Meckenstock, Axel; Unland, Rainer; Zimmer, Detlef

Working Paper

Controlling cooperation and recovery in nested transactions

Arbeitsberichte des Instituts für Wirtschaftsinformatik, No. 32

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

Suggested Citation: Meckenstock, Axel; Unland, Rainer; Zimmer, Detlef (1994) : Controlling cooperation and recovery in nested transactions, Arbeitsberichte des Instituts für Wirtschaftsinformatik, No. 32, Westfälische Wilhelms-Universität Münster, Institut für Wirtschaftsinformatik, Münster

This Version is available at:
https://hdl.handle.net/10419/59322

Arbeitsberichte des Instituts für Wirtschaftsinformatik
Herausgeber: Prof. Dr. J. Becker, Prof. Dr. H. L. Grob, Prof. Dr. K. Kurbel,
Prof. Dr. U. Müller-Funk, Prof. Dr. R. Unland, Prof. Dr. G. Vossen

Arbeitsbericht Nr. 32

Controlling Cooperation and Recovery in Nested Transactions

Axel Meckenstock*, Rainer Unland,

Detlef Zimmer*

*CADLAB, Bahnhofstr. 32, 33102 Paderborn

Institut für Wirtschaftsinformatik der Westfälischen Wilhelms-Universität Münster,
Grevener Str. 91, 48159 Münster, Tel. (0251) 83-9750, Fax (0251) 83-9754
März 1994

Contents

Abstract

Recovery is a hard problem in environments where transactions perform work in a cooperative style (e.g., design environments). We propose concepts to control cooperation and recovery within nested transaction hierarchies. By allowing different nodes to run different protocols, we can build so-called *recovery spheres* with well-defined properties. We characterize those properties and illustrate them by examples from design environments.

1 Introduction

Research work on recovery has emerged from techniques for simple transactions [HR83, BHG87] to techniques which can be applied within nested transactions [Mos85,HR87,WS92]. Most proposed algorithms for nested transactions are based on the *ACID*-properties [HR83] (*ACID* stands for *Atomicity*, *Consistency*, *Isolation* and *Durability*).

However, non-standard applications like design (e.g., CAD, CASE or CACE[1)]) strongly require a relaxation of the *ACID*-paradigm. These applications often need cooperation instead of isolation and partial recovery instead of atomicity. Conventional protocols like strict two-phase locking can only be used in special cases. Here, we have a typical trade-off: the more cooperation we allow, the more difficult is recovery.

To support advanced applications, we propose a nested transaction model which is based on a heterogeneous transaction hierarchy. Within this hierarchy, we can support different protocols for concurrency control and recovery, which allows us to adapt the hierarchy to the individual needs of applications. More specifically, we can build so called *recovery spheres*, which are a means to control cooperation and to limit propagation in case of cascading rollback. We can identify different kinds of recovery spheres depending on how a transaction has to interact with its environment.

In section 2 we present an application scenario for design environments which will be used as an example scenario in the rest of the paper. Then we describe two nested transaction paradigms, the conventional paradigm as proposed by Moss [Mos85] and the transaction toolkit approach [US92]. In section 4, we describe recovery mechanisms based on the toolkit approach, sketch an algorithm for handling cooperation and recovery and introduce different kinds of recovery spheres. Section 5 focuses on related work while section 6 gives a conclusion.

2 Application Scenario

In this section, we consider an application scenario for design transactions (in the following called DTs). We choose a software design environment, but we think the results can be applied to other areas as well.

---

1) *computer-aided design*, *software engineering* or *concurrent engineering*

Typically, design environments are structured hierarchically. If we regard the whole project as a DT, we can model all the activities and subprojects as child-DTs[2)] which possibly have further children. Thus, we get a nested DT hierarchy where the DTs can be regarded as workspaces.

