Transaction Support for Design Processes

—Requirements, Concepts, and Realization Primitives

Andreas Geppert

Technical Report 93.45

Forschungsbereich Datenbanktechnologie
Institut für Informatik
Universität Zürich
Email: geppert@ifi.unizh.ch

June 10, 1992

Abstract

During the last decade or so, concepts for database support for non-standard applications like CAD or CASE were developed. Simultaneously, it turned out that the classical transaction concept does not sufficiently support those design applications. New transaction models —such as long-lived transactions or cooperative transactions— have been specified for this application domain. This paper first describes common requirements for design transactions. It then surveys proposals that address these requirements. Furthermore, commonalities of the proposed approaches are factorized and a set of primitives for the realization of design transaction management is proposed.

Contents

1 Introduction

After it turned out that classical database systems served traditional applications like book-keeping rather well, other (so-called “non-standard”) application domains like CAD, CASE, office information (OIS), geographic information systems (GIS) desired to benefit from database support as well (application-oriented data modelling capabilities, data independence, transactions). While there is sufficient agreement on the requirements for database systems for “non-standard” applications as well as on the insufficient support of classical systems and models for those new application domains [Gray 81, Lockemann 85, Dittrich 85, Bernstein 87b], no consense exists how data models and database systems supporting those domains should look like. Consequently and unfortunately, a broad variety of data models has been advocated: extensions of existing (i.e. relational) ones like [Haskin 82] or object-oriented [Atkinson 89] ones.

Of course, new requirements have also been raised for transaction management, e.g. support for cooperation of designers in CAD or CASE applications. However, since the underlying data models already differed to a large extent, transaction concepts for design applications vary considerably among the different non-standard database systems.

The intension of this paper is first to survey the various requirements and concepts for design transaction management, and second to figure out the commonalities between them. Based on the commonalities, *primitives* for the realization of those basic functionalities are specified, which later on may be used in extensible or configurable database system for the construction of design transaction components in advanced database systems.

1.1 Terminology

Up to now, I have intuitively used the well-known term “transaction”, although actually a *variety* of concepts is meant. Indeed it is questionable to use the term for different kinds of units of work out of at least two reasons:

1. If “everything” is termed as “transaction”, this leads to an abuse of notation: sometimes readers cannot be sure what semantics are really intended.
2. Transactions and “problem solving cycles” [Wedekind 89] (i.e., design transactions) are quite different things in that they have distinct logical backgrounds.

Thus, it seems desirable to clearly separate classical transactions from problem solving cycles as far as notation and notion is concerned, and to introduce a new general term “unit of work”. However, most of the authors active in the field use the term “transaction” even if “problem solving cycles” are meant. Therefore, in this paper I also use “transaction”, though with additional adjectives. Concretely,

- the transaction concept and model as found in [Bernstein 87, Gray 78] is called the *classical transaction* model.
- Transactions are called *flat transactions* when the fact is important that they cannot contain other transactions (i.e. have a flat structure). Like in [Moss 81], a transaction that in turn may spawn (sub)transactions is called a *nested transaction*.
- When focussing on the property that transactions cannot span more than one user session or operating system process they were initiated in, they are termed *short transac-*

tions. If a transaction execution can exceed its original operating system process, it is called *long transaction* or *long-lived transaction* [e.g. Garcia-Molina 87].

- Finally, a transaction supporting design applications in the sense that it allows for controlled and synergistic cooperation (mostly beyond serializability), is termed *design transaction*. Note that a long transaction need not necessarily be a design transaction.
- The term *unit of work* may be used as a general notion where no special characteristics are assumed.

1.2 Scope

In this paper, I only survey concepts and models for design transactions, i.e. transactions for design and related applications. What the requirements for such application domains are will be described in the next section. Nevertheless, I assume the reader to be familiar with the classical transaction model and only give a short introduction in this paper (see [Bernstein 87, Gray 92] for a detailed overview). Semantic concurrency control (e.g. for abstract data types) is also not covered here. Finally, I will not survey transaction concepts dedicated to heterogeneous environments or groupware systems [e.g. Ellis 89]. While (in the long run) I investigate the realization of single transaction management subsystems (i.e., of subsystems of a single DBMS, which natually is a homogeneous one), heterogeneous transaction management deals with the integration of a variety of existing transaction managers. Hence, although some techniques may be useful in both areas (e.g., frameworks [Drew 92]), underlying problems are quite different.

2 Classical Transaction Management

Since throughout the rest of this paper I will occasionally refer to characteristics and properties of traditional transaction management, and since some approaches surveyed in the next section extend specific concepts, I will give a short overview on (classical) transaction management in this section.

In the classical model as described by [Gray 78, Bernstein 87], a transaction is a sequence or partial ordering of read or write operations. A *schedule* (or history, log) is a (possibly interleaved) execution of multiple transactions, and therefore can be identified by the corresponding set of transactions and an ordering among the operations of the transactions. A schedule that does not interleave transactions, i.e. executes one transaction at a time, is called *serial*.

In a schedule, a write operation issued by a transaction *conflicts* with any other operation performed by another transaction on the same object, while read does not conflict with any other read of the same object (hence, read is said to *commute* with itself). Two schedules are (*conflict*) *equivalent* if they execute the same set of transactions and order conflicting transactions in the same manner. A schedule is said to be (conflict) *serializable* if it is (conflict) equivalent to a serial schedule.

Different kinds of serializability are possible [Hadzilacos 88b], the one alluded to above is called *conflict serializability*. A weaker kind is state serializability, which allows for the reordering of even conflicting operations, but requires the two (equivalent) schedules to result in the same database state. Nevertheless, conflict serializability is the more attractive one out of complexity theoretical reasons.

While conflict serializability is a stronger criterion than state serializability, there are even stronger versions of serializability (used in practice):

- *Recoverability:* a schedule H is recoverable, if for any two transactions T_1 and T_2 of H the following condition holds: if T_2 reads an item that was written by T_1, than T_2 does not commit before T_1,
- *Avoiding cascading abort* (ACA): a schedule H is ACA if for any two transactions T_1 and T_2 of H the following condition holds: if T_2 reads an item that was previously written by T_1, then T_1 commits or aborts before T_2 actually reads,
- *Strict* serializability: a schedule H is called strict if it is ACA, and the following condition holds for any two transactions T_1 and T_2 of H: if T_2 writes an item that was previously written by T_1, then T_1 commits or aborts before T_2 actually performs its write.

Apart from their mathematical characterization, classical transactions are often described by means oof the so-called ACID principle [Haerder 83]:

- ***A****tomicity*: a transaction is either executed in its entirety or not at all ("all or nothing"-principle).
- ***C****onsistency*: while the database state during a the execution of a transaction may be inconsistent, consistency has to be preserved at the latest on transaction termination.
- ***I****solation:* Any transaction can solely "see" committed changes of other transactions.
- ***D****urability*: After commit, all the modifications of the corresponding transaction are ensured to be permanent; no modifications of uncommitted transactions can become durable.

Additionally, in the inititial proposal typical transactions were assumed to be short, i.e. their execution was assumed to last in the range of milliseconds or seconds. Transactions were flat in the sense that from the transaction management point of view they were composed out of database operations only, but could not spawn other transactions in turn.

Different techniques are proposed for ensuring (strict) serializability, they are usually subdivided into conservative, pessimistic, optimistic, and multi-version concurrency control techniques. Conservative concurrency control determines the set of objects to be accessed by a transaction at transaction start time. Thus, potential conflicts are detected then, and a conflicting transaction can be refused to execute. Pessimistic approaches check whether the execution of an operation conflicts with those of other transactions just before the operation would be executed. If a conflict occurs, the corresponding transaction is blocked or aborted. Examples for pessimistic techniques are timestamps, strict two-phase locking, and (less prominent) serializability graph testing.

While optimistic approaches check for conflicts at the end of a transaction (and in the presence of conflicts disallow the transaction to commit), multi-version concurrency control [Bernstein 83] (in principle) even allows parallel reads and writes on different versions of the same item.

The most important criterion for a concurrency control technique is the *degree of parallelism*, while the overhead (due to synchronization) has to be tolerable. In order to support higher parallelism, a number of extensions (mainly to two-phase locking) has been proposed. A first extension is to consider the semantics of operations and types beyond read and write. As [Korth 83] shows, the conventional locking technique can be extended to incorporate lock modes for operations like increment and decrement of counters; if conflicts are defined properly, a higher degree of parallelism can be achieved while serializability is still preserved. [Schwarz 84] proposes to synchronize operations of abstract data types, whereby again the exploitation of dependencies (similar to conflicts) between type specific operations beyond read and write implies an increase in concurrency. The approach of [Schwarz 84] later on has been extended for synchronization in object-oriented database systems (e.g. [Hadzilacos 88a]). These approaches are also subsumed under the notion of *semantic concurrency control*.

Another way to increase parallelism is to change transaction structures, i.e. to subdivide transactions into smaller pieces which are then synchronized with each other. In the concept of *nested transactions*, a transaction in turn can execute (sub)transactions. Thus, in contrast to the classical (flat) transaction model, transactions form a tree, and a set of transactions can be represented by a forest. In the *closed nested transaction* approach [Moss 81], transactions that are not ancestors or descendants of each other are still isolated. However, siblings can execute concurrently, implying an increase in *intratransaction* parallelism. In the *open nested transaction* approach, locks are released at the end of a subtransaction, thus nested transactions are no longer isolated from each other. As a consequence, even *intertransaction* parallelism (i.e. parallelism between distinct transaction trees) increases. Open nested transactions can be used to realize transactions in layered systems [Moss 86, Weikum 91]: in this case, locks are released at the end of a subtransaction, while the next higher layer still holds lock on more abstract objects. While parallelism increases serializability is still preserved (*multi-level serializability*).

Although these different modifications of the original transaction concept increase parallelism, they are not sufficient for design transaction management, at least not without further modifications and extensions. In fact, parallelism in the classical approach is an “illusion”, since interleaved executions are equivalent to non-interleaved, serial ones. This is different in design applications, where designers should be allowed to work simultaneously. Likewise, other assumptions (like on transaction execution duration) are no longer justified. Therefore, I will describe requirements and characteristics of design applications for transaction management in the next section.

