NESTED TRANSACTIONS WITH MULTIPLE COMMIT POINTS: AN APPROACH TO THE STRUCTURING OF ADVANCED DATABASE APPLICATIONS

Bernd Walter

University of Stuttgart
Azenbergstr. 12, D-7000 Stuttgart-1
Federal Republic of Germany

ABSTRACT

A new type of transactions for higher level application programming in systems with databases is introduced. These so-called 'nested transactions with multiple commit points' support operations over multiple applications either atomically, independent, or in a combination of both. Furthermore, it is strictly distinguished between transactions as units of work and transactions as a part of so-called 'commit spheres' and 'backout spheres', which provides more generality and flexibility than existing models.

INTRODUCTION

A transaction is a partially ordered set of actions that logically belong together. Transactions are either executed completely and correctly or they leave the system as if they had never existed. Since from a user's point of view transactions abstract from failure handling and synchronization, most future systems will be transaction oriented.

Future applications will also be less monolythic than today's applications. Assume a system that contains transaction-oriented (distributed) application systems like a database system, a calendar system, a document storage system, and an electronic mail service (which can also be interpreted as a local service, that puts a message in a user's private mailbox). Then there might be users that just access only one of the application systems, but there might also be users that want to run more complex applications. Such complex applications might require higher level transactions that enable atomic updates over multiple application systems, however, there will also be applications permitting an independent updating of the various application data.

As an example for an atomic update over multiple applications assume an user that wants to arrange a meeting; for this purpose the appropriate date of the meeting will be entered into the calendar system and be distributed to all participants using the mail service. If the user wants to be sure that the calendar system only contains dates distributed to the users and the users only receive dates included in the calendar system, he has to use an atomic update facility.

As an example for an independent updating over multiple applications assume an user that enters some new business data into the database and then prepares a business report based on the database. Then it might not be necessary to update the database and the document storage system atomically. However, there might be various reasons for including both activities into the same transaction, for instance if the document may only be prepared after the data was entered into the data base, and if it is more efficient to use some output of the first activity as an input for the second activity. Nevertheless it might not be necessary to keep all database locks (provided that locking is used) until the document has been prepared and stored away. Note that the use of two separate transactions in this case could be less efficient, since additional bookkeeping as well as an additional output/input of intermediate results might be necessary.

Of course there might be even higher level activities built on top of our two example applications, i.e. an activity where a meeting has to be scheduled and where additionally all participants must be supplied with business reports. In this case we might have atomic as well as independent updates over multiple applications.

In this paper principles will be discussed for structuring transactions in a centralized environment (an extension of our approach to a distributed environment is given in a separate paper /Wa84/), that support atomic as well as independent updates over multiple applications. Independent updating means, that there are different commit points for the different applications, i.e. differents points of time where the updates become available to the general community of users. Hence, we also speak of transactions with multiple commit points. In /Li83/ (but not in earlier versions of this paper) so-called "nested top actions" are discussed, which also perform independent updates, however, there is no detailed discussion of the various consequences of this approach.

Our approach of structuring transactions leads to a generalized form of nested transactions. In comparison with earlier proposals our model provides the following advantages:

- It is more general than any other model, i.e. existing models can be shown as being special cases of our model.
- The notion of backout spheres permits a general representation of backout dependencies which is important for handling conversational transactions, which are not handled in other models (e.g. /Mo81/).
- The notion of commit spheres supports both atomic and independent updates over multiple applications and is independent of the notion of backout spheres, a feature not supported by any other models.
- The proposed locking scheme, which is based on two-phase locking is more flexible than the schemes proposed earlier for nested transactions (e.g. /Mo81, Mo82, Mu83/).

The remainder of the paper is organized as follows. At first we will give a model of our overall system. In this context our generalized model of nested transactions will be defined and the various protocols for synchronization, back out, and commit will be given. Then some implementation problems will be discussed. Finally, our approach will be compared with other proposals.