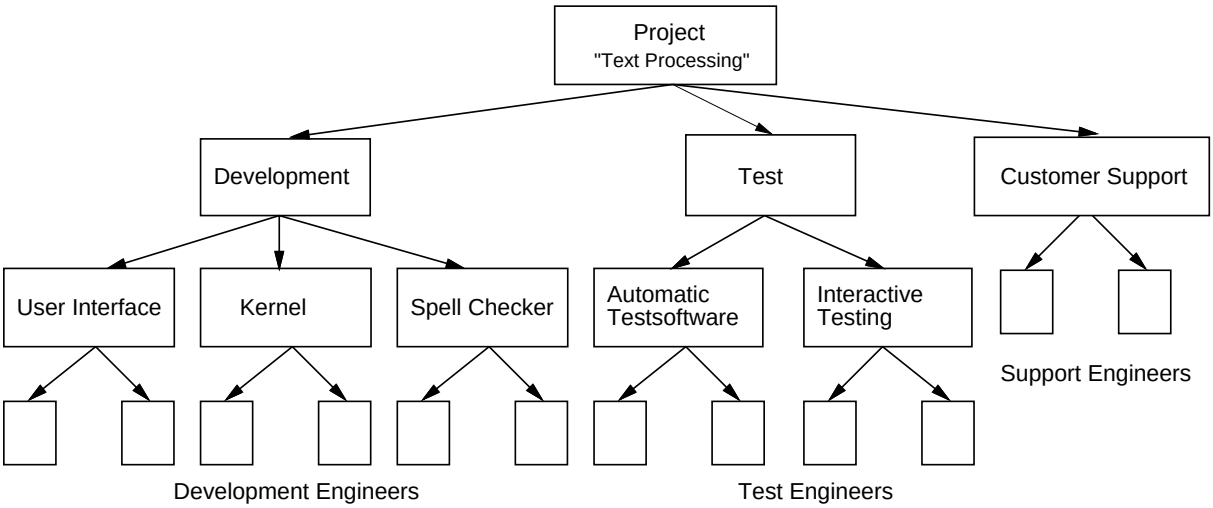

Figure 1: Example Project Hierarchy

As an example, consider the development of a Text Processing system (Fig. 1). First we can divide the project into Development, Test and Customer Support. The Development consists of the subprojects User Interface, Kernel Text System and Spell Checker. Each of these subprojects is realized by several developers who perform their work by executing tools (e.g., compiler or debugger) on objects within a DT. The Test is divided into a subproject for implementing Automatic Testsoftware and a subproject for Interactive Testing.

From this scenario, we can induce some requirements for handling DTs:

- Some DTs are of long duration, e.g., the development of the Spell Checker. Atomicity is not adequate since a complete rollback of such a DT would cause too much work to be lost.

- Some DTs have to interact with each other; for example, it is necessary to integrate the Spell Checker, the Kernel, and the User Interface. Traditional concurrency control protocols like strict two-phase locking are too restrictive since they prohibit the exchange of objects (e.g., module interfaces) before EOT. Such an interaction may cause cascading rollback since uncommitted information is spread between transactions.

---

2) Our terminology uses the words child, parent, sibling, ancestor and descendant in the obvious meaning.

– At some places in the hierarchy, we have strong requirements with respect to consistency. For example, it is important to release only tested software to the Customer Support. Moreover, a cascading rollback of the Customer Support DT would be intolerable. On the other hand, there are places with lower consistency requirements. For example, within the User Interface subproject it should be possible to exchange objects between developers without major restrictions.

We can deal with those contradicting requirements by supporting different protocols within the DT hierarchy. Thus we get a configurable hierarchy which guarantees different levels of consistency and recovery support.

3 Nested Transaction Concepts

The **nested transaction paradigm** has its roots with the spheres of control of Davies [Dav78] and became well-known by Moss' approach [Mos85]. The main idea is to give transactions a tree-like structure by allowing them to start child transactions. To the outside, a nested transaction looks like a conventional flat transaction, but its inner structure offers some advantages. First, nested transactions support a modular design. Second, they permit parallelism within a transaction (e.g., parallel execution of sibling transactions). Third, they provide finer recovery units by aborting only child transactions instead of complete transactions.

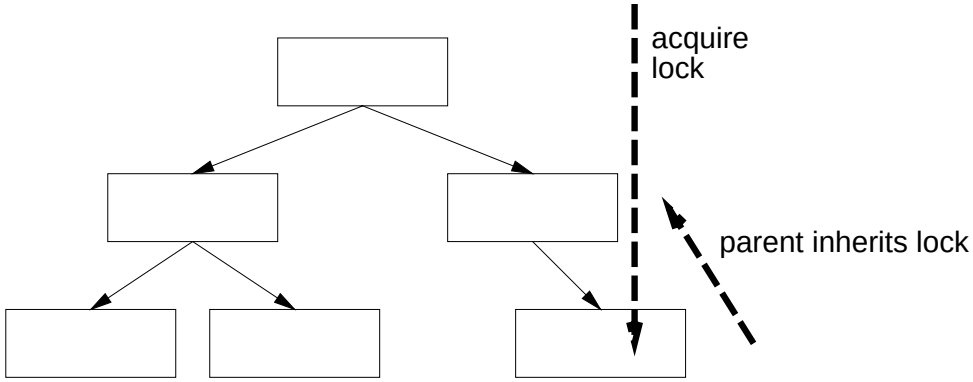

Figure 2: Nested Transactions with Upward Inheritance

The classical model for nested transactions, *closed nested transactions* (Fig. 2), is based on the notion of *upward inheritance* of locks[3)]. A transaction can acquire a lock on an object if all

[3)] We do not consider other concurrency control techniques like timestamping here.

conflicting locks (if any) are held by ancestors of the transaction. If a non-root transaction commits, its locks will be upward inherited by its parent. If the root transaction commits, the locks will be released. In Moss' original concept, only leaf transactions are allowed to perform work on objects. To remove this restriction requires additional parent-child synchronization, e.g., by using retention locks [HR93] or implicit child transactions [US92]. The approach of upward inheritance relies on the strict two-phase locking protocol, i.e., it cannot be applied to arbitrary concurrency control protocols [US92].

The **transaction toolkit approach** [US92,MUZ94] (Fig. 3) is also based on the closed nested transaction model. But the main goal of the toolkit is to support different protocols within a transaction hierarchy. This permits the definition of transaction hierarchies which are especially adapted to the individual needs of the application.