3 Requirements for and Characteristics of Design Transaction Management

This section describes some characteristics of most design applications, which result in the demand for new transaction concepts.

Long-Lived Applications

One major difference between design and conventional applications is their *duration*. While classical applications like banking or booking have a duration in the range of (milli) seconds, design applications may take from hours up to weeks or months. As a consequence, design transactions have to be independent of the operating system process in which they have been initiated. The traditional approach to abort any active transaction in case of operating system crash is not feasible in that context. In other words, design transactions are not subject to recovery in the traditional sense.

Furthermore, in the classical model strict two-phase locking is often used in order to ensure *strictness* [Bernstein 87] (if locking is assumed), i.e. locks are held until commit. In design applications, however, some designers (or their transactions) would have to wait for a prohibitive amount of time until they might access desired objects. Thus, (strict) two-phase locking is not applicable for design environments, at least not without extensions or modifications. Anyway, design transactions in principle are not (synchronization) *atomic*.

Cooperation

In the classical model, changes are usually made visible to other users at transaction commit (i.e. at the end of a unit of work; *isolation*). Only those transactions leaving the database in a consistent state may be committed. In design applications multiple designers may (and often will) need to share preliminary (and thus still inconsistent) work of other users. Thus, isolation has to be circumvented for specific groups (or teams) of designers. The members of those teams should be able to *cooperate*, i.e. to exchange preliminary parts of their work. Due to cooperation, classical conflict serializability is again not an adequate correctness criterion for design transactions.

Concurrency

As stated above, design transactions are not useful as units of recovery and of synchronization atomicity, thus —in terms of the ACID principle— are neither atomic nor isolated. However, for specific units of work these properties may be desired. In other words, not every units of work is a design transaction, and hence design database systems should subsume the functionality of classical database systems, at least as far as transaction management is concerned.

Project Structures

As already mentioned, designs are typically not performed by single persons, but by groups of designers. Projects are usually subdivided into subtasks. These may be assigned to groups or teams, which in turn consist of subgroups or single designers. Designers may delegate specific tasks to other users. Thus, while conventional database systems often support the notion of “users” only, design database systems have to provide for constructs modelling the *structure of projects*. The design transaction concept has to cope with those structures: e.g. members of the same group may share intermediate results, while preliminary work is not visible to designers of other groups.

Flow of Control

Usually, the tasks of a complex design are not unrelated, but depend on each other, for instance in a software life cycle. In this case, a “procedure” defines which steps of a complex design have to follow each other, and rules are given which sequences of steps or tasks are not correct in a design process. Therefore, it would be nice to let a design database system organize and supervise the various steps, i.e. forbid or enforce specific sequences of unit of work.

Distribution

While conventional database systems evolved from centralized to distributed ones [Özsu 91], where data can be stored on various sites in an (ideally) transparent manner, design applications often are tailored to workstation/server configurations. In this case, public or released data are stored on a server, while the design efforts of the designers are performed on the respective workstations. For that matter, distribution transparency is not necessary and not provided, and the design transaction model has to fit together with the workstation/server model.

Data Models

Of course, design applications also have to be supported by the data model. It should at least provide for *complex objects* [Lockemann 85] and *versions* [Katz 90]. Furthermore, *object-oriented* data models are advocated to be well-suited for design applications. In this paper, we are not interested in the data models themselves, but rather in the interrelationships between the data model and the design transaction concept, or in the question how tightly the latter depends on the former.

Supervising Change and Consistency

In a design process, inconsistency can result from an individual designer's work, e.g. when design guidelines are not obeyed. Additionally, inconsistent database states can show up as designers cooperate and share work. In this case, the danger raises that designers are not aware of changes that have been performed by others, e.g. if new versions are created, thus probably invalidating versions which some designers think to be valid.

Hence, while the synchronization mechanism in a design environment disallows incorrect interleavings or cooperation and thus provides for operational integrity, consistency can be violated since design objects as well as design transactions semantically are not isolated from each other. Therefore, changes for one object may also affect other objects indirectly.

While the notion of "consistency" depends on the data model and the concrete design application (schema), the design transaction subsystem should support mechanisms to detect or supervise changes and to take appropriate actions, if necessary.

4 Concepts

In this section, I survey techniques and concepts as proposed in the literature. This survey follows the list of requirements and characteristics of section 3.

4.1 Project Structures

In general, two approaches exist for the representation of project structures: those can be modelled explicitly by *users* and *user groups*, or implicitly by nesting transactions, where a nested transaction corresponds to a group or team.

4.1.1 User Groups

The notion of *user* is known from traditional database systems, and the notion of *user groups* is also present in operating systems like Unix. In both cases, they are used mainly as a mechanism for protection and visibility.

XSQL [Haskin 82, Lorie 83] and DAMOKLES [Dittrich 86] are similar in their treatment of users and user groups. For that matter, both [Dittrich 87a] and [Rehm 88] propose a directed, acyclic graph of users and user groups, where the leafs are the individual users (e.g., designers). Furthermore, *logical databases* can be assigned to users and groups, and generic rules are defined for accessing databases of groups (e.g. only the group or a direct member may access a group database).

Group graphs explicitly model the structure of (possibly multiple) projects, teams, and so on. No structure is superimposed beforehand, and various tasks (in a design process) can be assigned to groups or users (outside the database system). That is, groups and users are not identified with their role in a concrete design process.

Given a graph of users and user groups and attached logical databases, a design transaction always spans two databases: the *original* database belonging to a group, and the *design* database, which belongs to a member of the group.

4.1.2 Nested Transactions

In various approaches *nested transactions* [Moss 81] are used for the representation of groups and their tasks. This idea seems rather straightforward, since modularization of units of work often is mentioned as an advantage of nested transactions.

The proposal of [Klahold 85] assumes a fixed two-level structure, consisting of the public, several groups, and users. Each group of designers is represented by a *group transaction*, while designers perform their work within *user transactions*. To each group transaction a group database is attached, which exists only for the lifetime of the group transaction. Group transactions operate on the public database and copy objects to their group database. Later on, different users can copy objects to their private database (according to the synchronization protocol).

Likewise, [Kim 84] is another approach that uses nested transactions. There, each *engineering transaction* has a semi-public and a private database attached. Additionally, a *public database* exists. An engineering transaction may checkout objects from the semi-public database of any of its ancestors. A top-level transaction may also checkout data from the semi-public database of another top-level transaction, and thereby becomes a child of the latter. Hence, transactions can become a child of another transaction dynamically (i.e., after begin of the transaction).

CAD transactions [Bancilhon 85, Korth 88a] also use a multi-level approach based on nested transactions. There, multiple *project transactions* can run concurrently. A project transaction is a set of cooperating transaction, where a cooperating transaction is a hierarchy of client/subcontractor transactions. A client/subcontractor transaction in turn is a directed acyclic graph of short transactions. Thus, following [Bancilhon 85], a design effort is subdivided into projects, which in turn are performed in cooperation by designers or groups. Designers may delegate specific tasks to subcontractors (transitively). Again, nested transactions represent groups *and* their tasks.

Finally, *cooperative transaction hierarchies* [Nodine 90a, Nodine 90b] also use nested transactions for the modelling of project structures. A transaction hierarchy is a tree, where the root and the internal nodes are transaction groups and the leaves are designer transactions. The hierarchy is assumed to represent the decomposition of the design task, that is, nested transactions again model the tasks of groups. In contrast to most other nested transaction based approaches, the nesting depth is arbitrary and new members can be added dynamically to a transaction group.

4.2 Long-Lived Units of Work

As mentioned above, the “all or nothing”-principle is not appropriate for design transactions, rather has such a system to support for durable, small units of work additionally.

The usual approach for durable chunks of work are *checkpoints***.** In case of a user-initiated abort or a system crash, all work performed after a checkpoint is undone, while modifications performed before the checkpoint is guaranteed to persist. Most approaches or systems propose the use of classical and short transactions for that matter, i.e. a checkpoint is done at commit of each of those transactions.

Sagas [Garcia-Molina 87] are units of work that support applications of long duration. A Saga is an open-nested transaction with short, flat transactions as subtransactions. Committed subtransactions are durable, while in case of a crash each active subiltransaction is undone. Hence, subtransactions are failure atomic. Entire Sagas can be aborted, for that matter a compensating transaction has to be specified for each subtransaction.

In [Lorie 83], *recovery transactions* are proposed for XSQL. The start of a recovery transaction establishes a savepoint, and no changes get permanent if the transaction is aborted later on. Since it is assumed that designers work on private databases in a single-user mode, recovery mechanisms for recovery transactions can be kept simple.

Likewise, [Kim 84] views long transactions as sequences of short ones providing for durability and atomicity. In this approach, even design transaction commands like checkout have to occur in some short transaction and are recoverable.

Although *CAD transactions* [Bancilhon 85, Korth 88a] differ from transactions in other approaches in their structure, they also support short transactions as units of recovery.

In DAMOKLES, Damokles Design Transactions (DDTs) [Rehm 88] and short transactions are orthogonal. Short transactions essentially are closed nested transactions [Moss 81], thus are units of (failure as well as synchronization) atomicity and durability. However, atomicity of operations like checkout and checkin is guaranteed.

4.3 Cooperation

The notion of “cooperation” in fact subsumes various facets:

- cooperation can have the meaning of exchange of data among designers or groups. Here, data can be preliminary or inconsistent, and exchange can be restricted to members of a specific group. Exchange in turn can be realized in that a (designer) transaction releases data (before final commit) that are not needed furthermore, or in that data is “handed out” explicitly to a specific transaction (representing a designer or group). In the latter case one speaks of a *subcontractor/client* relationship. In any case, uncon-

ditional isolation as in the classical model of [Gray 78] is weakened to conditional isolation.
- cooperation is possible in the sense of simultaneous access to data. Here, dirty read may be allowed as well as non-restricted read, i.e. read that allows updates of other transactions. Thus, serializability is given up as well as synchronization atomicity.
- cooperation can be realized in that specific sequences of operations (be they generic or user-defined) are allowed/enforced/forbidden. As an example, one might specify that any designer has to read the design before starting an implementation.
- one step further, cooperation can be thought of as arbitrary interleaved execution of actions (on behalf of designers and their transactions) that *leads to a consistent state*. For that matter, *pre-* and *postconditions* can be specified for transactions. Hence, this way it is made explicit what a consistent database state is.

