information Management in an Object-Oriented Database System*; Proc. 12th Int. Conf. on Very Large Data Bases (VLDB); Brighton, England; Aug. 1987.

Appendix: An example

As already mentioned, the tool kit is designed to offer a basis which allows a sophisticated applications designer or DBI to model his application layer in an appropriate and natural way. To do so he must be in a position to consider operations' semantics in his design. In this section we will demonstrate such an approach by modeling some operations of a simplified software development application. Since we are only interested in concurrency control aspects we will only describe the lock-specific part of the operations in more detail.

Let us assume that our application environment consists of a library which includes standard procedures as well as individual software modules. Standard procedures can be included "as is" in software modules. Software modules consist of standard procedures as well as individual code. A number of software modules can be combined in a software package which, in fact, corresponds to a product, for example, a word processing system.

Standard procedures

For standard procedures the following rules are valid:

1. Since standard procedures can only be used as is, a programmer is not allowed to modify them.
2. If a standard procedures is to be modified this is only possible by deriving a new version of it.
3. Standard procedures may be implemented in different programming languages. Therefore, different representations of a procedure may exist.
4. Since software environments may slightly differ, variants of a standard procedure can be defined which, for example, differ in the type of their parameters.

2., 3. and 4. mean that a standard procedure is represented by a version graph which may contain different representations as well as different variants.

The following (informal) operations are defined on standard procedures:

a. *browse_standproc (...)*: Read the complete version graph of a standard procedure (in a dirty mode). This operation needs to acquire a B/X-lock (browse lock).

b. *read_standproc (...)*: Read a specific representation (variant) of a procedure. This operation needs to acquire an S/D (derivation possible) or S/S-lock (derivation not possible).

c. *insert_standproc (...)*: Insert a new standard procedure. This operation needs to impose an (object related) PMD-lock (exclusive derivation) on the object which is to be inserted.

d. *derive_standproc (...)*: Derive a new version (representation, variant) of a standard procedure. This operation needs to acquire a D/S lock for exclusive derivation or a D/D lock for shared derivation. Moreover, it need to establish a PMD-lock on the new version of the object.

The concurrency control component will react as follows:

Every standard procedure is protected by a PMD lock. A PMD lock is only compatible with a B/*, S/* and D/* lock (* can be replaced by any possible external effect). Other locks are prohibited. Therefore, an S/D lock does not have to be considered by the concurrency control component since the PMD lock already excludes every incompatible lock mode. A B/X lock either has not to be considered since it is no lock in the literal sense of the word. Only an S/S, a D/S, and a D/D lock require the concurrency control mechanism to consider them.

Scenario 1: Software packages cannot be modified

If software packages cannot be modified every modification defines a new version of the package. All versions are arranged in a linear order, since at any time only one (unique) version of the software package can be valid, for instance, a time version mechanism needs to be implemented.

In the following we only want to take a closer look at such operations which demonstrate new aspects with regard to concurrency control.

Since at any time only one new version of a software package can be derived a PSD lock need to be imposed on each package (by every operation which inserts a new version of a package or a completely new package). A PSD lock only permits the acquisition of a B/*, S/*, or D/S lock from which only an S/S and a D/S lock need to be considered by the concurrency control component. This reduces the overhead for concurrency control substantially.

Let us consider two exemplary operations, a check-out_softpack (...) operation which checks out a special software package from the database and a *check-in_softpack (...)* which inserts a new version. The *check-out_softpack* operation needs to request a D/S lock (concurrent transactions are not allowed to modify the package or to derive a new version of it but they are allowed to read the package in S/D mode). The *check-in_softpack* operation has to establish a PSD lock on the new version. Moreover, it may release the D/S lock on the old version. Note, however, that the concurrency control component may delay the release of the lock, for example, in case the strict two-phase lock protocol is realized till the end of the transaction.

Scenario 2: Software packages can be modified

If we assume that software packages can be updated object related locks need not to be established. In this case we need different check-out operations, one for a check-out to derive a new version and one for a check-out to update. The first check-out operation is similar to the check-out operation of scenario 1. The *check-out_softpack_for_update* has to acquire a U/S lock for the software package which is to be updated. The U/S lock still allows concurrent transactions to read the package (in S/U mode). Now let us assume that a package which is to be updated contains some standard procedures P^S. Since usually these standard procedures will also be locked in U/S mode other transactions are excluded from locking some other software package P which also contains at least one P^S. This reduces concurrency unnecessarily. Since our tool kit supports object related locks, such an unfortunate situation can be avoided since the PMD lock preserves each P^S from being locked in U/S mode. Instead, the concurrency control mechanism needs not to consider any lock on a P^S at all.

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.

Nr. 32 Meckenstock, A., Unland, R., Zimmer, D.: Controlling Cooperation and Recovery in Nested Transactions, März 1994.

Nr. 33 Kurbel, K., Schnieder, T.: Integration Issues of Information Engineering Based I-CASE Tools, September 1994.

Nr. 34 Unland, R.: *TOPAZ:* A Tool Kit for the Assembly of Transaction Managers for Non-Standard Applications, November 1994.