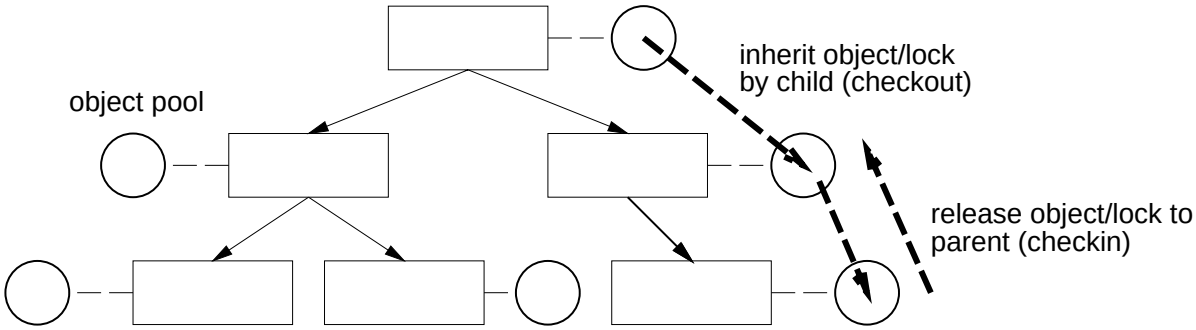

Figure 3: Nested Transactions with Downward Inheritance

In the toolkit, every transaction in the hierarchy has an object pool which (logically, not necessarily physically) contains those objects that are currently locked by the transaction. The root transaction has direct access to the database. In contrast to the classical concept, locks are inherited in *downward* direction. This means that a child transaction can acquire a lock resp. an object only if its parent already has acquired that lock/object. If not, the lock/object must be acquired step by step in downward direction from the higher levels of the hierarchy. When releasing a lock/object, the process works in the upward direction. Thus a *stepwise transfer* of locks resp. objects is performed. This mechanism corresponds to the *checkout/checkin*-paradigm often used in design environments.

The main advantage of stepwise transfer is the possibility to apply different concurrency control procotols at different nodes of the tree (Fig. 4). Each transaction defines the protocol to be used by its children for accessing its object pool. If, for example, a transaction requires a two-phase lock protocol for its object pool, its children may use strict or simple two-phase locking, but not an optimistic protocol to access the pool. Thus we get a *two stage control*

sphere: The protocol defined by a parent is only relevant for its children. The children may apply other protocols on their own object pools etc.

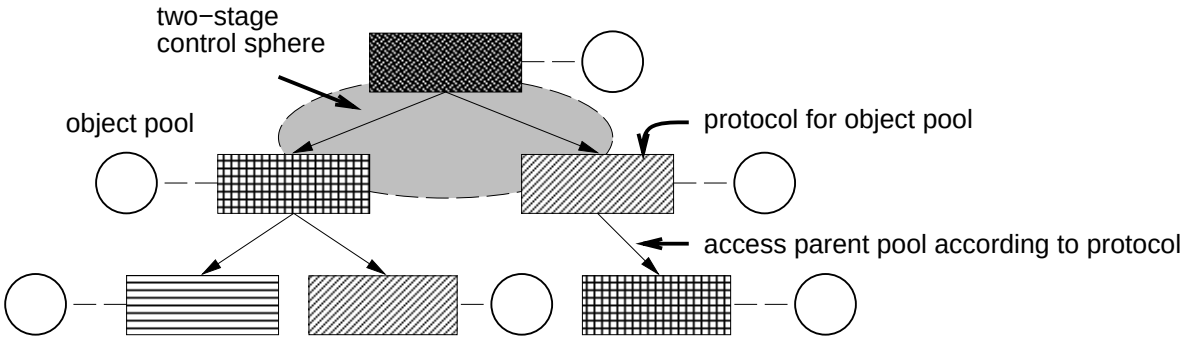

Figure 4: Heterogeneous Protocols within a Transaction Hierarchy

Cooperation between transactions is achieved by transferring objects along the hierarchy in a stepwise manner. While this may induce some overhead, it guarantees that all the protocols defined by transactions are obeyed[4]. This is a very important precondition for controlling cooperation and recovery in a nested transaction hierarchy as will be shown in the next section.

> As an example, consider a developer of the User Interface component. If he wants to pass on an object to a Kernel developer, he has to checkin this object to the User Interface DT and then to the Development DT. The Kernel developer has to check out the object from the Development DT through the Kernel DT. If the protocol of one of these DTs does not allow the transfer, the cooperation cannot be established.

4 Recovery for Nested Transactions

In this section, we look at the recovery aspects in a nested DT hierarchy. We base our discussion on the toolkit approach with downward inheritance and stepwise transfer because this approach gives us the possibility to define different protocols for the nodes of the DT tree.

Recovery is always caused by a failure. A typical classification of failures distinguishes between *transaction failures*, *system (site) failures*, *media failures* and *communication failures*. We assume that the underlying database system for storing design objects already supports recovery for conventional (short) transactions and thereby masks most of the failures. Thus,

---

[4] At the user interface, there may be high-level operations for cooperation that hide the stepwise transfer.

we restrict ourselves to logical failures within a DT, which are, for example, caused by an exception handler of an application or by a synchronization protocol.

For concurrency control, we assume a locking protocol using conventional lock modes like *shared* and *exclusive*. When checking out an object, a DT acquires a lock and copies the object to its object pool. When checking in an object, it releases the lock and removes its copy. We do not allow parallel modifications of the same object by several DTs and do not discuss any versioning.