Apparently, the first two variants are mainly based on the semantics of data (types) and operations, while the third alternative is based on information on the semantics of concrete transactions and on correct interleavings between them. Note that in the second approach, the knowledge on "permissible" interleavings is provided by the user, i.e. not inferred by the system. Therefore, these variants are termed the *data* and the *transaction approach* in [Skarra 89]. In addition, the last variant could be termed *consistency approach*, since it abstracts from transactions, and consistency is made explicit.

In the sequel, I will survey cooperation in the framework of design transactions along the lines of these three classes, though they are not disjoint in the "real world". For a concise description of cooperation and synchronization in the various approaches, some characteristics of the corresponding data models will be in place.

4.3.1 Data Approaches

Nearly all data approaches make use of the checkout/checkin paradigm. The checkout operator copies objects into the workspace (or logical database) of a user or group and usually locks the copied objects in the original or group database. Most approaches propose an enhanced set of lock modes in comparison to the traditional read and write locks, e.g. for "dirty read" of non-committed data or non-protected read that allows other transactions to perform modifications on the locked objects even before the reader terminates. Additionally, in order to support long-lived, collaborative work, locks granted at checkout have to be persistent. After working with/on (e.g. updating) the object, the designer checks the objects in; i.e. the objects are copied back to the original database. Locks may be released at checkin or at the end of design transactions. Checkout/Checkin themselves may be *two-phase* or not; in the first case no further checkout may be performed after the first checkin. In the strict case, checkin operations are grouped at the end of the design transaction.

XSQL

XSQL [Haskin 82] extends System R [Astrahan 76] with *complex objects*. Pre-defined attributes comp-of and identifier are introduced for relations. If a tuple contains the identifier value of another tuple in its comp-of attribute, it is treated as a "subobject" of the other tuple. Objects can be checked out, for that matter they are locked in S-mode (where subobjects are locked in the same mode implicitly). Furthermore, a user can specify what he/she intends to do with the object: he/she may desire to update the object, thus excluding further checkouts, or he/she may allow further checkouts, but not for update. As usually, checkin is the reverse operation of checkout. However, if a checkin has to be performed, first an X-lock was to be granted on the object to be checked in. When performed, checkin marks the object lock to be released as soon as the checkin transaction commits. The user then is responsible for updating the original object. The operation migrate copies entire complex objects and implicitly performs a checkout in the source and a checkin in the destination.

In [Lorie 83], *conversational transactions* are introduced. There, cooperation is supported in that the set of lock modes is extended: beside the usual modes shared read and exclusive write, two new modes are introduced: read without forbidding other transactions to write, and write without delete permission.

Furthermore, [Lorie 83] proposes to group checkin operations at the end of a design transaction, thus leading to some kind of isolation between multiple design transactions. Additionally, all checkins are performed together with the commit of the conversational transaction, thus a conversational transaction cannot checkin objects and later on abort.

Group Transactions

[Klahold 85] describes a two-level design transaction scheme with integrated version management. The group databases contain the objects in work, thus required objects are copied to these databases by means of checkout. *User* transactions operate on the user's private workspace and can be regarded as subtransactions of group transactions, since they are only allowed to access or manipulate data that had been checked out by the (parent) group transaction. Similar to group transactions, user transactions perform checkouts in order to copy objects in the private workspace.

This model then is enriched by the concept of *versions*. What "version" actually means is left to the user; the system simply manages a (directed, acyclic) version graph. Access to versions is integrated in the design transaction concept, i.e. operations on versions are synchronized by means of specific lock modes. Versions can be subject to checkout, the user has to indicate the kind of access he/she wants to perform.

The proposed model also supports client / subcontractor procedures. If a designer wants another one to work on versions he/she has checked out (and which are locked in an approbriate way), a specific operator grant can be applied in order to tell the system that the respective colleague is allowed to read (and modify, but not checkin) the versions in question.

[Kim 84]

As a major difference to XSQL, transactions in [Kim 84] can checkout objects from any transaction (or, more precisely, from their semi-public workspace) in their ancestor chain. Beside the usual checkout, two other operators are important: first, downward-commit allows other transactions to access (i.e. checkout) some objects in the transaction's semi-public database. Upward-commit is essentially a checkin, i.e. it transfers control over (formerly checkedout) objects back to the parent transaction. If a transaction terminates, all checkedout objects are upward-committed automatically.

CAD - Transactions

CAD transactions as proposed in [Bancilhon 85, Korth 88a] propose hierarchies of different types of transactions, as described above.

Project transactions are assumed to elaborate on different parts of a large design, and thus are isolated from each other. Cooperating transactions, on the other side, have the ability to exchange preliminary data. Furthermore, short transactions are atomic.

CAD transactions also are based on the checkout/checkin paradigm, whereby these operations are required to be two-phase. Each subcontractor or client transaction has a subcontractor, a client, and a private space assigned to it. The private space (respectively, the objects located there) can only be accessed by the transaction itself, while the subcontractor space contains objects that may be checked out by subcontractors. The client space of a transaction essentially is the subcontractor space of the respective client. A transaction controls its subcontractor space (and thus, the set of objects for which checkout is permitted) by means of the operations checkout-enable (checkout-disable), which moves (removes) objects to (from) the subcontractor space. Checkout/checkin have to be two-phase, i.e. after the first checkin no further checkouts are allowed.

Damokles Design Transactions

The underlying data model of *Damokles Design Transactions* [Rehm 88] provides for structured objects, versions, and explicit relationships, and for generic operators on those.

A DDT is initiated by a user or group, and has always two logical databases attached to it: the design database which must be possessed by the user or group, and an original database to be owned by a group where the initiator of the DDT is member of. Checkout and checkin always relate to a specific DDT, which in turn identifies the logical databases in question. Checkout always operates on structured objects, i.e. components of a complex object always are checked out whenever a superobject is to be checked out. The same holds for versions. However, versions cannot be checked out on their own, since they always have to be located in the same logical database as their generic object.

At checkout, it has to be specified if the (structured) object will be modified or solely read. As usual, in the first case, the object and all its subobjects and versions (and so on transitively) are locked in an appropriate mode. Additionally, it can be specified which access modes are permitted for other design transactions. Here, the designer can protect the object against modifications, against modifications of its context (e.g. external relationships), or against read access. Alternatively, non-protection is possible when the object was checked out for reading only. Nested checkout of objects along the group/user graph is possible as long as locks do not have to be upgraded (i.e. a further checkout does not require stronger locks than the existing ones).

As soon as the work in the design database is finished (i.e. not necessarily at the end of the DDT), the design object can be checked in the original database. Alternatively, the work with a checked out (perhaps modified) object can be aborted by means of an uncheckout operation. Similarly, the entire DDT may be aborted by means of abort_lta, which causes an uncheckout for all checked out objects that have not yet been checked in.

Furthermore, [Ranft 90] proposes some extensions to [Rehm 88] in order to increase potential parallelism. Due to complex object locking, it may be the case that “too much” of an object is locked when only parts of an object are to be modified. Hence, it is proposed to checkout objects with no or a weak lock mode and to aggravate (upgrade, intensify) locks afterwards. In this case, designers can lock exactly those required parts of a complex object. However, since modifications of different parts of an object may be related semantically, designers can specify *groups* of objects, with the semantics that all members of a group should always be locked in the same mode. Hence, if afterwards another design transaction requests an incompatible lock on one of the elements of the group, this lock request will be rejected.

Split and Join Transactions

Split transactions [Pu 88] provide for a mechanism to release objects that are no longer needed by a transaction. For that purpose, a transaction is *split* into two serializable transactions. Both have to declare their read and write sets as well as a procedure where to resume execution. Read and write sets are collections of objects accessed in the respective mode by the transactions. In order to guarantee serializability, three conditions have to be fulfilled: first, if an object is written by both transactions, then the second overrides the first. Furthermore, the first may not read any of the objects written by the second, but the second may read objects modified by the first. Furthermore, if both transaction do not modify the same object and the second does not read objects modified by the first, both can be serialized in any order. Since the system serializes the entire ("splitted") transaction with all others, overall serializability is guaranteed when the former two are serializable among each other. The *join* operation is the inverse of split, when issued against a transaction, the changes of that transaction are committed together (atomically) with the effects of the caller transaction.

Release of intermediate results is possible in that the respective transaction splits and one of the transactions (which then declares the objects to be released in its read and write set) commits. Objects to be modified further are assigned to the second, further running transaction.

Sagas

In the Saga approach, locks (if they are used) are released at the end of each subtransaction. Hence, objects are visible and accessible to the outside world after commit or abort of a subtransaction. However, this is not *controlled* cooperation, since access to released (possibly intermediate) result cannot be restricted to a specific group of colleagues.

An extension of Sagas proposes *nested Sagas* [Garcia-Molina 91]; then a nested Saga may be used to model a team or group task, and a Saga or subtransaction can be committed to the parent Saga.

4.3.2 Transaction Approaches

Cooperative Transaction Hierarchies

In [Nodine 90a], hierarchies of *cooperative transactions* are proposed. The root of such a transaction hierarchy represents the work of an entire project. The inner nodes of a transaction hierarchy represent the various groups, while the leafs represent the individual designers. A group transaction is responsible for correct interleavings of its members. Concurrent access to objects is defined by *protocols*. Each transaction group has an external and an internal protocol. While the internal protocol of a transaction group specifies the permitted interleavings of its members, the external protocol specifies correct interaction with its siblings. In general, protocols are specified by *operation machines*, which essentially are finite-state automatons. Each transition of an automaton is labeled with a quadruple, consisting of

- the identification of a concrete member, the negation of a member, or specification of any member,
- the operation (read or write),
- an object identifier,
- a return value (accept, refuse, or queue).

If no transition from a given state for a specific operation is defined, that operation is not relevant in that state.

