# Concepts for Transaction Recovery in Nested Transactions

**Theo Haerder**[1]
**Kurt Rothermel**[2]
**IBM Almaden Research Center, San Jose, CA 95120**

## Abstract

The concept of nested transactions offers more decomposable execution units and finer grained control over recovery and concurrency as compared to 'flat' transactions To exploit these advantages, especially transaction recovery has to be refined and adjusted to the requirements of the control structure

In this paper, we investigate transaction recovery for nested transactions Therefore, a model for nested transaction is introduced allowing for synchronous and asynchronous transaction invocation as well as single call and conversational interfaces For the resulting four parameter combinations, the properties and dependencies of transaction recovery are explored if a transaction is 'unit of recovery' and if savepoints within transactions are used to gain finer recovery units

## 1. Introduction

Atomic transactions were originally introduced for database systems (DBMS) to relieve the application programmer from all aspects of concurrency and failure [EGLT76] A transaction is defined to be a unit of work meaningful for the application environment, it must be executed completely or leave the shared database unaffected ('all or nothing' approach) despite the presence of failures and concurrent users This transaction concept was adopted by many existing DBMS, and now it goes without saying that a DBMS deserving this name has to support the application programmer with the full set of transaction functions

Although perfectly adjusted for short transactions [Anon85], it turns out that single level transactions do not obtain optimal flexibility and performance when executing more complex transactions involving, for example, sequences of joins and sort operations in relational DBMS Better suited execution support would require more decomposable and finer grained control as far as concurrency control and recovery is concerned Therefore, a number of researchers have proposed and advocated the so-called nested transaction concept [Moss81], in the meantime, several systems have incorporated this idea - however, their implementations display substantial differences [Allc83, Jess82, Lisk85, Muel83, SpSc83, Walt84, WeSc84]

Nested transactions aim at an extension of single level transactions by introducing an inner control structure consisting of a hierarchy of subtransactions The nested transaction concept offers a dynamic control structure which is capable of distributing the work within a transaction, executing it in parallel, and rolling back some unsuccessful parts without affecting others Although their full potential may be only utilized in distributed systems, they exhibit at least three important advantages over flat ones

**Intra-transaction parallelism** The larger a transaction is, the more potential parallelism may be anticipated and exploited during its execution The nested transaction concept offers an appropriate control structure to support supervised and, therefore, safe intra-transaction concurrency thereby increasing efficiency and decreasing response time

**Intra-transaction recovery control** An uncommitted subtransaction may be aborted and rolled back without any side-effects to other transactions if its execution was isolated against other parts of the nested transaction The use of conversational interfaces between transactions requires more careful control, since dirty information may be spread out and extend the scope of recovery The flexibility of in-transaction UNDO may be augmented by introducing an appropriate ***savepoint concept*** within nested transactions Hence, nested transactions contribute to a considerable refinement of the scope of in-transaction UNDO as compared to single level transactions where UNDO-recovery necessarily yields the state of 'begin of transaction' (BOT)

**System modularity** Subtransactions facilitate a simple and safe composition of a transaction program whose modules may be designed and implemented independently This system modularity serves other design goals as well encapsulation (information hiding), failure limitation, security

Further, less obvious advantages may be claimed by nested transactions [Lisk85] They include

- use of a powerful and explicit control structure enhancing the reliability of distributed processing
- support of distributed algorithms concerning cost-effective use of hardware, responsiveness, availability (replication of data) and robustness of the overall system



1 Current address University of Kaiserslautern, Department of Computer Science Postfach 3049, D-6750 Kaiserslautern West-Germany

2 Current address University of Stuttgart, Department of Computer Science, Azenbergstr 12 D-7000 Stuttgart 1, West-Germany

- flexible use of implementation techniques in various system parts under the same control structure

Our prime goal is the investigation of powerful intra-transaction recovery for nested transactions Therefore, we introduce a model for nested transactions and describe various degrees of freedom in this model to be utilized for enhanced parallelism and flexible cooperation As our main subject, we explore transaction recovery for single call and conversational interfaces and focus on the use of a savepoint concept for nested transactions Finally, we summarize our results on transaction recovery

# 2. A Model for Nested Transactions

## 2.1 General Properties of the Model

In order to introduce our model of nested transactions, we start with the one described by Moss [Moss81] In this model, a transaction may contain any number of ***subtransactions***, and every subtransaction, in turn, may be composed of any number of subtransactions - conceivably resulting in an arbitrary deep hierarchy of nested transactions We follow the terminology defined in [Moss81] The root transaction which is not enclosed in any transaction is called ***top-level transaction (TL-transaction)*** Transactions having subtransactions are called ***parents***, and their subtransactions are their ***children*** We will also speak of ***ancestors*** and ***descendants*** The ancestor (descendant) relation is the reflexive transitive closure of the parent (child) relation We will use the term ***superior (inferior)*** for the non-reflexive version of ancestor (descendant) In the following we will use the term 'transaction' to denote both TL-transactions and subtransactions

A nested transaction hierarchy may be explained as a collection of *nested spheres of control* [Davi73] where the outermost sphere is formed by the top-level transaction which incorporates the interface to the outside world (user) The transaction properties defined in [Gray80] and [HaRe83] such as

- Atomicity
- Consistency
- Isolated Execution
- Durability (Persistence)

remain valid for TL-transactions Hence, their essential characteristics guarantee that a TL-transaction transforms a database from a consistent state to another consistent state in an atomic manner In case of failures, they enable backout of uncommitted transactions without side-effects to other transactions (isolated execution) Automatic rollback applies to their modified data in the shared database, it does not apply to OS files or terminal messages issued by the transaction program Whenever a conversational interface [Gray78] is used between the DBMS and the program invoking the transaction, the user may be involved in an 'automatic rollback' As soon as a transaction commits, the DBMS is responsible for the persistence of its updates, then, a transaction's effects may be undone only by compensation actions