We can distinguish between two kinds of DTs, *Service DTs* which only serve as a database for their children and do not perform real work on objects, and *Working DTs* which perform work on objects. If a non-leaf DT acts as a Working DT, we assume a mechanism for handling parent-child synchronization (e.g., by starting an implicit child DT).

The Recovery Process

Recovery is always initiated by a certain *recovery event*, e.g., an application error. We call this recovery event a *primary recovery event*. When such an event is detected, a certain *recovery action* is performed.

When a DT in the hierarchy executes a recovery action, other DTs may get involved too, i.e., they have to do some recovery action as well because they are in some way *dependent* on the first DT. In this context, we talk of *secondary recovery events* that lead to *cascading recovery*[5]. While we cannot prevent primary recovery events, we should try to reduce secondary recovery events. In general, we can distinguish two ways to deal with those events: an *optimistic approach*, where we allow secondary recovery events to occur and therefore have to accept cascading recovery, and a *pessimistic approach*, where we try to prevent secondary recovery actions.

In the conventional model transactions either *abort* or *commit* (according to the *atomicity* principle). As noted above, this is too restrictive for DTs. Thus, we also allow a DT to abort or commit parts of its work. A DT can *selectively abort* a part of its work by rolling back the modifications on some objects[6]. We call this *rollback* of objects. A DT can *selectively commit* a part of its work by checking in some objects and giving up the right to rollback its changes (although the DT itself can still abort). We call this of objects.

---

[5] We use this term instead of *cascading rollback* since a rollback is only *one* possible recovery action
[6] We can also imagine advanced recovery techniques like compensation or forward recovery here.

As an example, let us assume that a Kernel developer has passed on a module interface to the User Interface subproject and now recognizes an error. He will rollback his changes on the module interface (a total abort of the developer's DT normally would not be acceptable). If the User Interface subproject has used this interface, it has to perform some recovery actions as well, possibly also on other modules.

Dependencies

Recovery is influenced by dependencies between DTs, which can be classified as follows (Fig.5):

– **parent-child dependencies**

The principle of nesting implies that we get dependencies between a parent and its children. In the classical model, a child is always *weak-abort-dependent* [CR92] on its parent, i.e., when the parent aborts, all its children are aborted too. A commit of a child and the durability of its results are always subject to the commit of its parent. A parent can be *abort-dependent* on some of its children (called *vital* [BOH+92]), i.e., it will abort when one of its vital children aborts.

– **reads-from dependencies**

Whenever a DT (say DT_a) modifies an object, this object is first in an *uncommitted* state, i.e. it is still subject to a possible rollback. The changes are committed only when DT_a releases this object or commits completely. When other DTs read an uncommitted object they become *reads-from dependent* on DT_a. This has two effects: First, if DT_a executes a recovery action, the dependent DTs have to execute an appropriate recovery action as well (cascading recovery). Second, the dependent DTs cannot commit as long as the object is in an uncommitted state. This ensures that when DT_a rolls back the object, a correct recovery can occur (the property of *recoverability* [BHG87][7]).

The parent-child dependencies are already known from the classical nested transaction model. The reads-from dependencies do not occur there because the two-phase locking protocol must be strict. As soon as we allow DTs to cooperate we have to cope with reads-from dependencies.

---

[7]) If we allow non-two-phase locking protocols, this condition may lead to cycles, i.e., no DT can commit before the other one (deadlock)

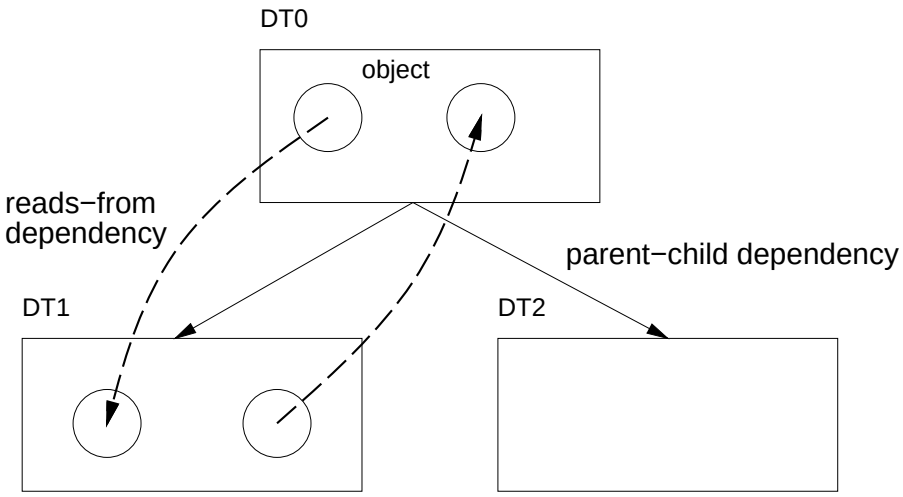

Figure 5: Dependencies between DTs

Service DTs are a special case: since they only serve as a database for their children (as a medium for transferring objects in a stepwise manner), we do not want a Service DT to abort due to reads-from dependencies on other DTs.

Cooperation and Recovery Algorithm

To make the above description more precise, we now sketch an algorithm for performing cooperation and recovery within nested transactions.

We model cooperation by transferring objects along the hierarchy using the operations *checkin* and *checkout*. We model recovery by describing the state of an object and the reads-from dependencies between DTs.