In [Nodine 90b], *operation machines* are state automatons that specify how transactions can be interleaved on an object. For each transaction T that accesses an object O the pattern machine is instantiated to a *synchronization machine* for the pair (T, O). With such pattern machines, serializability as well as other (more cooperative) interleavings can be described. Operation machines can also be used as *pattern machines* that describe how a specific task has to be achieved, or as protection machines for access control.

In [Nodine 90a], a transaction group possesses a set of object versions to be accessed by its members. Different versions of the same object can exist in the hierarchy. Any time a member reads a version, that version is copied from the parent into the member's set. When modification is finished, the version is written back. The existence of different versions impacts potential inconsistencies. Therefore, *intentions* are used for ensuring consistent access to versions. Intention grant a transaction the right to perform a specific operation (only once, in contrast to locks). If an intention has been granted, the group ensures that the corresponding operation can be executed any time without conflicting with other transactions.

4.3.3 Consistency Approaches

CAD - Transactions

While transaction structures in [Korth 88a] are similar to [Bancilhon 85], an extended synchronization mechanism is proposed there.

While project transactions are assumed to be isolated from each other, isolation should be relaxed for cooperating transactions. For that matter, [Korth 88a] makes use of explixit consistency constraints. For each transaction, its *invariant* is given in conjunctive form, i.e. as a conjunction of predicates. Then, if C is a conjunction $c_1 \wedge \ldots \wedge c_n$ and d_i is the set of objects mentioned in c_i, then predicatewise two-phase locking is two phase with respect to each set d_i solely. A lock compatibility function is called predicatewise correct wrt C iff, given a set of transactions which observe predicatewise two-phase locking, a scheduler can only generate schedules that preserve C. Intuitively, this means that transactions can be serialized in different orders for different predicates, i.e. it is possible that a schedule is not serializable in the classical sense, but nevertheless preserves consistency.

ConTracts

The concept of ConTracts [Wächter 90, Reuter 91] is the integration of single application steps and the control flow within the database system. For that matter, it is distinguished between *steps,* which are the single (atomic) application units or programs (e.g. specific units of work in a design process) and *scripts*, which specify the overall proceeding, i.e. the flow of control. A ConTract then is the execution of steps according to a script. For the various steps, *compensating steps* may be specified, but are not mandatory.

Additionally, *entry* and *exit invariants* can be defined. Invariants are predicates that have to hold at the beginning and at the end of steps, respectively. Thus, instead of locking all accessed or modified objects for long times, each step specifies its view of the (consistent) database. Additionally, each step can specify the invariant that is necessary for the next step(s) to execute.

In addition to the declarative specification of invariants, a *policy* can be specified how to ensure the invariants (e.g. check/revalidate). Furthermore, since nevertheless conflicts (with other steps of other contracts can occur), it can be described for a step how to resolve conflicts (e.g. to raise a budget in case it is overdrawn).

Nested Transactions with Predicates and Versions (NT/PV)

NT/PV was proposed in [Korth 88b] and its application demonstrated in [Speegle 90]. It is argued that serializability is too restrictive for design transactions and that it may be weakened when explicit consistency constraints are taken into account. Additionally, increased concurrency and cooperation are supported by the use of versions and nested transactions. As far as the formal model is concerned, assume that an overall database consistency constraint is given in conjunctive form. Instead of requiring the global serialization graph to be acyclic, such a graph is defined for each set of data items in one of the conjuncts[1]. If each of these graphs is acyclic, the schedule having produced them is said to be *conflict predicate correct.* Thus, transactions may be serialized in different orders on disjoint sets of data items according to the conjuncts.

This model is implemented as follows: a transaction is a quadruple (T, P, I, O), where T is a set of subtransactions, P is a partial order of the subtransactions, and I and O are predicates on the database state. I is called the input and O the output constraint. Whenever a transaction started in a database state satisfying the input constraint and is not interfered with other transactions, it is assumed to preserve its output constraint. Execution of a transaction consists of four steps: transaction *definition*, transaction *validation*, transaction *execution*, and transaction *termination*. While during the first phase input and output constraints are defined for the transaction, versions of data items are assigned to it during the second phase. For that matter, *read for validation* locks have to be acquired on the data items (not on the versions). Re-evaluation may be necessary, as well as assigning new versions to the transaction, e.g. when a predecessing transaction recently wrote a new version.

1.For the definition of serialization graphs, see [Bernstein 87]

4.4 Concurrency

Even in a design database system, not all transactions are desired to cooperate with others. For instance, assume the administration of project budgets to be supported by the same database system as the corresponding project designs; hence users will expect support of classical transactions as well.

This is possible in that the public database is designed as a “usual” multi-user system, like in XSQL [Haskin 82]. Perhaps a more sophisticated approach is *orthogonality* of short and design transactions, like in DAMOKLES, where short transactions are not required to execute within a design transaction, are serialized with other short transactions, and therefore can be easily used for applications that desire isolation.

Concurrency (with isolation) is not (at least not very well) supported in models or appraches that use nested transaction to represent group structures and that require short transactions to occur in cooperating or design transactions (like CAD transactions). In this case, it might be necessary to start a project transaction for each simple debit or credit of a project budget.

4.5 Consistency

The approaches surveyed so far mainly ensure consistency in the sense that they do not allow incorrect interleavings of (design) transactions (operational integrity). However, further mechanisms are required for ensuring *semantic integrity.* In a typical design or non-standard environment, this kind of integrity will be quite more complex than in traditional application domains. As an example, take programming or documentation guidelines in a CASE environment, or physical laws to be obyed in a VLSI design system. Hence, from my point of view it is questionable whether even the consistency approaches surveyed above can handle such complicated consistency constraints. Additionally, in such environments tools for checking designs usually already exist (e.g., compiler-like programs) such that it seems necessary to connect them to the design database system, rather than to enhance design transactions with explicit consistency constraints.

A general form of a consistency constraint is a quadruple that specifies

- the object(s) concerned by the consistency constraint,
- a logic expression to be fulfilled by the objects,
- the point in time when the consistency constraint has to be evaluated,
- a reaction to be taken in case the objects do not preserve consistency.

In the following, I shall survey some proposals for integrity management in the area of design transactions.

The Version Server

The *version server* [Katz 87] integrates checkout mechanisms with quite sophisticated version support, where “version” can have the semantics of configuration (i.e. it consists of other versions), revision (i.e. versions may be predecessor or successor of other versions in a version graph), and representations (i.e. two versions related by some equivalence relationship represent the same real-world object). Checkout of objects always creates new versions. Checkin, in contrast to the approaches described above, does not overwrite the original version, but establishes the new (and later on probably modified) version in the group workspace. Furthermore, the object or version to be checked in is validated at checkin time, i.e. is validated against a series of verification events in that CAD verification programs are executed.

A first-shot solution is to disallow checkin whenever an inconsistency is detected by the verification programs. A more sophisticated approach, however, is to “repair” inconsistent configurations. For that matter, *change* propagation is proposed. Change propagation means to create new versions (of related objects) and to incorporate them in a new configuration. Here, “related object” means related via an equivalence relationship (especially across representations). In order not to propagate changes for each new version, *group checkin* can be performed such that only *one* new configuration is created for the entire group of versions.

Concrete consistency constraints in this model are the following:

- *dependency constraints*: a configuration cannot exist outside another configuration,
- *timestamp consistency*: a configuration is not older than its components,
- *interface constraints*: it is possible to replace the ancestor with the new version,
- *representation-dependent containment*: the component of a configuration are properly contained within it.

Change propagation proceeds recursively; for the first case of constraints it stops as soon as an independent configuration is encountered. In the other cases propagation stops when otherwise the respective constraints would be violated.

Thus, compared with the quadruple model from above, the objects in question are the versions, the conditions are the various constraints, the point of time is execution of checkin, and reaction is not to allow checkin or change propagation, respectively.

ConTracts

ConTracts also provide some kind of forward recovery [Reuter 91]. When executing a ConTract, an inconsistent state may be detected in that some invariants are not valid. In those cases, the *conflict resolution* clause of a specific step can specify other steps to be performed in order to repair the inconsistent state.

ECA - Rules and Trigger Mechanisms

Another mechanism to check integrity constraints at specific points in time and to perform appropriate actions are *ECA rules* (*event-condition-action*) or *event/trigger mechanisms* (ETM).

In the former [e.g. Dayal 88, Gatziu 91], an event specifies when the corresponding condition has to be checked. If the event occured and the condition was evaluated to *true*, the respective action is performed. Of course, ECA rules may be used for supervising consistency. Hence, an event may be e.g. checkin of a design object, while the condition checks whether the design is consistent or not, and a possible action may be change propagation as above.

In the latter approach [Dittrich 87b, Kotz 88], a trigger consists of an event and an action. If the event is raised, the action is performed. Like in the ECA approach, specific events can be raised by users or applications (*abstract events*). Compared to the ECA model, a condition cannot be specified for a trigger, thus has to be encoded within the action.

However, this fact is not very harmful if the consistency check (i.e. the condition necessary for executing the action) is done by verification programs anyway.

ECA rules as well as triggers are very general and indeed are applicable not only for consistency supervision. Due to their general nature, they are often regarded as too complex when (ordinary) users are free to define arbitrary, large sets of triggers or rules. Therefore, from my point of view they are rather a basic, primitive mechanism that can be used —amongst other purposes— for integrity maintenance.

4.6 Recovery Issues

In a broader sense, recovery re-installs a *consistent* state in case of integrity violations. The usual mechanism is to *undo* transactions that would violate integrity when committed. Since this kind of recovery installs an earlier database state, it is called *backward recovery.* As a matter of fact, design transactions potentially can violate integrity, but are no reasonable units of backward recovery. What is desired is *forward* recovery that performs actions leading to a consistent state without backing out the work done so far within a design transaction. However, when compared to the cooperation aspect, only little work has been done within that context.

One alternative is change propagation, as described above. Essentially, change propagation is a kind of forward recovery.