As opposed to TL-transactions, it is sufficient to provide weaker properties for subtransactions They can terminate either normally by committing or abnormally by aborting like TL-transactions A subtransaction appears atomic to the surrounding transactions and may commit and abort independently It may abort without affecting the outcome of the surrounding transaction However, the commit of a subtransaction is conditional subject to the fate of its superiors, even if it commits, aborting one of its superiors will undo its effects All updates of a subtransaction become permanent only when the enclosing TL-transaction commits Furthermore, it would be unnecessarily restrictive to require consistency to be preserved by a subtransaction Assume, for example, the Debit_Credit transaction [Anon85] Its implementation in a distributed environment by using subtransactions for Debit and Credit would be prohibited by an overly restrictive consistency demand Therefore, we require for them only the following properties

- Atomicity
- Isolated Execution

## 2.2 Degrees of Freedom in the Model

To facilitate our discussion, we assume that every transaction is executed by a single process which keeps its state as long as it is alive, e g in case of a conversational interface This implies that every transaction has only one location in a distributed environment which may restrict the domain of desirable properties However, we don't want to burden our considerations with complex issues which don't seem to be helpful for the evaluation of the underlying concept Extensions may be explored in a subsequent paper

The static nesting structure often illustrated by a 'transaction tree' only sketches the decomposition aspect of nested transactions, that is, how a single level transaction could be broken into parts and then be put together to obtain the benefits listed above In particular, it does not specify whether or not transactions are executed concurrently to achieve some kind of intra-transaction parallelism Furthermore, other issues not covered are the invocation mechanism for subtransactions and the type of interface supported between calling and called transactions

The first aspect to be discussed is the role of the invocation mechanism used to organize the nesting hierarchy The invocation of a subtransaction establishes some kind of client-server relationship between caller and callee Each invocation (call) issues a ***request*** to the corresponding server which acts upon the request and returns an ***answer*** to the caller The execution of a request is done as part of a transaction and is called a ***work step*** of this transaction As will be seen below, transactions can comprise one or more work steps in our model An invocation is called ***open*** if the answer to it has not yet been received by the caller, otherwise it is called ***closed*** The ***invocation/answer-interval (i/a-interval)*** of an (closed) invocation is defined to be the interval between the time this invocation occurred and the time the corresponding answer was received by the caller

In our model, a client can invoke a server ***synchronously*** or ***asynchronously*** If a client calls a server synchronously, it is blocked until it receives an answer Of course, this kind of interaction implies that clients and servers never run concurrently On the other hand, if a client invokes a server asynchronously, it can proceed after the invocation, and hence, can run concurrently with this server Examples of synchronous invocation mechanisms are remote procedure call [Nels81] and the rendezvous concept [Ada83], while (non-blocking) send/receive message passing primitives [TeAn83] are examples for the asynchronous invocation mechanism

The second aspect to be considered is the type of interface used for a client-server cooperation A ***single invocation interface*** (single call interface) provides a mechanism to create a subtransaction and then run it in an atomic manner The subtransaction either aborts or commits and, hence, the caller receives the bad news (notification of abortion) or the (committed) results That is, a transaction created by means of a single invocation interface consists of one work step

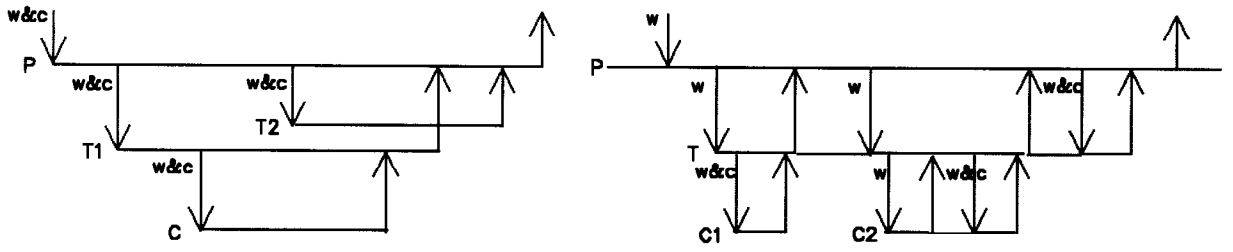


**Figure 1 Communication Among Transactions**

A ***conversational interface*** - in our context - allows for a sequence of invocations of the same transaction The first invocation creates the transaction and the last answer confirms the commit of the created transaction This type of interface within the scope of a nested transaction allows and encourages the exchange of information before the child's commit Obviously, to achieve atomicity for a child in case of a failure becomes more complicated, rollback usually cannot be performed without involving the parent and, of course, the inferiors Here, we will investigate this type of interface to reveal its strengths and weaknesses for its use in nested transactions

## 2 3 Cooperation of Transactions

In the following, we consider synchronous and asynchronous client-server relationships for both types of interfaces in a nested transaction environment For this purpose, we specify primitive operations in an abstract manner which support the appropriate description of the control structure within a nested transaction Therefore, we assume the availability of the following primitives

- work (w) issues a work request to a child transaction
- commit (c) issues a commit request to a child transaction
- abort aborts a transaction
- work&commit (w&c) issues a work request combined with a commit request to a child transaction
- accept accepts an answer or request
- done delivers a done answer to the corresponding parent

Using these primitives enables us to model the client-server relationship more accurately Their model of cooperation is illustrated in Fig 1 (done and accept are omitted in all figures for simplicity reasons) The single call interface shown for asynchronous transaction invocation in Fig 1a obviously contains synchronous transaction invocation as a special case, and, in turn, it could be identified as a special case of the conversational interface (Fig 1b) Its atomicity property implies 'hierarchical containment' of all inferiors, and there is no need to further restrict the client-server relationship

The conversational interface, however, deserves some specific discussion concerning the appropriate trade-off between simplicity, flexibility and complexity Since the client-server relationship is not symmetrical, we postulate that some restrictions on the conversational structure are reasonable A work request of transaction P is completely processed by transaction T (P is parent of T), before the done answer delivers the result, that is, we don't allow a sequence of done messages for a single work request A conversational interface helps in maintaining the environment of T for further work requests of P (e g search using a scan) Within its current work step, T may ask for 'external' support by creating a transaction C and by cooperating with C via a conversational interface Such a cooperation is assumed to be always used for performing a specific task within the current work step of T for P, hence, the service environment of C should not be retained for supporting 'unrelated' future work requests of P Therefore, we feel that the conversational transaction C should be finished within the work step of T where it was created This restriction recursively applied has the following advantages