Whenever a DT modifies an object, we assign to the DT a so-called *decide-right* [MUZ94] for this modification on the object. This means that the DT has the right to decide whether it commits or aborts its modification[8)]. The DT holds the *decide-right* until it comes to its decision, even if it has transferred the object to another DT in the meantime. The DT can commit its modifications on an object either by committing completely or by releasing the object (selective commit). The DT can abort its modifications on an object either by aborting completely or by rolling back the modifications on the object. If several DTs have modified an object one after the other, there may be an ordered list of *decide-rights* on the object (even for the same DT). When an object is transferred by *checkin* or *checkout*, it takes all the *decide-rights* with it. When a DT releases an object on which it has a *decide-right*, the *decide-right* is

---

8) This is similar to the notion of *responsibility* in the ACTA framework [CR92].

inherited by the parent (or removed if the DT is the root). When a DT rolls back an object on which it has a *decide-right*, the *decide-right* is removed.

The *decide-right* gives a DT the possibility to recognize whether an object it works on can still be subject to a rollback. We will use this in the following to control cooperation and recovery. When an object with a *decide-right* is checked out or in, we establish a reads-from dependency which is used in the case of recovery to identify the way the object has already covered within the hierarchy.

To describe the algorithm, we use the following notation:

- There is a predicate *dec*(*x*, *m*, DT), if DT owns the *decide-right* on object *x* for a certain modification *m*. If a DT has made several modifications without any intervening modification by another DT these modifications can be combined to one. The predicates are ordered by a relation "<".

- There is a relation $DT_a \xrightarrow{x,m} DT_b$, if DT_b is reads-from dependent on DT_a because of the modification *m* on object *x*. When we add a "*" we mean the reflexive and transitive closure.

- The parent of a DT is called p(DT).

Since parent-child dependencies are fairly simple and are only relevant for complete aborts of DTs, we restrict the algorithm to reads-from dependencies. We informally sketch the main steps for the operations *checkout*, *checkin*, *modify*, *release* and *rollback*. We also describe an operation *recover* which models the reaction of a DT that is affected by recovery. The algorithm does not treat the dependencies between objects within a Working DT (caused, e.g., by operations or semantic dependencies).

DT.checkout(x) (DT checks out object *x* from its parent.)

forall DT_d: *dec*(x, m, DT_d)

do add (p(DT) $\xrightarrow{x,m}$ DT) (establish the reads-from dependencies)

DT.checkin(x) (DT checks in object *x* to its parent.)

(There must be a *dec* on *x* from the DT or its descendants. The object can only be checked in if no child has checked out the object for modification.)

forall DT_d : *dec*(x, m, DT_d)

do add (DT $\xrightarrow{x,m}$ p(DT)) (establish the reads-from dependencies)

DT.modify(x) (DT modifies object *x* by modification *m*.)

add *dec*(x, m, DT) (add a *dec*-predicate for this modification)

DT.release(x) (DT releases object *x* with *dec*(x, m, DT).)

(Here, *m* is the last modification the DT made on the object. Earlier modifications by the same DT are released implicitly, too.)

if $\exists$ DT_d not ancestor of DT: (*dec*(x, m_1, DT_d) < *dec*(x, m, DT))

then error ("release of object would violate recoverability")

if x was not already checked in

then DT.checkin(x)

forall object pools containing a copy of x with *dec*(x, m, DT)

do change *dec*(x, m, DT) to *dec*(x, m, p(DT)) (the parent inherits the *dec*.)

remove the corresponding reads-from dependency

if DT is the root DT

then remove the *dec*-predicate and the corresponding reads-from dependencies for all DTs.

DT.rollback(x, m) (DT rolls back the modification *m* on object *x* with *dec*(x, m, DT).)

forall DT_a: (DT $\xrightarrow{x,m^*}$ DT_a) **do** begin

DT_a.recover(x) ((cascading) recovery for (transitively) reads-from dependent DTs)

remove the corresponding reads-from dependency

remove *dec*(x, m, DT)

end

DT.recover(x, m) (DT performs recovery because of the modification *m* on object *x*.)

rollback modification *m* on *x* in the pool of DT (Here we expect the existence of *before images* or other recovery information.)

if there is a modification m_1 occurring after *m*

then DT.rollback(x, m_1) (later modifications on the same object have to be rolled back, too.)

if there is a modification m_2 on object *y* relying on the modifications on *x*

then DT.rollback(y, m_2) (other objects may be affected, too.)

The algorithm only demonstrates the basic principles of cooperation and recovery. It becomes more complex if we remove the restrictions, e.g. by allowing more flexible lock modes.

The algorithm works in a stepwise manner: a DT can only have direct dependencies on its parent or its children. All other dependencies between arbitrary DTs are transitive. This gives us the possibility to control dependencies by controlling the interaction between a parent and its children.

We illustrate the algorithm by an example (Fig. 6.). DT1 checks out object x from DT0, modifies it and checks it in. Then DT3 checks out the object through DT2, reads and modifies it. Thus, DT3 has read an uncommitted object, expressed by the *dec*(DT1) predicate, and must be aware of a possible cascading recovery. If DT1 releases the object, its *dec* will be inherited by DT0. If DT1 rolls back the object, the changes have to be rolled back in the pools of DT0, DT2 and DT3, too. Since DT3 is a Working DT, its own modifications and maybe some of its work on other objects will have to be rolled back as well.