In the change propagation approach, *checkin* (or more general, modifications) of versions can invalidate other versions. Likewise, modifications performed in one design transaction can invalidate work or views of others. Therefore, a less comfortable but more general approach than propagation is (*change*) *notification*. In contrast to propagation, other transactions or designers are solely *informed* that specific situations occured (e.g. that an object read by the designer or transaction was modified by another one), but do have to perform appropriate actions themselves. That is, (forward) recovery is not done by the database system, but is in the responsibility of a user or transaction, while the system informs the user only on the necessity of recovery. Compared to propagation, this is more flexible since the users or transactions themselves can decide upon what has to be done. Notification can be implemented by the use of *notification locks* [Hornick 87]: if an object is notification-locked, the holder of the lock is informed (notified) e.g., whenever another transaction updates the object.

As far as recovery from inconsistent states is concerned, another alternative is to *avoid* inconsistent states, i.e. to follow a “trial and error” strategy and to try to commit only those transactions that are likely to result in desired and consistent states. *Hypothetical transactions* are a possible mechanism for such a trial and error approach.

Hypothetical Transactions

Hypothetical transactions [Garza 88] were proposed (and implemented) for the object-oriented database system ORION [Kim 89] and are transactions that *always* abort. Although this sounds funny at first glance, those transactions are useful for designers who wish to experiment with “what-if” changes. Since the transactions always abort and changes therefore never get permanent, designers need not worry about a possible corruption of their database. Once the results of a hypothetical transaction are as expected, a normal transaction can be executed to install the effects permanently in the database. Hypothetical transactions could be realized by “usual” transactions (in that the transactions executes an explicit abort), though this implies unnecessary logging overhead.

Hypothetical transactions always work on copies of objects which are discarded as soon as the transaction terminates. The transaction needs read permission for the originals in order to prevent dirty read access. While redo logging is completely unnecessary, undo logging is required only for those objects flushed to permanent storage on behalf of the buffer manager.

4.7 Flow of Control

Cooperative Transaction Hierarchies

Cooperating transaction hierarchies use protocols for the specification of allowed / enforced sequences of operations of multiple cooperating transactions (recall the pattern machines mentioned above). These protocols can also be seen as describing allowed or expected flow of control. However, since protocols are defined in terms of read/write actions on simple objects, this is a very basic approach.

ConTracts

One of the design issues of ConTracts essentially was to seperate single units of work from the flow of control. Essentially, ConTract support a two-level programming approach, where the lower level consists of atomic computations (steps) and the upper level describes flow of control between steps by means of *scripts*. The specification of control flow is given in a Concurrent Pascal like style in [Reuter 91], but in principle a graphical style is also possible. In any case, constructs like sequence, branch, loop and parallel execution are provided. For the internal representation, predicate transition nets are used.

Triggers and Transactions

As proposed in [Dayal 90, Dayal 91], triggers (essentially, ECA rules) may be used to organize activities of long duration. For those events occuring in a transaction, it can additionally be specified when and how to perform the action in relation to the triggering transaction (i.e. the transaction in which the event took place). This relation then is specified by so-called *coupling modes*. In general, two possibilities exist:

1. condition evaluation and/or action execution take place in a subtransaction of the triggering transaction,
2. condition evaluation and/or action execution are performed in an own top-level transaction.

In the first case, the subtransaction can be executed immediately after the occurance of an event, or at the end of the triggering transaction, but before commit. The second case can be subdivided in the case where the triggered transaction is independent of the triggering one, or where it is commit-dependent.

4.8 Summary

In this subsection, I will summarize the proposed concepts, options and choices (see Table 1 and Table 2). These will be used in the next section for the specification of transaction primitives for design transaction management. For the understanding of these tables,

Approach	Project Structures	Long-Lived Applications	Synchronization Mechanism
XSQL	explicit	recovery TAs	locking
DAMOKLES	explicit	ACID transactions	locking
Group Transactions	2-Level Nested TAs		locking
[Kim 84]	Nested TAs	recovery TAs	locking
CAD Transactions	Multi-Level TAs	short transactions	locking predicatewise 2 PL
Cooperative Transaction Hierarchies	Nested TAs	unclear	finite automatons
Split and Join TAs	-		release of results at split
Sagas	-	recoverable sub-TAs	any
ConTracts	?	atomic steps	
NT/PV	Nested TAs		predicatewise 2PL
Version Server	-		checkout
ETM / ECA	-	-	-

Table 1: Properties of Proposals for Design Transaction Management

note that not all approaches address all relevant aspects identified for design transaction management.

In nearly all approaches, the notion of *designer*s and *groups* exist. Groups may be defined explicitly, or may be represented by (nested) transactions.

Designers and groups have sets of objects attached to them. These objects are those the designers possess or currently are working with. They may be contained in logical databases or workspaces. Both may be defined statically, or may be constructed dynamically, e.g. may contain all objects a group has checked out.

Approach	Controlled Cooperation	Consistency	Flow of Control
XSQL	checkout	not explicit	-
DAMOKLES	checkout	not explicit	-
Group Transactions	checkout	not explicit	-
[Kim 84]	checkout	not explicit	-
CAD Transactions	checkout	pre- and post conditions	-
Cooperative Transaction Hierarchies	transaction semantics	not explicit	expressable through protocols
Split and Join TAs	release before commit	not explicit	-
Sagas	-	not explicit	-
ConTracts	no control	entry and exit invariants	scripts + steps
NT/PV	no control	pre- and postconditions	-
Version Server	checkout	change propagation	-
ETM / ECA	-	triggered	triggered

Table 2: Properties of Proposals for Design Transaction Management

Authorization (i.e. read and write permission) is possible via object ownership and group membership, or by means of some nested locking protocol.

Long-lived units of work are supported in that the overall unit is broken up into smaller chunks of work which are made durable. Obviously, from an implementation viewpoint, information on active long-lived transactions and locks has to be made permanent as well. For permanence of this information, the usual internal, permanent data structures can be used.

In many approaches, Checkout/Checkin is one of the cornerstones. Checkout is simply used to copy objects from an original to a destination database. Therefore it is a useful operator whenever the global database is separated into multiple logical databases or workspaces. Additionally, checkout may be used as a primitive for cooperation (e.g. controlled sharing of intermediate results, see below).

Checkout can also be used for synchronization, whereby objects are locked at checkout and locks are released at checkin. In this case, the semantics of checkout depends on the lock modes, their attached compatibility function, and the operations that other transactions are permitted to excute on the locked objects. Additionally, the propagation of checkout to e.g. versions or subobjects depends on the corresponding data model.

Alternatively, semantics of transactions can be taken into account for synchronization purposes. In this case, the DBA or another responsible person either specifies breakpoints or correct interleavings of transactions, or invariants have to be specified for each transaction.

Recovery usually is not provided in the context of design transaction management, for that purpose short or recovery transactions are included. The use of short transactions for support of durability implies that different types of transactions have to be supported by a transaction management subsystem, one of them supporting atomic, isolated and durable transactions. These transactions might (or should?) be orthogonal to the transactions of other types (i.e. need not necessarily be subtransactions of a design transaction), thus supporting short or classical applications as well as design transactions.

Consistency and forward recovery depend on the data model and the application schema to a high degree. Anyway it seems that the database system can do very few in an automated way; hence the users or designers are in charge of performing appropriate actions.

Differences among the various concepts exist in their interrelations to other components like the data model, version support, and integrity maintenance. Nevertheless, due to the complex structure of data in design applications, it seems reasonable to support complex objects in a data model for engineering applications. The transaction management subsystem should be able to cope with complex objects, e.g. synchronize access to complex objects instead of to simple, flat objects only [Garza 88, Eckhardt 90, Herrmann 91, Geppert 92].

Additionally, most of the approaches use some notion of “version”. However, while in the well-known multi-version concurrency control techniques [Bernstein 83] versions are solely temporarily created and are not visible from “outside” the DBMS, they are supported as a first-class concept in many engineering database systems. Nevertheless, multi-version concurrency-control is argued to be useful in order to increase parallelism in design transaction management [Barghouti 91].

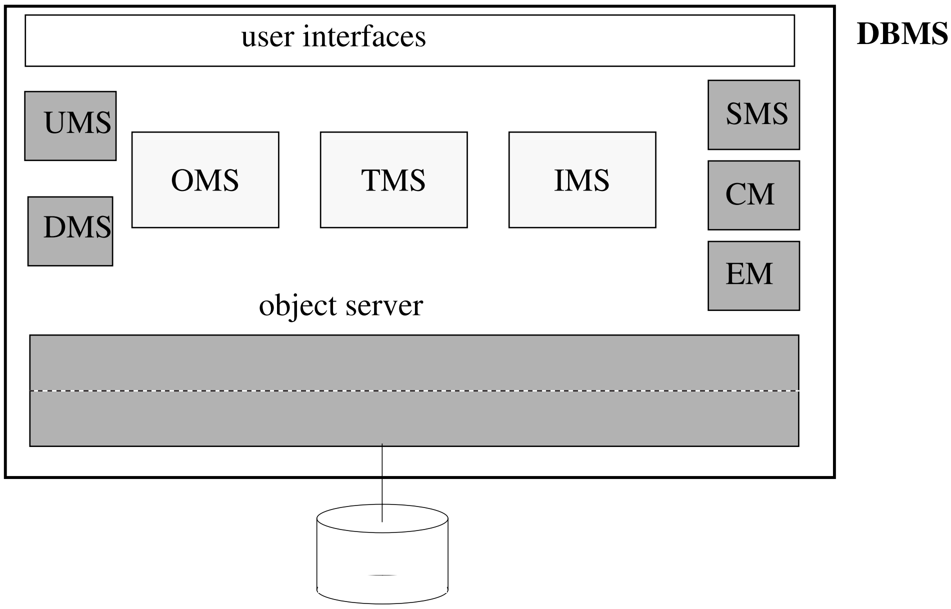
5 Primitives for Design Transaction Management

In this section, I will develop a number of primitives applicable in order to realize design transaction management. In the sequel, the set of primitives is also refereed to as *Traps*[2].