- It leads to a 'hierarchical containment' of inferiors within work steps, that is, related conversational interfaces always span only two levels in the nesting hierarchy
- It greatly simplifies (although still complex) the overall nesting structure, confines domino effects when a transaction failure occurs thereby limiting the domain of recovery and making recovery easier and more efficient

Strict 2-phase-locking protocols are applied for all transactions For simplicity, we restrict ourselves to the use of Read and Write locks for synchronizing nested transactions [Moss81] Locks are either acquired from the 'outside world' (e g lock manager) or, when already retained by an ancestor, from the transaction's environment When a subtransaction commits, all its locks are inherited (retained) by its parent Newly acquired locks of an aborting subtransaction are freed, while retained locks are kept by its parent A committing or aborting TL-transaction releases all its resources to the 'outside world'

Transactions are considered to run on a collection of sites interconnected by a communication network [MLO86] The (low-level) communication aspects involved are not important for the recovery concepts discussed

# 3. Recovery in Nested Transactions

Thus far, we have described various model aspects of nested transactions and how they cooperate The main issue to be investigated is their behavior in the presence of various failure events To deal with the concepts of database recovery, we need a clear comprehension of

- *the types of failures the database has to cope with,*
- *the notion of consistency that is assumed as a criterion for describing the state to be reestablished* [HaRe83]

## 3.1 Model of Failures

Failure models are discussed in detail in [Gray80] Here, we only summarize the failure types to be considered in database recovery

**Transaction failure** The transaction program does not reach commit (EOT) for reasons like program-enforced abort or rollback in case of deadlocks This type of failure is assumed to be very frequent in large DBMS Dealing with short transactions, it is expected that transaction recovery from such a failure takes place within the time needed for its regular run As more complex transactions are executed, a substantial portion of the resources may be consumed by this type of recovery On the other hand, response time is influenced more significantly due to the longer duration of rollback and reexecution Therefore, special consideration of the optimization of transaction recovery seems to be necessary Nested transactions seem to provide an appropriate framework for this goal

**Node failure** In a distributed environment, a single processor may crash without affecting the remaining parts of the system Consequently, the recovery actions should be restricted to the failing processor and, as a result, influence the surviving nested transactions as little as possible

**System failure.** The entire system crashes and needs a restart to resume its operation With respect to the recovery actions required for a nested transaction, this type of failure is similar to a node crash involving the TL-transactions

**Media failure** Recoverable data are lost due to damage of disk devices

As far as recovery requirements are concerned, atomicity of transaction execution has to be provided as well as the conditional persistence of their results subject to the commit of all superiors Accordingly, system and media failures do not demand new types of recovery algorithms In any case, the most recent transaction consistent database state must be reconstructed which only causes some adjustment and coordination of recovering the related subtransactions Hence, these issues are not touched very much by our considerations Instead, our main emphasis is on transaction recovery

## 3.2 States of a Transaction

For our purpose, it is sufficient to describe the abstract states which are reestablished for a transaction after successful in-transaction recovery without considering 2-phase-commit protocols in detail The specific recovery actions necessary are discussed in subsequent sections

We introduce the recovery-relevant states of a transaction as seen by the transaction manager (TM) A subtransaction running at site S is either in state 'unknown' or 'active' for the TM of S Fig 2a illustrates the state diagram of subtransaction T The initiation of T changes its state from 'unknown' to 'active' T remains 'active' during its execution and its interactions with parent or children Problems may be solved locally when an appropriate savepoint concept is used Hence, RESTORE(T) keeps the transaction's state active after successful in-transaction recovery A system crash or an explicit abort of an 'active' T washes away all information about T, that is, its state becomes 'unknown' After committing T, T's state returns to 'unknown', that is, S can forget T However, successful commit implies the inheritance of T's resources (e g locks and data) to its parent P which is assumed to be represented at S by an agent [Moss81] Note, only 'active' subtransactions may be restored or aborted directly After commit the effects of T may be wiped out indirectly via ABORT(ancestor(T))

A TL-transaction may be in an 'unknown', 'active' or 'prepared' state from the point of view of the TM As shown in Fig 2b, a TL-transaction T enters an 'active' state as soon as T is initiated at site S System crash, ABORT(T) and RESTORE(T) have the same effect for an 'active' TL-transactions as for a subtransaction To commit a TL-transaction, all sites involved have to be coordinated which may appropriately be achieved by using a 2-phase-commit protocol (2PC) Note, the inheritance of resources of committed subtransactions may have created agents of the TL-transactions in a number of sites In a first phase, all participating sites are requested by a coordinator (at the site of the TL-transaction) to enter the 'prepared' state (write modified data or log information to a safe place) Then, a crash does not hurt the transaction anymore During the second phase, T is terminated - either committed or aborted according to the coordinator's decision If all ('important') agents reach the 'prepared' state, the coordinator may decide to commit T As soon as one agent fails to become 'prepared', the coordinator must abort the entire transaction In either case, all resources of T are freed and T returns to the 'unknown' state

TM's are responsible for the overall coordination of transactions and their recovery For a comprehensive description of their specific functions and services, we refer to the literature [Allc83, Roth85]

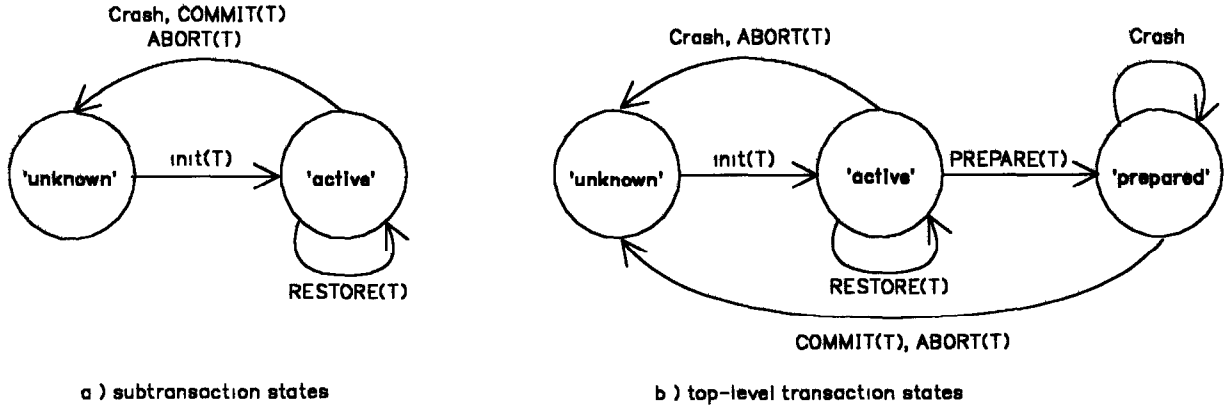


**Figure 2 Transaction State Diagrams**

## 3 3 Recovery in Nested Transactions without Savepoints

In DBMS supporting flat transaction structures, the state to be reconstructed during transaction recovery is the 'unknown' state before transaction start (BOT) Intermediate recovery points were discussed in [Gray81], but no efforts were made to exploit partial in-transaction UNDO The framework of nested transactions, however, offers an appropriate control structure to achieve finer grained recovery domains which establish natural units of in-transaction UNDO