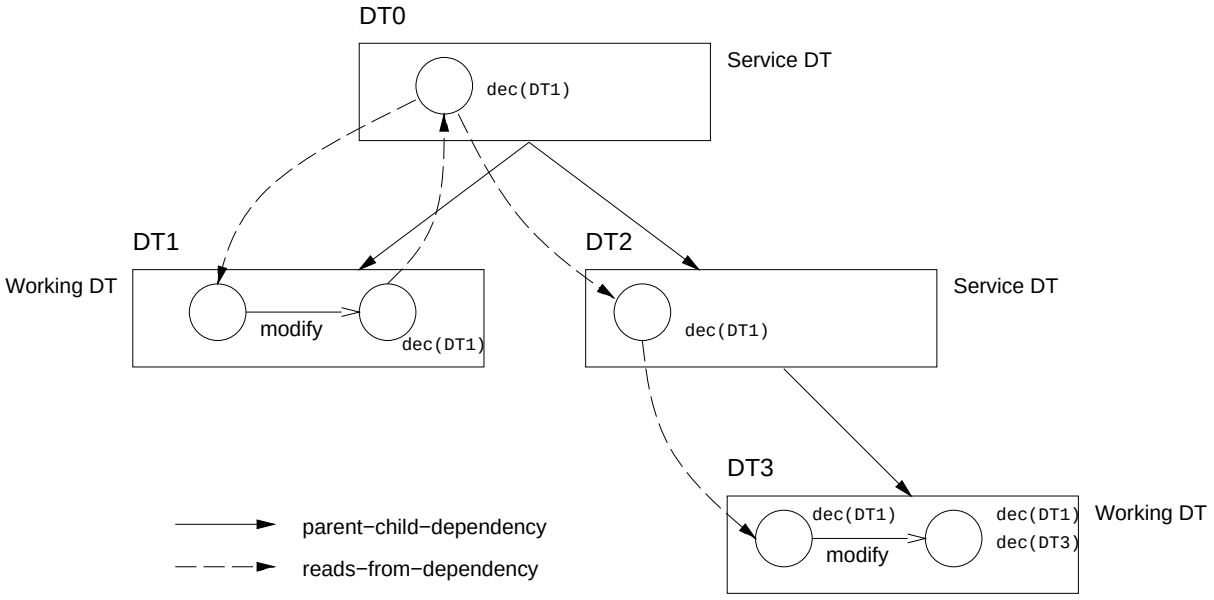

Figure 6: Example for the Algorithm

Controlling Cooperation and Recovery

The above algorithm can lead to a complex dependency graph if there are many dependencies within a hierarchy. Our goal is to regulate the structure of such dependencies in order to prevent arbitrary cascading recovery.

There are three ways to influence dependencies:

- Declare a child as vital or non-vital. In the non-vital case, the parent will not become abort-dependent on the child.

- Allow or disallow reads-from dependencies from the child to the parent. If we want to disallow this, we have to prevent the child from checking in objects on which the child or some of its descendants have a *dec*. This means that the child must either run a strict two-phase locking protocol or must release objects on checkin. We say that the child is *checkin-safe*.

- Allow or disallow reads-from dependencies from the parent to a child. If we want to disallow this, we have to prevent the child from checking out objects on which a descendant of an ancestor has a *dec*[9)]. A *dec* from an ancestor is not critical. Due to the

---

9) The child can *browse* such objects. A browse has lower consistency requirements and therefore will not cause reads-from dependencies.

nesting the whole object pool of the child is dependent on its ancestors anyway. We say that the child is *checkout-safe*.

Thus, we get eight (theoretical) combinations of recovery properties of DTs. If we disallow dependencies, we follow a pessimistic approach, i.e., we prevent cascading recovery. Such an approach limits cooperation between DTs. Otherwise we follow an optimistic approach. This gives us more freedom for cooperation between DTs for the price of cascading recovery. Of course, there are other combinations in between.

As an example for a checkin-safe DT consider the Test DT. If it transfers an object to the Customer Support DT, it must guarantee that it will not rollback this object, because this is not acceptable for the Customer Support DT. On the other hand, the Customer Support DT should be defined as checkout-safe.

Recovery Spheres

The advantage of the principle of stepwise transfer is that there may be transitive but no *direct* dependencies between two arbitrary DTs (which are not in parent-child relation). All the dependencies are local within a two stage control sphere which is a noticeable advantage in comparison to traditional recovery concepts.

The two stage control sphere gives us the possibility to adapt recovery to the special needs of each DT. In the toolkit approach, a DT has to use a protocol as required by its parent. Because of the principle of stepwise transfer, this protocol can act as a barrier. It defines the information flow between the DT and its descendants on the one side and all other DTs on the other side. We call the DT together with all its descendants the *recovery sphere* (RS) of the DT (Fig. 7). Of course, RSs can be nested.

A protocol for a DT can, e.g., have the following properties:

- The DT may be non-vital. In this case, its parent is safe against a rollback within the DT's RS (provided, there are no other dependencies).

- The DT may be checkout-safe. In this case, the DT's RS is safe against reads-from dependencies on other DTs.

- The DT may be checkin-safe. In this case, all other DTs are safe against reads-from dependencies on the DT's RS.