5.1 Architectural Issues

Before the primitives can be described, some details related to the architectural context have to be clarified. [Geppert 92b] describes an extension of the well-known layered DBMS architecture [Haerder 87], namely a subsystem architecture for configured or generated DBMS (e.g. KIDS [Dittrich 91]). For this general architecture, see Figure 1. Most

2.***Tra***nsaction ***P***rimitives ***S***et



= fixed = configured = implemented by DBI

OMS = object management, TMS = transaction management, IMS = integrity maintenance, UMS = user and group management, DMS = logical database manager, SMS = schema manager, CM = communication manager, EM = event manager

Figure 1 Subsystem Architecture of Generated DBMS

important is the fact that nearly all subsystems are realized on top of an object server, which realizes physical objects, physical access paths, and basic transaction management (to name the most interesting for my purposes). Thus, it is a design issue whether design transaction primitives are offered by and realized by the object server, or whether they are realized in a subsystem on top of the object server.

A principal design rule for the object server is to offer only those functionality that is commonly required by all (or at least “most”) of the DBMSs that one could desire to realize with a DBMS generator. Any application domain specific functionality naturally can use the object server (indirectly), but has to be realized in upper subsystems. Therefore, since a lot of aspects of design transaction management are application domain specific or even model-dependent, those should be realized on top of the object server. This statement is valid for features related to project structures, transaction structures, synchronization, and recovery. Since I assume that e.g. synchronization is done for data model objects (e.g., complex objects) instead of for physical objects or pages, synchronization, recovery and so on cannot be done (completely) inside the object server.

Additionally, one could expect that long-lived transactions are supported by the object server (e.g., in case of a warmstart, upper layers can ask the object server which long transactions are still active). However, as we will see below, some information on long transactions is data model or application domain specific, and thus long transactions are supported on upper layers as well. Since long transactions, on the other side, may (or probably will) access data model objects, it happens to be cleaner to deal with those kinds of transactions in the transaction management subsystem, which is realized on top of the object server. Likewise, the same holds for generic, object-independent primitives like notification or transaction statements (e.g. split). Hereby, the design rule is to assemble related primitives (i.e. primitives that alltogether contribute to a specific (sub) aspect) in one subsystem, rather than to assign them to different ones.

Furthermore, as can be seen from Figure 1, some specific subsystems realize functionality that was described for some of the concepts of the previous section. These subsystems namely are user and group management, logical database management, a generic event manager, and a consistency maintenance subsystem. Naturally, the functionality offered by these subsystems is not covered by the design transaction primitives.

5.2 Traps

In this subsection, I (informally) survey the selected set of primitives (see Table 1).

Some primitives may be parameterized, e.g. in the sense that they depend on information on the data model or desired transaction model. These parameters have to be provided at *configuration time*, and they are used to instantiate a *generic* primitive. Instantiating a generic primitive mostly results in the construction of classes, methods or functions, whose realization then depends on the configuration primitives. Naturally, these instantiated primitives require parameters when they are executed. These parameters are referred to as *runtime parameters*.

5.2.1 Project Structures

Both of the two alternatives to represent project structures are supported; however, none of them requires an extra primitive. If project structures are represented explicitly, the user manager and the logical database handler are responsible. If nested transactions are applied to model project structures, no extra primitives are necessary.

5.2.2 Long-Lived Transactions

Of course, design transaction management has to support long-lived applications. In general, I assume that a long transaction is somehow split into “smaller” units which act as recovery transactions. Therefore, like in usual transaction management, short transactions active at a crash will have to be rolled back. Additionally, at warmstart, the (design) transaction manager has to recognize which long transactions were active at the most recent crash or shutdown, and has to reinstall or recover information on these transactions (transaction control information, possibly including information on subtransactions, execution state, and so on). This information is usually extracted from the log in classical transaction

Primitive	Configuration Parameters	Runtime Parameters
PersistentTA		
checkout	• granularity • synchronization technique	• transaction id • object id(s) • original db id • design db id • operation mode
checkin	• granularity • synchronization technique	• transaction id • object id(s)
uncheckout	-	• transaction id • object id(s)
Locking	• lock modes • compatibility function • granularities • MGL protocol	grantLock: • transaction id • object id(s) • mode releaseLock: • transaction id • object id releaseLocks: • transaction id
State Automatons	permitted transition structure	validate: • transaction id • object id • operation mode
SplitTA	-	see [Pu 88]
JoinTA	-	see [Pu 88]
NotifiableTA	-	ReceiveMessage: • message identifier

Table 3: Summary of *Traps*

management; however, in order not to blow up the log information, transaction control information will be stored as persistent (physical) objects as supported by the object server.

Additionally, information on accessed information has to be persistent. If locking is used to record objects that were checked out, those locks have to be persistent. Again, the object server will be used in order to support persistent locks. Since access to persistent locks has to be fast for synchronization efficiency, locks are made persistent by means of an (extensible) hash manager index technique that has to be offered by the object server.

Otherwise, if finite state automatons are used for synchronization, like in the cooperative transaction hierarchy approach, protocols (i.e. the automaton) as well as the current state has to be made persistent.

In general, the information used by any synchronization technique that shall be applicable for design transactions has to be made persistent.

5.2.3 Client/Server Environments

Client/server environments are supported by the three primitives checkout, checkin, and uncheckout. They have the usual meaning, namely: checkout copies objects from one logical database to another, checkin copies modified objects back, and uncheckout removes objects from a database into which they were checked out. Configuration Parameters for checkout as well as for checkin are the granularity of objects to be copied and the synchronization mechanism to be applied in both databases. At runtime, checkout takes a transaction identifier, identifiers for the original and the design database, a set of objects and a operation mode as parameters.

From a technical point of view, the communication manager is invoked in order to transfer objects from one database to another.

5.2.4 Synchronization and Cooperation

In principle, one could offer any known synchronization mechanism for design transaction management as well. However, since aborts of long-lived transactions have to be avoided, timestamps-based as well as optimistic synchronization can be ommitted. Locking, on the other hand, is supported as well as the finite state automaton approach.

Locking

For the lock primitive, a variety of configuration parameters have to be specified (similar to the “normal” case):

- the set of lock modes,
- the compatibility function,
- granularities and the related lock protocol (mainly: propagation of locks).

In principle, the overall lock protocol (e.g., two-phase, strict) is also a parameter of the primitive[3]. Furthermore, note that more than one lock manager can exist in a DBMS, e.g. one for short transactions and one for design transactions (managing persistent locks). As stated above, persistent locks will in turn be realized as physical objects, and require support by physical access paths (extensible hashing).

The result of the instantiation of the lock primitive is the lock manager class, which manages the lock table and support methods to grant and release locks. The grantLock method takes a transaction identifier, a set of object identifiers, and a lock mode as runtime arguments. releaseLock takes an object identifier and a transaction identifier as arguments, while releaseLocks requires only a transaction identifier as argument. The first method releases exactly one lock, while the second one releases all the locks a transaction holds.

3.Note that the notion of “protocol” occurs in two different roles in transaction concepts: first, as a set of rules which locks have to be acquired in which order when multi-granularity locking is applied. Second, as a set of rules which locks a transaction has to acquire during its execution, and when these locks can or have to be released.

Finite State Automatons

Although I guess that finite state automatons are not practicable in the general case, they can be reasonably applicable e.g. for synchronizing checkout/checkins in a group database. The concrete automatons, naturally, can be specified at runtime (not at configuration time). Hence, at configuration time the permitted transitions have to be determined. In principle, the possible transitions are labelled with a subset of the cartesian product of the following four sets:

- transaction id, user or group name
- operation (DML, checkout, checkin, commit, abort, ...)
- object identifier,
- reaction (accept, refuse, abort,...).

The result of configurating the finite state automaton primitive is a class that permits the definition of automatons according to the specification. This class also supports a method validate, which takes a transaction identifier, an object identifier, and an operation name as input and determines whether the operation can be executed legally in the current situation, i.e., if there is a transition compatible with the input parameters and the reaction accept.

Split and Join

The last primitives for synchronization and cooperation are split and join. Both do not require any configuration parameters, their runtime parameters are similar to [Pu 88].

5.2.5 Active Mechanism

I already mentioned that a generic event management subsystem is assumed to be present in a configured DBMS. This active mechanism can be used for consistency maintenance, flow of control, and recovery. A still open design issue is the permitted structure of events. At least the execution of begin_transaction, commit, abort, DML-actions, checkout, checkin, and so on should be permitted as events. The action corresponding to the event can be any program. Furthermore, operations raise_event, activate, deactivate and so on are defined for triggers.

5.2.6 Communication / Notification

Incidentially, users or transactions will have to communicate or have to be notified about the occurence of specific situations. In contrast to ObServer, where *notification locks* are used, we apply a more general approach. Actually, notification is an application of the active mechanism. Beside the active mechanism, a primitive is used for transactions to receive messages. This primitive essentially is a class NotifiableTA which has only one method receiveMessage. This method has a message identifier as runtime parameter.

Alternatively, users (instead of transactions) can be notified. In this case, transactions are identified with designers (like in the nested transaction approach to model project structures), and the DBMS can send mail to the owner of a transaction (using mechanisms provided by UNIX).

5.3 Discussion and Further Work

In principle, the domain analysis and the primtives presented in this paper have to be evaluated through a series of constructions (of DBMSs supporting design transaction management). Especially, the set of primitives is rather small so that one may question whether it really supports broad variety of realizable concepts. However, the the proposed set is estimated to be sufficient out of the following reasons:

- Some of the primitives are quite complex and powerful and in turn comprise a variety of techniques(e.g., the locking primitive).
- To a large extent, design transaction management relies on functionality provided by other subsystems. Especially the active mechanism (event manager), though it in turn requires primitives like notification, is proposed for different purposes (consistency, (parts of) recovery, flow of control, ...).
- Some functionality of design transaction management is not very different from primitives for traditional transaction management, e.g., transaction structures, thus no extra primitives are required here.

Transaction structures and behaviour (e.g., abort and commit dependencies) are also good examples for the relevance of the specification part of configurable DBMS. Although this paper is not dedicated to the overall DBMS construction method, it is nevertheless possible to illustrate some characteristics. To that end, three levels have to be distinguished:

- specification languages (e.g., one for transaction models),
- the level of primitives, and more general, the level of reusable artifacts,
- an intermediate level that helps to map specifications to primitives. This is also the level to express constraints and information on alternative strategies.