SYSTEM MODEL AND DEFINITIONS

Our system consists of application systems (or for short: applications) and transactions.

An application consists of data elements and a nonempty set of functions for retrieving, manipulating, and controlling the data elements. A data element is a basic data item accessed directly by the application or a higher level data element that is provided by some other application. A basic data item may be accessed directly by multiple applications.

Our system supports the construction of higher level applications on top of existing applications. Each application interface can be used by end users as well as by higher level applications.

Generally, a transaction can be defined as a partially ordered set of actions that logically belong together and that possess the atomicity property. The definition of the term transaction in our system depends on the level of abstraction:

- From an end user's point of view a transaction is a partially ordered set of requests to an application system (we use the more general notion of a partially ordered set, i.e. parallel programs, even if often only totally ordered sets, i.e. sequential programs, are supported).
- From an application system's point of view, a transaction consists of a set of internal operations, a set of "sub"-transactions executed on lower level application systems and a precedence structure defining a partial order over the elements of these two sets.

So transactions in our system are in fact nested transactions. However, as will be seen later on, we will use a much more generalized structuring of nested transactions than earlier proposals, e.g. /Mo81, Mu83, Al83/.

In our model of a centralized system two types of failures can occur:

- Processor failures. A processor is defined as an abstract entity consisting of the physical processor and the system software. It is assumed that such a processor shows a so-called failstop behaviour (for a discussion of failstop processors see /Sc83/), i.e. in the case of a failure the processor stops immediately, no garbled outputs will be produced. The contents of the volatile memory are lost if a processor fails. This class of failures includes all failures that require a restart of the system such as failures of the physical processor, of the operating system, etc.
- Logical application failures. It is assumed that all logical failures can be detected and that they cause the affected application to stop. Examples of such failures are the violation of integrity constraints or deadlocks. No other applications are affected by such failures, no memory is lost.

With each transaction executed by an application system two states are associated:

- Volatile state. The volatile state is presented by the values of all variables kept in the main storage. The volatile state is lost during each processor failure.
- Stable state. The stable state is presented by the data stored on stable storage /La79/. Stable storage is assumed to survive each processor failure.

The overall structure of our system can be given as a set of cycle-free directed graphs with the applications as the nodes. An arc from application a_1 to application a_2 means that application a_1 is built directly on top of a_2, i.e. a_1 is a user of the services of a_2. If a new application is added to the system, a new node is added to one of the graphs with arcs leading to all lower level applications which are utilized by this new application. Since multiple higher level applications may utilize the same lower level application, the directed graphs are not trees. The various graphs may be unconnected, since there might be sets of applications, which do not utilize members of other sets. An example of such an application graph is given in Fig. 1. Note, that even applications at various levels of abstraction may share the same data elements.

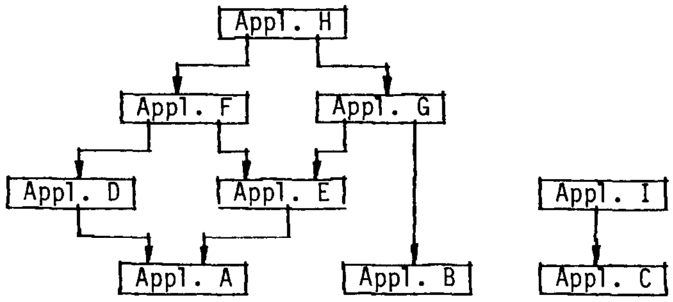

Fig.1: An Application Graph

Within our graph structured system, users (end users or higher level applications) may select any application to work with, no matter at what point of the graph the application is located. For each application, the user wants to work with, he must start a separate transaction. However, an end user may work with multiple applications at a time by constructing a higher level transaction on top of all these separate transactions. In this higher level transaction it can be decided whether the lower level transactions are executed independently or whether they commit together (i.e. whether they are committed at the same point of time or not). This means, the user is permitted to extend the system provided graph structure temporarily with his own private applications. An example of such an extension is given in Fig. 2, where a user has added a private application P on top of the applications H and G of Fig. 1.