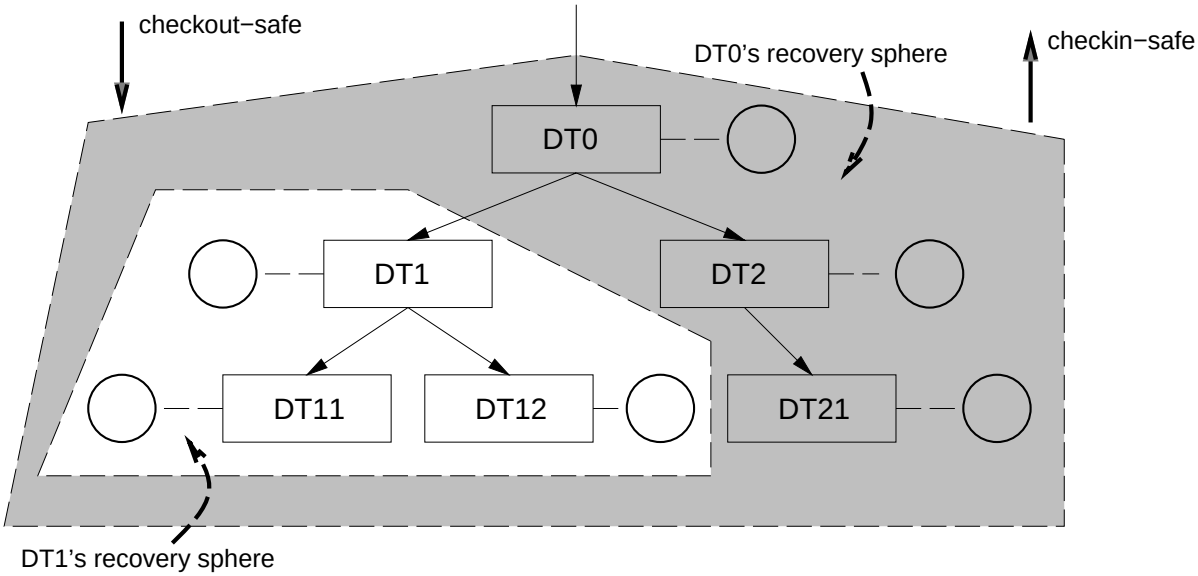

Figure 7: Recovery Spheres

The concept of RSs gives us the possibility to define special environments and their relationship to outer or inner environments. By specifying protocols, we define the interaction of the RS with the outer DTs and the inner DTs.

> For example, the Development's RS defines the relation of all the development DTs to the DTs outside the development. This RS can, e.g., be defined as checkin-safe, so that no uncommitted objects may leave the development. Within the Development's RS we have several other RSs, e.g., the Spell Checker's RS within which arbitrary cooperation may be allowed.

Extensions

In the previous discussion, we used a simplified model. We distinguished between objects with or without *decide-right* and we defined rules for *each* object a DT accesses. For a realistic environment, this model is too restrictive.

A possible extension is to define several consistency states for objects. This is typical for design environments where an object is not just consistent or not consistent, but goes through a number of states which eventually lead to the required consistency level. A typical example is a software module which may be uncompiled, compiled, passed through a module test, tested together with other modules etc. If we have a mechanism to describe the consistency state of an object (e.g., by *features* [Kae91]), we can use this to define an RS on a more specific level. More precisely, this means that we can define

– what consistency state an object must have when it is checked out by a DT.

– what consistency state an object must have when it is checked in by a DT.

> As an example, consider the Test DT. It should only checkout objects which have been compiled and passed a module test, and it should only checkin objects for which a certain test suite has been run successfully.

Another extension is to distinguish between several object types. For some types of objects, there are stronger consistency requirements than for other types. Thus, a DT may refuse to checkout or checkin objects of a certain type with a certain consistency state, but it will allow to checkout or checkin objects of another type.

> For example, consider the Test DT. It should only checkin objects for which the test suite was run successfully. But if it wants to transmit a test protocol to the Development DT, this should be allowed. Thus, the rules should be defined differently for different object types.

In our model, we can release or rollback single objects. Since objects are often dependent on other objects, this can cause inconsistencies. The classical transaction model deals with this problem by aborting complete transactions and implicitly assuming that transactions are consistency-preserving units. Thus, to permit selective recovery, we have to consider the dependencies between objects. These dependencies may be caused by operations on objects, by the data schema (e.g., objects containing other objects) or by the application semantics. We have to refer the reader to forthcoming results here.

> Consider a developer checking out an interface and using some of its functions in another module. If the interface is rolled back, a rollback or a correction of the changes in the other module has to be performed.

Summary

We presented a number of recovery protocols which can be independently applied for different nodes of a nested transaction:

– DTs can be defined as vital or non-vital.
– DTs can be checkin-safe and/or checkout-safe.
– For checking objects in and out, certain consistency states can be defined.
– The protocols can distinguish between different types of objects.

Because of the principle of stepwise transfer, we can define such features for whole environments, called recovery spheres. This allows us to gain control over the behaviour of certain environments within a DT hierarchy. Thereby, we can avoid arbitrary information flow between DTs and thus avoid arbitrary dependencies that could lead to cascading recovery. Another advantage of this locality occurs in distributed environments: a node in the DT hierarchy can autonomously decide about its protocols and has to communicate directly only with its parent and children.