The transaction model specification language is responsible for the definition of *transaction types*. One property of a type is its structure, i.e. which operations any of its instances may legally perform (this should be straightforward for the concepts and models surveyed above). The second aspect is transaction behaviour, essentially which dependencies to other transactions exist (abort and commit dependencies, for instance) [Chrysanthis 90]. Other important aspects are synchronization and recovery. Additionally, more options for a transaction type will be necessary: for instance, it has to be possible to specify the instances of a transaction type to be of long duration.

Thus, in addition to the definition of *Traps* for standard transaction management, the specification language for transaction models has to be extended (with respect to characterstics of design transaction management).

Mainly for the second kind of language, some additional necessary characteristics can be derived from this paper. Most important, as turned out for some parameterized primitives (like locking), those are not fully specified by their configuration parameters, but there can yet be some degree of freedom, e.g., in that various execution sequences are possible. As an example, take the various protocols for concurrency control, which determine when transactions may and have to acquire and release locks. Hence, a mechanism to restrict and constrain the correct execution sequences has to be supported on the intermediate level. Additionally, a general solution that is useful for other aspects as well (i.e., not only for transaction management) is preferrable.

6 Related Work

In contrast to the large variety of works on non-standard transaction management, little work has been done on both, a complete set of primitives as well as on a method how to construct subsystems for design transaction management. The few existing works will be surveyed in this section.

ObServer

ObServer [Hornick 87] supports locking and notification, which both can be tailored to the upper layer needs. Five lock modes exist: the Null lock, and the Read and Write lock. The latter two exist in two forms: a restricted and a non-restricted version. R-Read and R-Write behave as usual, while NR-Read allows competitors to acquire any other lock and NR-Write is compatible with an NR-Read (hence, the lock compatibility function is not symmetric).

Additionally, ObServer provides for *communication modes*, which are used for notification. Again, five modes exist: update, reading, writing, reading or writing, and no notification. The semantics of communication modes is that a lock holder is informed about the status of objects, i.e., what other transactions (request to) do with the locked object. For that matter, locks are combined with communication modes: for example, one can combine a NR-Read lock with U-Notify, which means that the lock holder desires to read the object in a non-restricted way, but that he/she wants to be informed about object changes.

Furthermore, *lock filters* can be used to restrict the five lock modes to a subset, i.e. undesired modes are excluded (e.g., for ensuring serializability). Likewise, the explicit unlocking and explicit commit of objects within a transaction execution can be permitted or forbidden.

Summarizing, ObServer offers primitives that can be used for synchronization/cooperation and for notification. The use of these primitives (and necessary add-ons) for the implementation of a model supporting transaction groups is demonstrated in [Fernandez 89]. Nevertheless, this primitives are offered on the level of physical objects. Thus, it is unclear how to include data model or application specific features. For instance, ObServer supports two granularities for locking: objects and segments, while more subtle granularities like complex objects are not provided. Additionally, it is questionable whether the set of lock modes is appropriate in all cases.

Trent

While ObServer can be regarded as a kernel, [Unland 89, Unland 90] proposes a tool kit approach for concurrency control and recovery. In this approach, there is also the notion of “transaction type”, whereby multiple transaction types may differ in their *structure* and *behaviour*. According to [Unland 91], the overall approach is based on nested transactions (cf. the “nested transaction” approach in the section on project structures). Thus, it is questionable whether models like XSQL that are not based on nested transactions can be realized easily.

In this approach, a transaction has an object pool attached to it, i.e., a collection of objects that can be accessed legally. Whenever a transaction requests access to an object, it has to copy the object from its parent into its object pool. In order to avoid inconsistencies, pools can be checkout-blocked or checkin-blocked. Checkout-blocked means that child transactions may not copy objects from the checkout-blocked pool, while child transactions cannot copy objects into checkin-blocked pools. Reason for that is to ensure two-phase behaviour.

Furthermore, locking is the prominent technique for concurrency control. In [Unland 89], two "effects" of a lock are defined: while the internal effect specifies which operations the lock holder can perform on the locked object, the external effect specifies the operations competitors still may perform legally. Both effects can be combined; some of these combinations are naturally permitted, while others are prohibited in general and further ones should only be used in special cases. The concrete supported modes are

- exclusive (as usual),
- update,
- derivation (of new versions),
- shared,
- browse (dirty read).

Beside the usual kind of locks ("transaction related locks"), object and subject related locks can be requested. Object related locks are assigned to an object permanently; they cannot be released, but only upgraded (thus modelling the evolution of objects). Subject related locks are held by applications, users, or transactions for a specific period of time. Of course, if a lock of any kind is requested, the compatibility with existing locks of all three types have to be considered.

As far as the selected set of lock modes is concerned, the same holds as for ObServer - it is questionable whether the specified modes are sufficient for all possible cases. Likewise, it is not clear how to incorporate data model dependent functionality like multi-granularity locking. Although "objects" are talked about, it is by no means clear what an object is, or how DBIs can refer to different notions of objects. Apparently, as stated in [Unland 91] states, semantics of operations and types is not taken into account, such that e.g. locking of object parts like in DAMOKLES [Abramowicz 89] (which relies on the semantics of meta types and associations) can at most be added by the DBI manually[4].

Furthermore, the overall model based on nested transactions may be too specific. As shown in the section on concepts, different models (e.g. with explicit project structures or orthogonal types of transactions) are possible, which however cannot be modelled in this approach. In general, the specification of transaction structure and behaviour should be left to the DBI, e.g. like in [Chrysanthis 90].

Unland also proposes a methodology how to construct transaction managers out of a given set of primitives [Unland 91]. In principle, components (of a transaction manager) are organized into classes and are realized as abstract data types. The development of a transaction type starts from a library of *basic components*. A subset of them is plugged together (via inheritance or aggregation) to build a *skeleton*. A skeleton is a fundamental transaction type, but is not yet executable. The enhancement of skeleton with specific semantics (whatever this is and how it is specified) leads to an executable transaction type.

4.Admittedly, how to support those (subtle) lock protocols in a configurable DBMS is not necessarily a weakness of the approach but a generally open problem.

As described in [Unland 91], the realization of such an executable transaction type is manually.

Kala

Kala [Simmel 91] is a "low-level storage management system". It provides for transaction management, and some functionality refers to design transaction management. Especially, transactions can be easily made persistent (i.e. of long duration), and by means of the supported mechanism checkout and checkin are easy to realize as well. However, from [Simmel 91] it is not clear to me how e.g. cooperation and non-serializing synchronization could be realized (both of which in turn may happen to be desired in combination with checkout/checkin).

7 Conclusion

This paper has performed a domain analysis for design transaction management. Important aspect have been identified, and concepts addressing these aspects have been surveyed. Furthermore, some primitives useful for the realization and implementation have been described. Hereby, it turns out that current approaches do not completely support realization of design transaction managers.

A domain analysis as performed in this paper is the first basic step for the development of a construction method. The next step consists of a (specification) language that can be used for describing the desired functionality of a system (e.g., a DBMS including design transactions) in a declarative and abstract way. The final step then comprises the development of the mapping from specifications into realizations, i.e., a description how specifications are transformed into an executable (sub) system.

The last two steps are actually subjects to future work. For the second step (specification) the ACTA framework [Chrysanthis 90] may happen to be useful. In this way, the final step specifies how transaction models (specificied with —possibly extended/modified— ACTA) are implemented.

References

Abramowicz 89

K. Abramowicz, R. Längle, M. Ranft, T. Raupp, S. Rehm: *DAMOKLES Systemdokumentation.* FZI Publikation 13, Forschungszentrum Informatik, Karlsruhe 1989.

Astrahan 76

M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, V. Watson: *System R: Relational Approach to Database Management*. ToDS 1:2, 1976.

Atkinson 89

M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik: *The Object-Oriented Database Manifesto.* Proceedings DOOD, Kyoto 1989.

Bancilhon 85
F. Bancilhon, W. Kim, H.F. Korth: *A Model of CAD-Transactions*. Proc. VLDB 85, Stockholm.

Barghouti 91
N.S. Barghouti, G.E. Kaiser: *Concurrency Control in Advanced Database Applications*. ACM Computing Surveys 23:3, 1991.

Bernstein 83
P.A. Bernstein, N. Goodman: *Multiversion Concurrency Control - Theory and Algorithms*. ToDS 8:4, 1983.

Bernstein 87
P.A. Bernstein, V. Hadzilacos, N. Goodman: *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

Bernstein 87b
P.A. Bernstein: *Database System Support for Software Engineering*. Proc. 9. Intl. Conference on Software Engineering, Computer Science Press.

Bancilhon 85
F. Bancilhon, W. Kim, H.F. Korth: *A Model of CAD-Transactions.* VLDB 85.

Chrysanthis 90
P.K. Chrysanthis, K. Ramamritham: *ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior.* Proc. SIGMOD 90.

Cooper 90
R. Cooper, A. Roberts, F. Wai: An Implementation of a Co-operative Locking Scheme for a Persistent Programming Language. In [Dearle 90].

Dayal 88
U. Dayal: *Active Database Management Systems*. Proc. 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, 1988.

Dayal 90
U. Dayal, M. Hsu, R. Ladin: *Organizing Long-Running Activities with Triggers and Transactions.* SIGMOD 90.

Dayal 91
U. Dayal, M. Hsu, R. Ladin: *A Generalized Transaction Model for Long-Running Activities and Active Databases*. In [Elmagarmid 91].

Dearle 90
A. Dearle, G.M. Shaw, S.B. Zdonik: Implementing Persistent Object Stores. Principles and Practice. Proc. Intl. workshop on Persistent Object Systems. Morgan Kaufmann Publishers.

Dittrich 85
K. R. Dittrich, A.M. Kotz, J.A. Mülle, P.C. Lockemann. *Datenbankunterstützung für den ingenieurwissenschaftlichen Entwurf.* Informatik-Spektrum 8, 1985.

Dittrich 86
K.R. Dittrich, W. Gotthard, P.C. Lockemann: *DAMOKLES - A Database System for Software Engineering Environments*. In R. Conradi, T.M. Didriksen, D.H. Wanvik (eds.): Advanced Programming Environments. LNCS 244, Springer 1986.