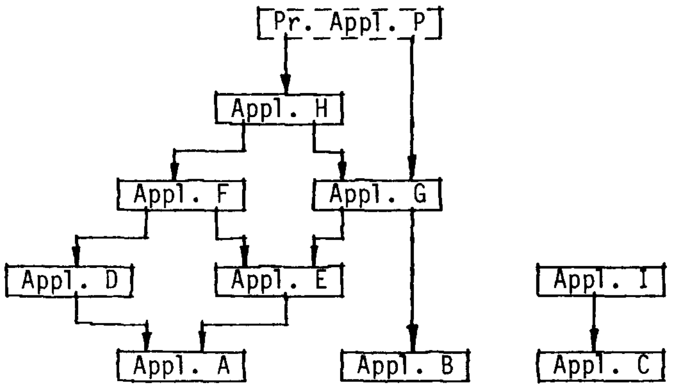

Fig.2: An Extended Application Graph

Each application a_i utilized by the end user may start 'child' transactions on an application a_k if there is an arc that leads directly from a_i to a_k. If within such a nesting hierarchy two applications create child transactions on the same lower level application, then they start different child transactions. This means that the nesting hierarchy of transactions in our system forms a tree structure. An example is shown is Fig. 3, where a user has created a transaction on its private application P, which in turn has created one child transaction on application H and two child transactions on application G etc. Note, nothing is said about whether child transactions exist at the same time or not.

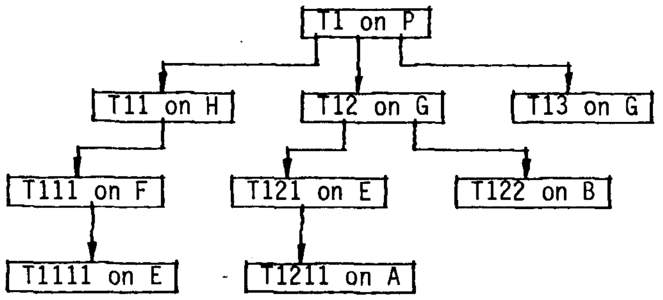

Fig. 3: A Nested Transaction

In the remainder of the paper we will use the following terminology:

- The transaction at the root of a nesting tree is called top level transaction.
- If a transaction T_i has created a transaction T_k, then T_k is the child transaction of T_i and T_i is the parent transaction of T_k (we will also use the short terms parent and child).
- All transactions on the path from the top level transaction to a transaction T_i are ancestors of T_i (of course excluding T_i itself).
- All transactions that belong to the subtree of which T_i is the root are the descendants of T_i (again excluding T_i itself).

So far we have only defined, that nested transactions are used in our system, however, nothing has been specified concerning synchronization, commitment and backing out of the transactions at the various levels of our graph structure. Before we will define the semantics of creating a child transaction more precisely, we will informally discuss several aspects of the relationship between parent and child transactions.

NESTING TRANSACTIONS

In this chapter it will be discussed how the transactions in our system should be nested. Attributes will be defined, that determine the various ways a child transaction can be created by a parent transaction.

A) The Interface Aspect

Applications generally provide their users with two types of interfaces (in the following an user is always a higher level application, otherwise the term end user will be used):

- 'Single-request'-interfaces. In this case there are two interaction points between the user and the application. The first interaction is needed to hand over the complete query or update request to the application, the second is needed to return the result or some status value back to the user. Between these two points the application is responsible for controlling the processing of the user's request. An example for an application with such an interface is a data base system with a stand-alone SEQUEL-interface.

- 'Conversational' interfaces. In this case the user issues a sequence of requests to the application and then decides whether to commit or to abort the corresponding transaction. During this conversation the control changes between the user and the application, each time the user issues an request, the application is responsible for generating an answer, each time the application has issued a response, the user is responsible to tell the application what to do next. An example for an application with a conversational interface would be a data base system with a 'next-tuple'-interface.