At first, we focus on a recovery concept where transaction UNDO is always applied to an entire transaction wiping out all its effects, that is, the transaction is backed out to its BOT state Since sometimes a significant share of work may be lost due to overly restrictive rollback, we then investigate means to refine the granules of UNDO recovery in case of transaction failure Partial recovery may be achieved by the suitable use of savepoints Such an adjusted savepoint concept can be generalized to be selectively applied to subtransactions

The two types of interfaces play an important role when the scope of recovery has to be determined Here, we explore them separately assuming only one type of interface used within the domain of a nested transaction If both kinds of interfaces occur, appropriate adjustments of the concepts are necessary

### Single Call Interfaces

For our purpose, the most important property of a single call interface is the strict isolation of the superiors against a failing transaction which holds for synchronous as well as asynchronous invocations All transaction failures and their consequences are confined to the very transaction and its inferiors, because a parent only becomes involved when it receives a child's results These properties apply to TL-transactions as well as subtransactions at any level

A successful recovery from a transaction failure requires a complete abortion of the respective transaction T to its BOT state (see Fig 3a) This, in turn, affects all its inferiors - either in progress (only possible for asynchronous invocation) or already committed It is obvious that all its inferiors must be wiped out completely, since their existence is based on the transaction to be aborted Uncommitted children like C2 cannot have released any information to their parent, hence, they can be rolled back without any side-effects to their parent or siblings Since all resources of committed children are inherited (C1 to T, C3 to C2), their UNDO coincides with the rollback of their parents Let us summarize the properties of transaction recovery for single call interfaces

- All descendants are affected
- Synchronous invocation implies that only (the effects of) committed inferiors have to be undone (System-enforced rollback may behave like the asynchronous case)
- Asynchronous invocation may require the rollback of active inferiors as well

How does transaction abort affect the superiors? In case of transaction failure, a parent transaction receives an **aborted** instead of a **committed** message preferably indicating the specific type of failure After failure analysis, it may decide to invoke a new transaction to perform the required task If it cannot figure out an appropriate deviation after a number of unsuccessful attempts, it will finally issue its own rollback and deliver a corresponding message to its parent Hence, this process may be continued recursively until the TL-transaction aborts

The implementation of such recovery protocols will not be discussed in detail in this paper due to space restrictions To enable transaction recovery, some log protocol has to be used to maintain an appropriate structure reflecting the calling hierarchy of the nested transaction Transaction UNDO then refers to this log structure to rollback the effects of the transaction and its inferiors involved subject to restrictions of the internal precedence order Depending on the processing model used it becomes very complex, since quite a number of parameters and options have to be considered Most influential criteria are the kind of distribution of processing, buffer replacement, update propagation and mapping support provided, e g atomic vs non-atomic schemes [HaRe83]

### Conversational Interfaces

Using a conversational interface for the communication between parent and child transaction allows for consecutive work requests to the same child transaction and for consecutive delivery of results to the parent transaction (Fig 1b) The key property of such a scheme is that the parent receives information before EOT of the child, or expressed in a different way, a child passes uncommitted (dirty) information to its parent Since no semantic properties of this information and its usage can be exploited for recovery, worst case assumptions concerning the domain to be poisoned must be applied

The conversational model may be explained as a generalization of the model for the single call interfaces A transaction T processes a number of consecutive work requests of its parent P For each work request, in turn, T may invoke some subtransactions $C_i$ which may deliver their results in conversational mode, too (see fig 3b) An important property of this scheme is that all services of subtransactions are invoked within a single work step of T such that only committed information of $C_i$ may reach P Due to the strict hierarchical containment of transactions, this property holds recursively Hence, the spread out of dirty information is always confined to one level of the nesting hierarchy

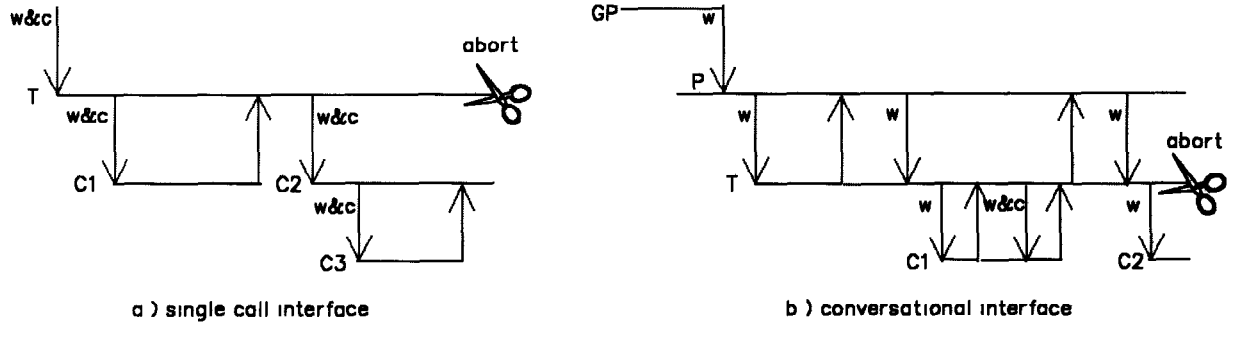


**Figure 3 Transaction Recovery in a Nested Hierarchy without Savepoints**

When a transaction failure occurs, it is obvious that all descendants of the failed transaction have to be completely rolled back similar to the case where single call interfaces are used However, the particular property limiting the scope of recovery to the failed transaction and its inferiors is not valid anymore due to information exchange with P Therefore, rollback of T affects the state of P and consequently, the dependency domain of a failing transaction is increased Note, however, our cooperation model guarantees 'hierarchical containment', i e , the failing transaction has only affected the current work step of P