5 Related Work

The problem of recovery within cooperative environments is dealt with rarely in literature. The work of [HR87,HR93] is based on the classical nested transaction model. It does not allow for configuration of the nodes in a transaction hierarchy and for building recovery spheres.

In [NRZ92], selective recovery is proposed based on individual operations. Dependencies between operations within a transaction and operations in different transactions are recorded in a log. When cascading recovery occurs, a transaction can react in several ways (e.g., reread invalid versions or compensate operations). Again, there is no way to build recovery spheres with defined properties.

The concept of *Split Transactions* [PK92] allows transactions to selectively commit or abort parts of their work by splitting off this work as a separate transaction. The model relies on serializability and does not deal with the problem of cascading recovery caused by cooperation between transactions.

In some work, cooperation is achieved by building special relationships (e.g., a usage, delegation or negotiate relationship as described in [RMH+94a]). Since arbitrary transactions can cooperate in such a way, it is very difficult to control the consequences of recovery. Therefore, we prefer the more restrictive approach of stepwise transfer.

Many papers (e.g., [KLS90,WR92,WS92]) suggest compensation as a means of recovery in cooperative environments. Compensation could be integrated into our approach too: it allows to pass on objects with *decide-right* even if DTs are checkout-safe or checkin-safe. Compensation can be performed without causing cascading recovery if other DTs are restricted to execute only those operations which are commutative to the compensating operation. Thus, we either must define restrictive rules for compensation to be successful or

must accept cascading compensations [WR92]. In any case, compensation is restricted to special environments and therefore not generally applicable.

6 Conclusion and Future Work

In this paper, we have described how recovery problems can be handled in a cooperative transaction environment. We have presented a nested transaction approach with downward inheritance and stepwise transfer. This gives us the possibility to define different protocols for the nodes of the hierarchy and to control the cooperation and recovery behaviour of subhierarchies (called recovery spheres).

We have shown several primitives to define recovery protocols for the nodes. These primitives already allow to describe a lot of different kinds of recovery spheres. There are some other facilities which should be included as well:

- Compensation should be integrated because it permits more application-specific recovery.

- Other recovery mechanisms should be integrated. E.g., it should be possible to reread invalidated objects or make automatic or manual corrections instead of rolling back the changes.

- Event-Trigger mechanisms should be considered. On the one hand, they could be used as a mechanism to define application-specific recovery actions. On the other hand, it has to be prevented that by such mechanisms information can flow within the hierarchy without considering the protocols of the DTs.

- Additional dependencies (e.g., data schema dependencies) should be handled.

The implementation of the proposed mechanisms on the base of our transaction toolkit prototype and the investigation of the above facilities are topics of future work.

Bibliography

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

[BOH92] A. Buchmann, M.T. Özsu, M. Hornick, D. Georgakopoulos, and F.M. Manola. A Transaction Model for Active Distributed Object Systems. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 123 - 158. Morgan Kaufmann, 1992.

[CR92] P.K. Chrysanthis and K. Ramamritham. ACTA: The SAGA Continues. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 349 - 397. Morgan Kaufmann, 1992.

[Dav78] C.T. Davies,Jr. Data processing spheres of control. *IBM Systems Journal*, pages 179 - 199, 1978.

[HR83] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287 - 317, December 1983.

[HR87] T. Härder and K. Rothermel. Concepts for Transaction Recovery in Nested Transactions. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 239 - 248, 1987.

[HR93] T. Härder and K. Rothermel. Concurrency Control Issues in Nested Transactions. *VLDB Journal*, 2(1):39 - 74, 1993.

[Kae91] Wolfgang Kaefer. A Framework for Version-based Cooperation Control. In *Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications (DASFAA)*, April 1991.

[KLS90] H.F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In *Proc. Conf. on Very Large Data Bases*, pages 95 - 106, August 1990.

[Mos85] J.E.B. Moss. *Nested Transactions - An Approach to Reliable Distributed Computing*. MIT Press, 1985.

[MUZ94] A. Meckenstock, R. Unland, and D. Zimmer. Flexible Unterstützung kooperativer Entwurfsumgebungen durch einen Transaktions-Baukasten. In *Proc. STAK `94 - Datenbanken unter Realzeit- und technischen Entwicklungsanforderungen*, March 1994.

[NRZ92] M.H. Nodine, S. Ramaswamy, and S.B. Zdonik. A Cooperative Transaction Model for Design Databases. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 53 - 85. Morgan Kaufmann, 1992.

[PK92] C. Pu and G. Kaiser. Dynamic Restructuring of Transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 265 - 295. Morgan Kaufmann, 1992.

[RMH+94] N. Ritter, B. Mitschang, T. Härder, M. Gesmann, and H. Schöning. Capturing Design Dynamics - The Concord Approach. In *Proc. IEEE Data Engineering*, February 1994.

[US92] R. Unland and G. Schlageter. A Transaction Manager Development Facility for Non Standard Database Systems. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 399 - 466. Morgan Kaufmann, 1992.

[WR92] H. Wächter and A. Reuter. The ConTract Model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219 - 263. Morgan Kaufmann, 1992.

[WS92] G. Weikum and H.J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 515 - 553. Morgan Kaufmann, 1992.

Arbeitsberichte des Instituts für Wirtschaftsinformatik

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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