To understand the difference between these two interfaces, we have to consider the consequences for the backout behaviour of the complete system, i.e. the system consisting of an application and its user (user means either an end user or an higher level application). Between two calls both the application and the user have to remember their previous state, i.e. the application must know what for instance 'next' means in a 'next-tuple'-call and the user must know that a delivered tuple is the next to the tuple delivered before. The usual implementation technique is to keep the state information of both the application and the user in volatile storage and to change this state incrementally after each interaction.

If an application fails logically, it must back out at least to the previous stable state. In the case of a conversational interface the state of the user strongly depends on the state of the application. Since the user will usually not be able to reconstruct an arbitrary previously volatile state, he must also back out to an earlier stable state, which of course must be synchronized with the stable state to which the application has backed out. In the case of a single-request interface the backing out of the application does not require the backing out of the user. If the user backs out, then the application must back out as well independent of the type of the interface.

So far we have only spoken about the interactions between an user and an application. In fact, anything done in our system is related to transactions. So we have to reformulate the above statements in terms of transactions. If we replace 'user' by 'parent transaction' and 'application' by 'child transaction', we can say: a parent transaction may interact with its child transactions either via a single-request or via a conversational interface.

This leads to the concept of backout spheres. A backout sphere includes one or more transactions. The backout of a transaction of a given backout sphere always implies the backing out of all other transactions in this backout sphere as well. The transactions of a backout sphere form a tree structure, which must be a connected part of the original transaction tree.

In the case of a conversational interface normally both the user and the application will belong to the same backout sphere, in the case of a single-request interface the application will normally belong to a separate backout sphere which is an inner sphere of its user's backout sphere (see also the concept of sphere-of-control in /Da73/ and the discussion in /Mo81/). Clearly, not only technical reasons will influence the decision whether two transactions will be included in the same backout sphere or not. If, in the case that a child fails, the parent should have the chance to create some other child for processing its requests elsewhere, then the parent and the child must be in different backout spheres.

Fig. 4 shows the nested transaction of Fig. 3 with possible backout spheres. Note, that although T121 and T122 belong to different backout spheres they can be backed out together, if such a dependency is realized in their parent transaction T12.

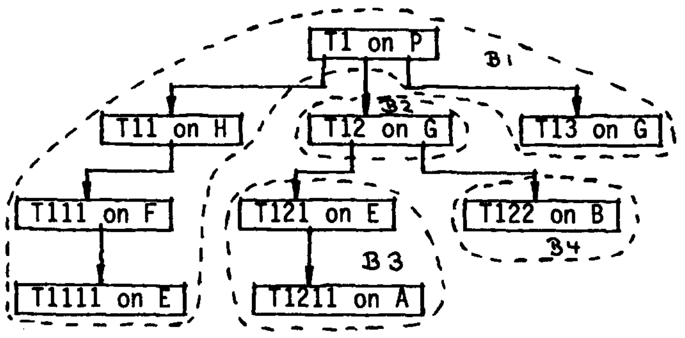

Fig. 4: A Nested Transaction with Backout Spheres

If all nodes belonging to the same backout sphere are put together into one node, then this again results in a tree structure, a tree of backout spheres. In this tree each backout sphere b_k that is a child of a backout sphere b_i, is an inner sphere of b_k. Fig. 5a shows the corresponding tree of backout spheres. The backout sphere B2 is an inner sphere of B1 and the spheres B3 and B4 are inner spheres of B2. An alternate, nested representation is given in Fig. 5b. Backout of the outer sphere implies the backout of the inner spheres but not vice versa.