Dittrich 87a
K.R. Dittrich: *Controlled Cooperation in Engineering Databases*. DENG 87.

Dittrich 87b
K.R. Dittrich, A.M. Kotz, J.A. Muelle: *Basismechanismen fuer komplexe Konsistenzprobleme in Entwurfsdatenbanken*. Proc. BTW 87.

Dittrich 91
K.R. Dittrich, A. Geppert, V. Goebel, S. Nittel, S. Scherrer: *KIDS: A Declarative Approach to DBMS Construction*. In S. Spaccapietra (Ed.): Proc. of the Conference on Database Research in Switzerland, Lausanne 1991.

Drew 92
P. Drew, R. King, D. Heimbigner: *A Toolkit for the Incremental Implementation of Heterogeneous Database Management Systems*. VLDB Journal 1:2, 1992.

Eckhardt 90
H. Eckhardt: *Entwurfstransaktionen fuer modulare Objektsysteme.* IFB 263, Springer, 1990.

Ellis 89
C.A. Ellis, S.J. Gibbs: *Concurrency Control in Groupware Systems*. Sigmod 89.

Elmagarmid 91
A. Elmagarmid (ed.): *Special Issue on Unconventional Transaction Management*. IEEE Data Engineering 14:1, 1991.

Elmagarmid 92
A.K. Elmagarmid (ed.): *Database Transaction Models For Advanced Applications*. Morgan Kaufmann Publishers, 1992.

Fernandez 89
M.F. Fernandez, S.B. Zdonik: *Transaction Groups: A Model for Controlling Cooperative Transactions*. In [Rosenberg 89].

Garcia-Molina 87
H. Garcia-Molina, K. Salem: *Sagas.* SIGMOD 87.

Garcia-Molina 91
H. Garcia-Molina: *Modelling Long-Running Activities as Nested Sagas*. In [Elmagarmid 91].

Garza 88
J.F. Garza, W. Kim: *Transaction Management in an Object-Oriented Database System.* SIGMOD 88.

Gatziu 91
S. Gatziu, A. Geppert, K.R. Dittrich: *Integration of Active Mechanisms into an Object-Oriented Database System*. Proc. 3rd Intl. Workshop on Database Programming Languages (DBPL), Morgan Kaufmann, 1992.

Geppert 92
A. Geppert, S. Scherrer, K.R. Dittrich: *Transaction Management in CoOMS*. Technical Report ITHACA.Unizh.92.X.5#2, University of Zurich, 1992.

Geppert 92b

A. Geppert: *On the Architecture of Generated DBMS and DBMS Generators Or: Can Kids Generate DBMS?* Technical Report, Institut fuer Informatik, Universitaet Zuerich.

Gray 78

J.N. Gray: *Notes on Database Operating Systems*. In Bayer, R., R.M. Graham, G. Seegmüller (Eds.): Operating Systems. An Advanced Course. LNCS 60, Springer, 1978.

Gray 81

J.N. Gray: *The Transaction Concept: Virtues and Limitations*. VLDB 81.

Gray 92

J. Gray, A. Reuter: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

Hadzilacos 88a

T. Hadzilacos, V. Hadzilacos: *Transaction Synchronization in Object Bases*. PoDS 88.

Hadzilacos 88b

V. Hadzilacos: *A Theory of Reliability in Database Systems*. JACM 35:1, 1988.

Haerder 83

T. Haerder, A. Reuter: *Principles of Transaction-Oriented Database Recovery.* ACM Computing Surveys 15:4, 1983

Haerder 87

T. Haerder: *Realisierung von operationalen Schnittstellen*. In P.C. Lockemann, J.W. Schmidt (eds.): Datenbankhandbuch. Springer 1987.

Haskin 82

R.L. Haskin, R.A. Lorie: *On Extending the Functions of a Relational Database System.* SIGMOD 82.

Herrmann 91

U. Herrmann: *Mehrbenutzerkontrolle in Nicht-Standard-Datenbanksystemen*. IFB 265, Springer 1991.

Hornick 87

M.F. Hornick, S.B. Zdonik: *A Shared, Segmented Memory System for an Object-Oriented Database System*. ACM Transactions on Office Information Systems 5:1, 1987.

Kaiser 91

G.E. Kaiser, D.E. Perry: *Making Progress in Cooperative Transaction Models*. In [Elmagarmid 91].

Kaiser 92

G.E. Kaiser, C. Pu: *Dynamic Restructuring of Transactions*. In [Elmagarmid 92]..

Katz 84

R.H. Katz, S. Weiss: *Design Transaction Management.* Design Automation, 1984.

Katz 87

R.H. Katz, E. Chang: *Managing Change in a Computer-Aided Design Database.* VLDB 87.

Katz 90
R.H. Katz: *Toward a Unified Framework for Version Modeling in Engineering Databases*. ACM Computing Surveys 22:4, 1990.

Kim 84
W. Kim, R. Lorie, D. McNabb, W. Plouffe: *A Transaction Mechanism for Engineering Design Databases.* VLDB 84.

Kim 89
W. Kim, N. Ballou, H.-T. Chou, J.F. Garza, D. Woelk: *Features of the ORION Object-Oriented Database System*. In W. Kim, F.H. Lochovsky (Eds.): Object-Oriented Concepts, Databases, and Applications. ACM Press, New York 1989.

Klahold 85
P. Klahold, G. Schlageter, R. Unland, W. Wilkes: *A Transaction Model Supporting Complex Applications in Integrated Information Systems.* SIGMOD 85.

Korth 83
H. Korth: *Locking Primitives in a Database System*. JACM 30:1, 1983

Korth 88a
H. Korth, W. Kim, F. Bancilhon: *On Long-Duration CAD-Transactions.* Information Science, 1988.

Korth 88b
H.F. Korth, G.D. Speegle: *Formal Model of Correctness Without Serializability.* Sigmod 88.

Kotz 88
A.M. Kotz, K.R. Dittrich, J.A. Muelle: *Supporting Semantic Rules by a Generalized Event/Trigger Mechanism*. Proc. EDBT 88.

Lockemann 85
P.C. Lockemann, K.R. Dittrich, M. Adams, M. Bever, B. Ferkinghoff, W. Gotthard, A.M. Kotz, R.-P. Liedtke, B. Lueke, J.A. Muelle: *Database Requirements of Engineering Applications. An Analysis*. University of Karlsruhe, Internal Report 12/85.

Lorie 83
R. Lorie, W. Plouffe: *Complex Objects and Their Use in Design Transactions.* Engineering Design Applications, Database Week, 1983.

Moss 81
J.E.B. Moss: *Nested Transactions: An Approach to Reliable Distributed Computing.* Dissertation, MIT, 1981.

Moss 86
J.E.B. Moss, N.D. Griffeth, M.H. Graham: *Abstraction in Concurrency Control and Recovery Management*. COINS Technical Report 86-20, University of Mass. at Amherst, 1986.

Nodine 90a
M.H. Nodine, S.B. Zdonik: *Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications.* Proc. VLDB 90.

Nodine 90b
M.H. Nodine, A.H. Skarra, S.B. Zdonik: Synchronization and Recovery in Cooperative Transactions. In [Dearle 90].

Özsu 91

M.T. Oezsu, P. Valduriez: *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs (NJ) 1991.

Pu 88

C. Pu, G.E. Kaiser, N. Hutchinson: *Split-Transactions for Open-Ended Activities.* VLDB 88.

Ranft 90

M.A. Ranft, S. Rehm, K.R. Dittrich. *How to Share Work on Shared Objects in Design Databases.* DENG 90.

Rehm 88

S. Rehm, T. Raupp, M. Ranft, R. Längle, M. Härtig, W. Gotthard, K.R. Dittrich, K. Abramowicz: *Support for Design Processes in a Structurally Object-Oriented Database System.* In K. R. Dittrich (Ed.): Advances in Object-Oriented Database Systems, LNCS 334, 1988.

Rosenberg 89

J. Rosenberg, D. Koch: *Persistent Object Systems*, Proceedings of the 3. Intl. Workshop.

Reuter 91

A. Reuter, H. Waechter: *The ConTract Model*. In Elmagarmid 91.

Schwarz 84

P.M. Schwarz, A.Z. Spector: *Synchronizing Shared Abstract Data Types*. ACM ToCS 2:3.

Simmel 91

S.S. Simmel, I. Godard: *The Kala Basket*. Proc. OOPSLA 91.

Skarra 89

Skarra, A., S.B. Zdonik. *Concurrency Control and Object-Oriented Databases.* In W. Kim, F.H. Lochovsky (Eds.) Object-Oriented Concepts, Databases, and Applications. ACM Press, 1989.

Speegle 90

H.F. Speegle, G.D. Korth: *Long Duration Projects in Software Desing Projects*. DENG90.

Unland 89

R. Unland: *A Flexible and Adaptable Toolkit Approach for Concurrency Control in Non Standard Database Systems.* Informatik Berichte 88, Fernuniversität Hagen, 1989.

Unland 90

R. Unland: *A Flexible and Adaptable Tool Kit Approach For Concurrency Control in Non Standard Database Systems*. Proc. ICDT 90.

Unland 91

R. Unland, G. Schlageter: *A Flexible and Adaptable Tool Kit Approach for Transaction Management in Non Standard Database Systems*. Informatik Berichte 109, Fernuniversitaet Hagen, June 1991.

Wächter 90

H. Wächter, A. Reuter: *Grundkonzepte und Realisierungsstrategien des ConTract-Modells*. Informatik Forschung und Entwicklung 5:4, 1990.

Wächter 92

H. Waechter, A. Reuter: *The ConTract Model*. In [Elmagarmid 92].

Wedekind 89
H. Wedekind: *Eine logische Analyse des Verhältnisses von Anwendungs- und Datenbanksystemen*. In T. Härder (ed.): Datenbansysteme in Büro, Technik und Wissenschaft (BTW), IFB 204, Springer 1989.

Weikum 91
G. Weikum: *Principles and Realization Strategies of Multilevel Transaction Management*. ToDS 16:1, 1991.