On the other hand, the unit of UNDO is always an entire transaction Since there is no natural firewall within a transaction other than BOT, P, in turn, has to be rolled back completely thereby possibly provoking some problems for its parent Such a forced rollback issued by a minor 'hierarchical' contamination (of only P) may cause some kind of domino effect for the entire ancestor hierarchy An ancestor transaction that receives an **aborted** message has to abort itself up to the TL-transaction unless it
- receives the bad news in its initial conversational step or
- uses a single call interface

Let us recapitulate the effects of aborting a transaction T for synchronous as well as asynchronous invocations

- The grandparent transaction GP of T is completely isolated
- Parent P is affected because of its dependency on dirty information
- Cascaded rollback of all inferiors is necessary
- With synchronous invocation, only children of T may be active, all other inferiors have already committed when T issues the abort command
- With asynchronous invocation, transaction abort has to cope with rollback of inferiors at any level
- In addition, rollback of superiors may have to be applied recursively due to the lack of appropriate recovery units Therefore, the entire ancestor hierarchy is vulnerable to an inferior's failure, although the propagation of dirty information is always confined to the immediate parent

Apparently, the exploitation of conversational interfaces for nested transactions has severe drawbacks in the event of transaction failures However, it may not be advisable to forbid their use at all They seem to be the appropriate choice in a distributed environment when complex functions are best performed in a piecewise manner keeping the context of the subtransaction, e g scans or temporary results which may be created before by some large sort operations Some improvements are discussed in the next section

## 3.4 Recovery in Nested Transactions Using Savepoints

Transaction recovery has been discussed so far for the case where the subtransaction is unit of rollback Here, we are going to investigate the concept of savepoints within subtransactions in a nested transaction framework in order to provide more adjusted means for transaction UNDO

Are savepoints useful at all? Savepoints are exploited to support a finer grained transaction UNDO to allow for partial rollback of in-progress transactions at each level of the nested hierarchy Using this concept we gain a new kind of flexibility for transaction recovery, since a number of savepoints may be established within a transaction serving as 'restart points' of a running transaction when some problems are encountered (e g wrong operation, deadlock) The transaction/DBMS may then decide which savepoint is the best choice in a given situation

Recovery consisting of a partial transaction rollback is more sophisticated While in the former case only the UNDO log and the list of acquired locks of the respective transaction and its inferiors was sufficient to perform transaction UNDO, we now need at least the UNDO log with appropriately marked savepoints, the list of locks held in each savepoint, and furthermore a state description of the DBMS cursors and the program belonging to the respective transaction for each savepoint After having reestablished the state of the transaction and its inferiors at the chosen savepoint, a restart attempt may be invoked

Hence, compared to the redundancy to be maintained for complete transaction UNDO, the savepoint concept implies more overhead during normal processing Conceivable benefits are tailored recovery options and saving of some work to be otherwise reprocessed during restart These are especially valuable in complex nested transactions for the higher level transactions having long execution paths or numerous inferiors Furthermore, savepoints enable a transaction to better react to undesired events caused by conversational subtransactions, that is, they may effectively limit the scope of recovery Thus, our main goal is the thorough investigation of the savepoint concept We try to evaluate possible approaches including their implications in order to exhibit more clearly the trade-offs involved

### Single Call Interfaces

We continue our discussion of transaction recovery in the presence of single call interfaces from section 3 3 The results derived there can be applied respectively

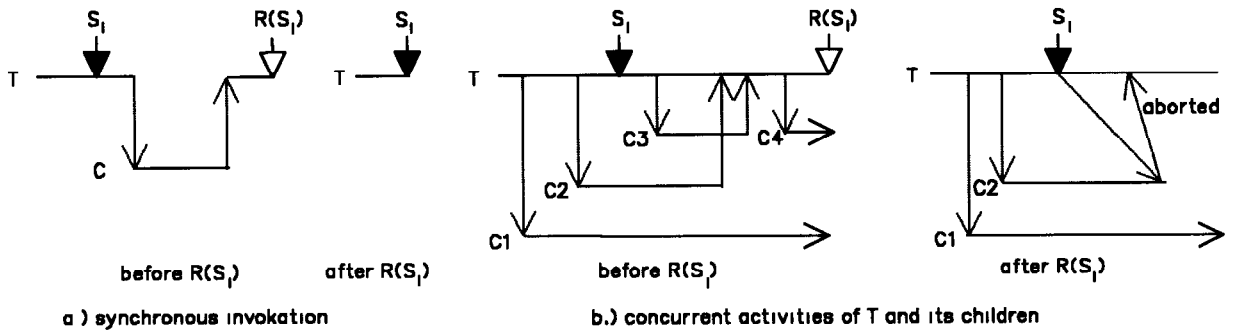


**Figure 4 Savepoint Support for Single Call Interfaces**

Using the savepoint concept in nested transaction hierarchies, it is not obvious what its appropriate semantics is or should be Therefore, we approach the problem by explaining the simple cases first A synchronous invocation of a subtransaction is shown in an illustrative scenario in Fig 4a Assume a savepoint $S_i$ was established ($E(S_i)$) before the execution of child C If T decides to restore to savepoint $S_i$ ($R(S_i)$) at time $t(R(S_i))$ after (>) $t(E(S_i))$, C was either not started yet or already completed (due to the synchronous invocation interface) Hence, the result of $R(S_i)$ is well defined It resembles the use of savepoints for flat transactions, except that recovery has to cope with resources of committed subtransactions

As soon as we permit concurrent execution of T and C, the problem becomes much more complicated, since savepoints may be established as well as restore commands executed while concurrent child transactions are on their way How should such cases be treated? What is the appropriate savepoint/recovery semantics for a restore when concurrent child transactions are allowed? The scenario in Fig 4b sketches the important cases to be covered when $R(S_i)$ has to be performed

We propose the following solution

1 All child transactions of a transaction T whose 'committed' answer was accepted by T before creating the savepoint $S_i$ are not influenced by $R(S_i)$