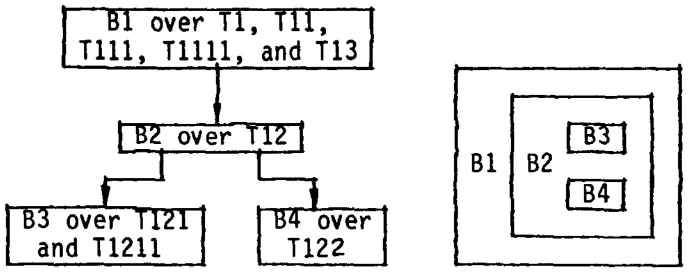

Fig. 5: Backout Spheres,
a) Tree Representation, b) Nested Representation

In our model a child transaction can be created in both ways, either as part of the backout sphere of its parent transaction or as part of a new, separate backout sphere. The selected alternative is specified by the parent transaction as an attribute value in the corresponding creation command (BACKOUT, if a new backout sphere is created, NOBACKOUT, if the parent's sphere is extended).

B) The Dependency Aspect

As already discussed in the first chapter of this paper it might be useful to permit that updates on different applications commit independently, even if they have been initialized under the same top level transaction. Our philosophy is, that a child transaction which is permitted to commit its updates independently of its parent transaction should be handled as a totally different transaction with the following exceptions:

- Input and output data are exchanged directly with the parent transaction.
- Child and parent transaction may belong to the same backout sphere.

The latter is necessary mainly because the interface may be conversational, i.e. the control may switch multiple times between parent and child. Clearly, if the child transaction has committed, the parent transaction must

- either be able to remember the commitment and the committed child's output even after a restart (i.e. by writing the necessary information into stable storage)
- or it should be a desired fact, that the committed child transaction is executed again after a restart of its parent.

The latter might make sense, for instance, if a child transaction is started in order to write some statistical information about the caller into some data base (e.g. for bookkeeping reasons). If the parent is restarted after a processor failure, this could be considered as a different execution which makes it necessary to write the statistical information again.

Since in our system both should be supported, independent child transactions as well as child transactions that commit in coincidence with their parent, we require that each time a child transaction is created, it must be specified whether the child is independent (COMMIT) or not (NOCOMMIT). It should be clear, that a transaction may start multiple independent child transactions, that nevertheless can commit all at the same time (if the parent acts as a coordinator during the commitment of these child transactions).

The corresponding concept to backout spheres is the concept of commit spheres. A commit sphere includes one or more transactions. The commitment of each of these transactions is only possible if all other transactions of this commit sphere also commit. The transactions of a commit sphere form a tree structure, which is a connected part of the original transaction tree. Fig. 6 shows the transaction of Fig. 3 with a possible arrangement of commit spheres. In our model the arrangement of a transaction's commit spheres may be different from the arrangement of its backout spheres. The problems that are caused by this fact will be discussed in a later chapter of this paper.

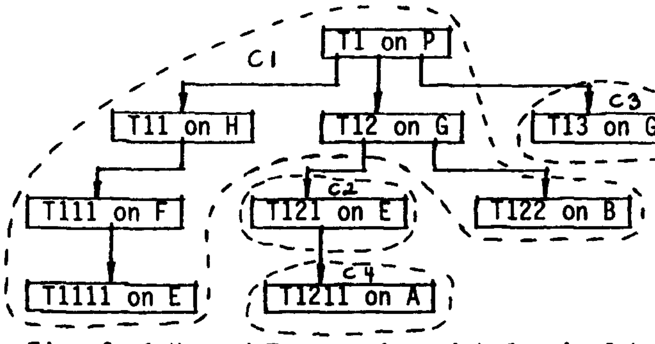

Fig. 6: A Nested Transaction with Commit Spheres

Fig. 7a shows the corresponding tree of commit spheres. The commit spheres C2 and C3 are inner spheres of C1 and the sphere C4 is an inner sphere of C2. The alternate nested representation is given Fig. 7b. Commitment of the outer sphere implies commitment of the inner sphere but not vice versa.