2 All children of T invoked after $t(E(S_i))$ are rolled back no matter whether or not they had finished (C3, C4)

3 All children of T started before $t(E(S_i))$, but whose 'committed' answer was not yet accepted at $t(R(S_i))$ are not immediately affected (C1), after having finished their execution, T may decide whether or not to accept their results

4 All committed children of T that were activated before $t(E(S_i))$ and whose 'committed' answer was not accepted by T before $t(R(S_i))$ should be rolled back completely As part of $R(S_i)$, the 'aborted' answer (e g of C2) could be created by restore processing which then could be accepted by the restarted transaction Hence, T can accomplish the appropriate measures in its 'new life'

The last decision needs some justification, since various solutions seem to be conceivable for those situations To understand the problem, let us discuss various alternative solutions

**Simulation of history** At first glance, the given solution appears to be unnecessarily restrictive, since T will decide to reprocess such subtransactions in most cases Hence, wouldn't it be much better to avoid reprocessing and to simulate some kind of 'reaccept' by T? To achieve such a task, after restart of T the reception of the child's answer must be performed as if it was sent by the child for the first time It turns out that such a simulation of history is quite a complex task, since data, control information as well as the list of locks to be inherited have to be saved for an eventual reaccept triggered by the savepoint processing This necessity implies to keep potentially all messages for a very long time which may cause some complexities in the implementation of the concept

**Selection of 'usable' savepoints** A second approach tries to prevent or circumvent the sketched problem at all By applying strict rules for the eligibility of savepoints, only those savepoints should be chosen which don't overlap in time with committed child transactions For example, the commit of C2 in Fig 4b makes $S_i$ unusable, that is, $R(S_i)$ as shown could not be processed The selection of the nearest usable savepoint would represent the simplest solution for rollback Apparently, such an approach has severe disadvantages

- It costs some efforts to determine the eligible savepoints in a given (dynamic) situation, an 'accept' may make some savepoints unavailable For example in Fig 4b, 'accept' of C2's answer implies to give up $S_i$ by T
- It may be overly expensive since the next savepoint in question could have been made unusable by an earlier committed transaction Therefore, long rollbacks may be anticipated by cascading effects

**Generation of 'aborted' answers** As indicated above, savepoint processing will roll back the respective transaction and generate an 'aborted' answer Hence, our proposed approach represents a trade-off Since existing concepts and tools may be exploited, generation of 'aborted' answers at savepoint processing may not be overly expensive Compared to the 'simulation of history', however, the aborted work has to be redone (in most cases)

As a result of our discussion, the table below summarizes the recovery actions for the different invocations of child transactions w r t the savepoint to be restored

Let us summarize our discussion of the single call interface so far

- Perfect isolation of superiors was already achieved by single call interfaces without savepoints Hence, no further improvement is possible
- As indicated by Fig 4, synchronous as opposed to asynchronous invocation greatly facilitates savepoint semantics and processing, since all savepoints of a transaction remain 'usable' (Rules 3 and 4 cannot apply)
- All inferiors invoked after $S_i$ have to be undone Similar to section 3 3, synchronous invocation implies only UNDO of committed inferiors, whereas rollback of transactions invoked asynchronously has to cope with active transactions as well

## Conversational Interfaces

Nested transactions with conversational interfaces require even more complex solutions for savepoint recovery, because dirty information can flow from a subtransaction to its parent Here, we will apply the principles and extend the solutions given for savepoint recovery with single call interfaces First, let us consider the simpler case where partial rollback of T is performed within a work step with its parent P, before we discuss transaction rollback to an arbitrary savepoint of T The key property of such a limitation is that savepoint recovery may be confined to T and its inferiors leaving P unaffected Fig 5 attempts to illustrate the problems involved

| t(BOT(C)) | t(EOT(C)) | | |
|---|---|---|---|
| | < $t(E(S_1))$ | > $t(E(S_i))$ and < $t(R(S_1))$ | > $t(R(S_1))$ |
| < $t(E(S_1))$ | do nothing | rollback, accept aborted, decide what to do | do nothing, accept answer later |
| > $t(E(S_1))$ and < $t(R(S_1))$ | not possible | rollback | rollback |

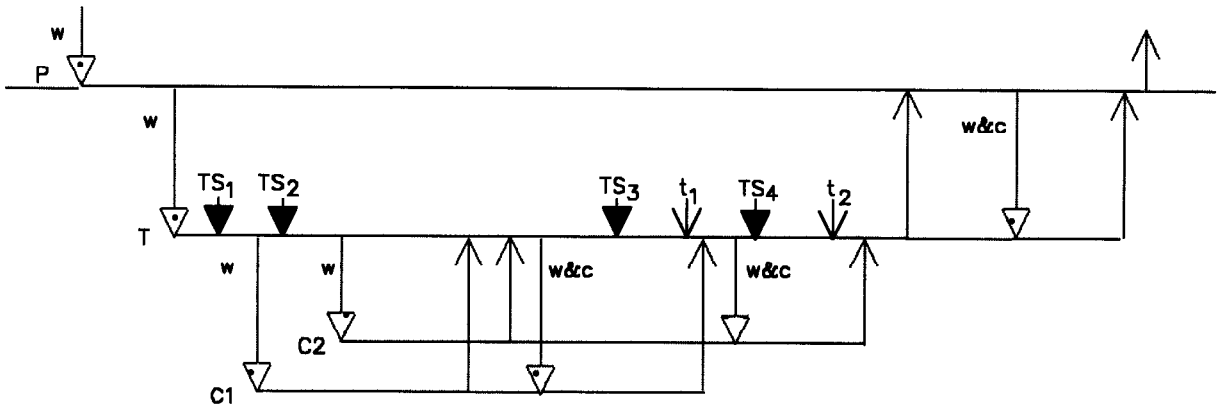


**Figure 5 Savepoint Support Using Conversational Interfaces**

As an important design principle, partial rollback of T should be kept independent of the actions of its children This is in accordance with modularity of design claimed earlier In order to gain a kind of independence for T and to avoid cascading rollback of all inferiors, we assume that implicit savepoints are taken at the begin of every work step of every transaction - specially marked in Fig 5 When T is restored to an intermediate savepoint, the appropriate implicit savepoints of all active children of T are determined After successful rollback to these savepoints, they may be restarted together with T Such a measure helps to save the work of 'conversational' children as far as possible without getting dependent on their explicitly created intermediate savepoints

The simplest solution would be exclusively based on the use of these implicit savepoints at the beginning of each work request Each problem occurring during the execution of a work request is resolved by T rolling back to the very savepoint and retrying the execution or sending an appropriate message to P This approach, however, sacrifices the work of all inferiors generated to support the work request when in-progress transaction recovery becomes necessary

Refined recovery actions are possible as soon as intermediate savepoints are taken As compared to single call interfaces, the following properties provoke considerable additional complexities of savepoint recovery

- Work steps of a child of transaction T may span several savepoints within T (but not several work steps of T), e g the work steps of C1 span $TS_2$ and $TS_3$ in Fig 5
- Stepwise rollback of a subtransaction is only possible before its EOT As a consequence, only the implicit savepoint at BOT(T) and $TS_1$ could be reached from anywhere within T after their creation (refer to Fig 5) At time $t_1$, restore to $TS_2$ and $TS_3$ could be executed, but not at time $t_2$ where $TS_4$ could be reestablished After EOT, the process carrying the transaction does not exist anymore Hence, only its complete rollback using the log is achievable

Our 'single call' approach provided a solution where every intermediate savepoint could be selected Using a conversational interface each committed child transaction causes an 'invalidation interval' for savepoints of T As a consequence, we have to exclude those savepoints which overlap in time with committed children To determine the usability of a savepoint $TS_i$ at $t(R(TS_i))$, we have to check for all committed children $C_j$ of T whether or not

$$t(BOT(C_j)) \leq t(E(TS_i)) \leq t(EOT(C_j))$$

holds If true for any $C_j$, $TS_i$ cannot be used anymore and the next older $TS_k$ has to be examined

When a usable savepoint $TS_i$ is encountered (in the worst case BOT), we can determine the state of every child transaction which has to be reestablished after successful $R(TS_i)$ by using one of its implicit savepoints Here are the simple cases

- All children of transaction T whose commit was accepted by T before creating the savepoint $TS_i$ are not influenced by $R(TS_i)$
- All children of T invoked after $t(E(TS_i))$ are rolled back entirely no matter whether or not they had finished

Child transactions spanning $TS_i$ and not yet committed when $R(TS_i)$ is issued, are much more difficult to cope with The rules for rolling back children of T when T is restored to a usable savepoint are illustrated in Fig 6 Every child transaction $C_k$ spanning $TS_i$ has to be restored according to the following rules, where the $R(TS_i)$-interval is defined as the time span between $t(E(TS_i))$ and $t(R(TS_i))$

1 $C_k$ is not involved in $R(TS_i)$ if the last invocation of $C_k$ is open and $E(TS_i)$ occurred after this invocation

2 $C_k$ is not involved in $R(TS_i)$ if the $R(TS_i)$-interval does not overlap with any i/a-interval of $C_k$

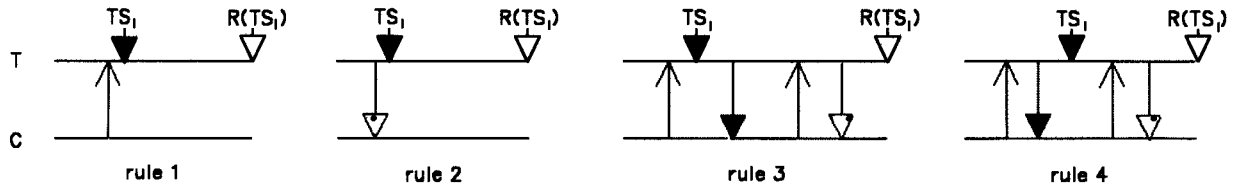


**Figure 6 Use of Implicit Savepoints in Conversational Transactions**

3 If $E(TS_i)$ is not contained in an i/a-interval of $C_k$ and T invoked $C_k$ at least once, then the oldest implicit savepoint IS of $C_k$ such that $t(IS(C_k)) > t(E(TS_i))$ is chosen

4 If $E(TS_i)$ is contained in an i/a-interval of $C_k$ and if T accepted at least one answer of $C_k$ in the $R(TS_i)$-interval, then the most recent implicit savepoint IS of $C_k$ meeting the condition $t(IS(C_k)) < t(E(TS_i))$ is chosen

Remark An option to simplify the savepoint scheme would be to exclude the use of savepoints contained in an i/a-interval of $C_k$ which would permit to abandon rule 4

Fortunately, these rules are not applied recursively (as a consequence of our cooperation model), by restoring $C_k$ to an implicit savepoint, the concerned inferiors are rolled back entirely

After having discussed the essential aspects of savepoint recovery within a work step of T, we will shortly consider backout to a previous work step of T As indicated by Fig 5, the parent P gets involved in the savepoint recovery such that rollback cannot be performed independent of P Optimal solutions w r t amount of work saved would require the coordination of intermediate savepoints of P with those of T which contradicts our design philosophy Therefore, we propose to treat this case like transaction abort The parent gets the 'aborted' answer and has the choice of at least three options

- It issues the backout of its current work step using the implicit savepoint at the very beginning of the step
- It rolls back to an appropriate intermediate savepoint, if available
- It decides something else at its own risk (e g do nothing)

Finally, let us conclude our main results on savepoint recovery for conversational transactions

- Using the concept of implicit savepoints, savepoint recovery may be performed 'independently' thereby saving completed work steps of children as far as possible This results should be compared to transaction UNDO where always all descendants are wiped out
- Cascaded backout of superiors is prevented In any case, the scope of transaction abort may be restricted to the current work step of its parent
- Synchronous invocation facilitates the selection of the implicit savepoints of the children considerably Only rules 1 and 3 apply
- Rollback of inferiors is simpler in the synchronous case compared to the asynchronous case, since only child transactions are active, when a transaction decides to backout to a savepoint (see section 3 3)

# 4. Conclusions

We have presented a framework for the design of algorithms for transaction recovery in nested transactions It includes the use of synchronous and asynchronous transaction invocation as well as the application of single call and conversational interfaces for transaction cooperation

The focus of the paper has primarily been the concepts of transaction recovery and the use of savepoints in nested transactions As pointed out, no particular differences for crash and media recovery as compared to flat transactions are visible at the conceptual level We could not touch the aspects of node recovery In our transaction model, it will be treated like crash recovery However, some optimizations may be applicable if the transaction's data are kept 'prepared' at the node Hence, recovery could cope with a single node failure and reestablish the 'prepared' state with appropriate REDO actions which may leave the nested transaction unaffected Issues of node autonomy after transaction commit, but before final commit of a TL-transaction could also be considered (network partitioning, long message delays, local deadlocks, etc ) These open problems enforcing a refinement of our transaction model are currently explored

Our discussion has revealed the strong isolation of single call interfaces which transaction UNDO may greatly benefit from In any case, transaction recovery is confined to descendants whereas conversational interfaces permit the spread out of dirty information to the parent This makes the entire nested transaction vulnerable to a single transaction abort within its scope when adjusted recovery units are not available

Without exploiting savepoints, recovery unit and concurrency control unit within a transaction coincide By the use of savepoints both issues may be separately treated allowing for greater flexibility in design and implementation as well as better performance and failure isolation in distributed systems We have discussed the application and complexities of savepoints to minimize work to be redone when a local problem is encountered and to limit the recovery scope (against ancestors) when conversational transactions are used Intermediate savepoints involve complex rules for their application Simpler solutions could be found when savepoints are taken at BOT and at the begin of work steps Such actions could be done automatically by a recovery component to facilitate transaction programs

Nevertheless, minimizing the use of conversational interfaces within a nested DBMS transaction greatly improves the isolated recovery of minimal scopes Typically, at least the TL-transaction has a conversational interface towards the application The application itself may be structured in a nested fashion using special types of nested DBMS transactions [BKK85, KLMP84] In such an environment, the application may have further recovery requirements which must be satisfied by the application itself Hence, the DBMS is responsible only for the DB part of such a transaction

## Acknowledgments

We would like to thank C Mohan, J Palmer, P Schwarz, L Svobodova, and B Yost for careful reading and detailed comments of an earlier version of the paper

# Bibliography

[Ada83] Reference Manual, *Ada Programming Language,* United States Department of Defense, Washington, ANSI-MIL-STD-1815A-1983 (1983)

[Allc83] Allchin, J E , *An Architecture for Decentralized Systems,* Technical Report GIT-ICS-83-23, School of Information and Computer Science, Georgia Institute of Technology (1983)

[Anon85] Anon et al , *A Measure of Transaction Processing Power,* Datamation, April issue (1985)

[BKK85] Bancilhon, F , Kim, W , Korth, H F , *A Model of CAD Transactions,* Proc 11th Int Conf on VLDB, Stockholm, Aug 1985, pp 25-33

[Davi73] Davies, Ch T , Jr , *Recovery Semantics for a DB-DC System,* Proc ACM National Conference 28 (1973) pp 136-141

[EGLT76] Eswaran, K P , Gray, J N , Lorie, R A , Traiger, I L , *The Notions of Consistency and Predicate Locks in a Database System,* CACM **19 11** (1976) pp 624-633

[Gray78] Gray, J N , *Notes on Database Operating Systems,* Operating Systems - An Advanced Course, Lecture Notes in Computer Science 60, Bayer, R , Graham, R M , Seegmueller, G (eds ), Springer-Verlag (1978) pp 393-481

[Gray80] Gray, J N , *A Transaction Model,* Research Report RJ2895, IBM Research Laboratory, San Jose, Calif , August 1980

[Gray81] Gray, J , McJones, P , Blasgen, M , Lindsay, B , Lorie, R , Price, T , Putzolu, F , Traiger, I , *The Recovery Manager of the System R Database Manager,* ACM Computing Surveys **13 2** (1981) pp 223-242

[HaRe83] Haerder, T , Reuter, A , *Principles of Transaction-Oriented Database Recovery,* ACM Computing Surveys **15 4** (1983) pp 287-318

[HaRo86] Haerder, T , Rothermel, K , *Concurrency Control Issues in Nested Transactions,* Research Report, IBM Almaden Research Laboratory, San Jose, Calif , (in preparation) (1986)

[Jess82] Jessop, W H , *The EDEN Transaction Based File System,* Proc 2nd Symp on Reliability in Distributed Software and Databases (1982)

[KLMP84] Kim, W , Lorie, R , McNabb, D , Plouffe, W , *Nested Transactions for Engineering Design Databases,* Proc 10th Int Conf on VLDB, Singapore, Aug 1984, pp 355-362

[Lisk85] Liskov, B , *The ARGUS Language and System,* Distributed Systems - Methods and Tools for Specification An Advanced Course, Lecture Notes in Computer Science 190, Chapter 7, Paul, M and Siegert, H J (eds ), Springer-Verlag (1985) pp 343-430

[MLO86] Mohan, C , Lindsay, B , Obermark, R , *Transaction Management in R* Distributed Data Base Management System,* ACM Transactions on Database Systems, Dec 1986

[Moss81] Moss J E B , *Nested Transactions An Approach to Reliable Computing,* M I T Report MIT-LCS-TR-260, M I T , Laboratory of Computer Science (1981)

[Muel83] Mueller, E T et al , *A Nested Transaction Mechanism in LOCUS,* Proc 9th ACM Symposium on Operating Systems Principles (1983)

[Nels81] Nelson, B , J , *Remote Procedure Call,* PhD thesis, Carnegie-Mellon University, May 1981

[Roth85] Rothermel, K , *Communication Support for Distributed Database Systems,* Proc GI-NTG Conf Communication in Distributed Systems (1985)

[SpSc83] Spector, A Z , Schwarz, P M , *Transactions A Construct for Reliable Distributed Computing,* Operating Systems Review **17 2** (1983) pp 18-35

[TeAn83] Terry, D B , Andler, S , *The COSI Communication Subsystem Support for Distributed Office Applications,* IBM Research Report RJ4006, San Jose, Calif (1983)

[Walt84] Walter, B , *Nested Transactions with Multiple Commit Points An Approach to the Structure of Advanced Database Applications,* Proc 10th Int Conf on VLDB, Singapore, Aug 1984, pp 161-171

[WeSc84] Weikum, G , Schek, H -J , *Architectural Issues of Transaction Management in Layered Systems,* Proc 10th Int Conf on VLDB, Singapore, Aug 1984, pp 454-465