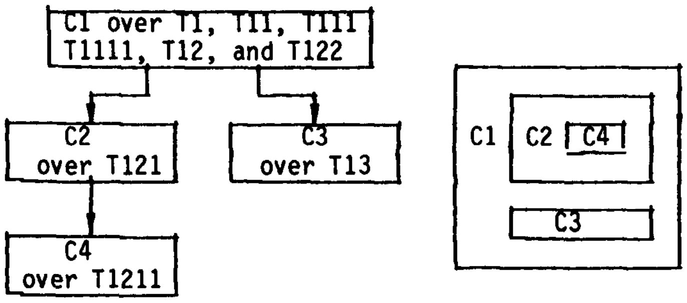

Fig. 7: Commit Spheres,
a) Tree Representation, b) Nested Representation

C) The Synchronization Aspect

In the context of normal transactions, i.e. transactions without internal parallelism, synchronization always means synchronization against other transactions. Nested transactions permit internal parallelism or concurrency, such that synchronization might also be necessary within a transaction. Two types of internal concurrency can occur in our model of nested transactions.

- Concurrency between a transaction and its ancestors and descendants. This happens if such transactions are executed in parallel and do access the same data elements. For a better understanding we have to consider, that the request sent from a parent transaction to one of its child transactions can be issued in two ways, in form of either a synchronous call or an asynchronous call. In the first case, the transaction, that has called one of its child transactions, does not proceed in its own processing until the result of the call has been returned. In the latter case, the parent transaction does proceed in its own processing and then fetches the returned results at some later point of time. Obviously, concurrency between parent and child can only occur in the latter case. If a child is called synchroneously, then it is possible to permit, that it accesses data elements already locked by its parent, if the parent was called synchronously as well, then the child may also access data elements locked by its grandparent and so on (provided that synchronization is based on locking as discussed in a later chapter). The same might be possible if the parent could guarantee that it does not access certain data elements during asynchronous activities.

- Concurrency between children. This type of concurrency can occur between children that have at least one ancestor in common (parent or grandparent or grandgrandparent,...), but where no one of the children is an ancestor of any other of the children. As in /Mo81, Mu83/ we adopt the philosophy, that such children should by synchronized against each other.

Each time a child transaction is created, an attribute value must be passed to the child indicating whether it must synchronize against its parent (SYNC) or not (NOSYNC). In the latter case it may use the locks of its parent (provided that both lock at the same level of abstraction). Additionally, the child must be informed whether its parent was called synchronously and if so, whether its grandparent ... and so on. In fact, as will be seen later on, information is forwarded about the complete transaction path from the top level transaction to the concerning child. Since a fixed strategy is used for handling concurrency between children, no information must be passed to a child about sisters and brothers.

We do not require that NOSYNC is only used in the case of a synchronous call, i.e. a parent transaction can also decide to execute unsynchronized child transactions asynchronously if it is sure that no interference can occur. The parent transaction must also take care, that a child created in parallel with other children is either created SYNC or does not interfere with the other children when accessing objects locked by its parent (note, that children are only synchronized against each other on objects where they are required to set their own locks).

D) Relationships between Parent and Child

So far we have defined, that the three attributes are necessary to describe the relationship between parent and child transactions. However, it is still not quite clear whether all combinations of the attribute values are really necessary. In theory, eight combinations are possible which will be discussed in the following:

- COMMIT, BACKOUT, SYNC. A call with these attribute values creates an independent child transaction with its own backout sphere. This child transaction may not use locks held by its parent. If this would be permitted in a case where both principally request the same locks, then the child transaction could update a data element already updated by its parent, such that the later update, which depends on the first would be committed first. To avoid such anomalies, in the above mentioned case COMMIT should always imply SYNC.
- COMMIT, BACKOUT, NOSYNC. This combination makes sense in the case where it is clear that the child will not request the same locks as its parent. One case of special interest is a system where synchronization takes place at different levels of abstraction (see for instance /We84/). Assume a parent transaction setting locks at the record level and a child transaction setting locks at the page level. Now, if a record is updated, it will be locked by the parent, whereas the child, which performs the physical update, will lock the corresponding page. When the child has finished the updating, it can commit and unlock the page such it can be used by other transactions. The record remains locked by the parent such that no update anomalies can occur (of course, if the parent backs out, a new child transaction must be created to compensate the effects at the page level; this will be discussed again in a later chapter of this paper). Obviously, this is an example of the combination COMMIT/NOSYNC that makes practical sense.
- COMMIT, NOBACKOUT, SYNC. A child transaction created with these attributes is independent, but does not possess its own backout sphere. If such a child fails, its parent must backout as well, if it commits successfully, its parent should take the proper measurements in order to remember this commitment. So if the parent fails after the commitment, it must either desire a further execution of the committed child or be able to avoid a restart. Under this combination of attribute values, the child transaction must synchronize against its parent.
- COMMIT, NOBACKOUT, NOSYNC. The child may commit independently but not backout. As in the second combination, synchronization against its parent is not required.
- NOCOMMIT, BACKOUT, SYNC. The created child commits together with its parent, possesses its own backout sphere and must synchronize against its parent. A backout of the child does not require a backout of its parent, however, the backout of the parent implies the backout of the child (more details to be given later on).
- NOCOMMIT, BACKOUT, NOSYNC. The same as above, however, the child must not be synchronized against its parent and may use the locks of its parent (if on the same level of abstraction).
- NOCOMMIT, NOBACKOUT, SYNC. The created child commits and backs out together with its parent, it must synchronize against its parent.
- NOCOMMIT, NOBACKOUT, NOSYNC. The same as above, however, no extra synchronization is necessary for data elements accessed by both the parent and the child.

This shows, that all eight possible combinations make sense. In the case of distributed transactions there would be one additional attribute (values: LOCAL, REMOTE) and sixteen combinations would make sense /Wa84/.

PROTOCOLS

In this chapter a locking protocol will be described that provides the suitable synchronization between and within transactions. Additionally, the protocols will be described for initiating, committing and aborting transactions and child transactions.

The locking protocol we use is a consistent two-phase locking protocol /Es76/, resources must be locked before they are accessed, all locks are held until commitment such that a transaction cannot set new locks after the first unlocking has occurred. The protocol is in several aspects similar to the one described in /Mo81/, however, our proposal provides more flexibility and is also suitable for child transactions that commit independently and permits a more efficient passing of locks between child transactions (in /Mo81/ child transactions must pass locks explicitly via their next common ancestor).

Since the purpose of this paper is to discuss transaction structuring rather than transaction synchronization, we will present a rather simple method for synchronization. We will assume that all locks are set on the page level and that the only lock modes are SHARED and EXCLUSIVE in their usual meaning /Es76/. For a discussion of more general lock modes and locking at various level of abstraction the interested reader is referred to the literature e.g. /Ko83, Sc83, We84/ but also to a short discussion given in a later chapter of this paper.

A transaction can either hold a lock or retain a lock. 'Holding a lock' is defined in its usual meaning, 'retaining a lock' is defined as in /Mo81/. A child transaction that holds a lock and finishes its processing (a more precise definition of 'finish' is given later on) converts its locks from 'held' to 'retained'. A retained lock can be locked by any other transaction in the same commit sphere but not by a transaction of a different commit sphere. If a transaction T1 sets a lock L (starts to hold a lock) retained by another subtransaction T2, then the association between T2 and L stops (now T2 neither holds nor retains L).

In order to enable a transaction to determine whether the holder or retainer of a lock belongs to the same commit sphere, the ID (identifier) of a transaction consists of two parts:

- The first part consists of the unique ID of the top level transaction (unique within the context of all other top level transactions in the system) plus the concatenated numbers of the transaction's other ancestors. For this purpose each transaction numbers its children consecutively in the order 1, 2, ... (see for instance the numbering in the example of Fig.3). This part is similar to the IDs used in /Li84/.
- The second part contains for each ancestor the values of the three binary attributes concerning synchronization and commit and backout spheres.

Now a transaction is able to determine whether some other tansaction that holds or retains a lock belongs to the same commit sphere or not.

The protocol for a transaction T, that requests a lock on a data element D is given in Fig. 8:

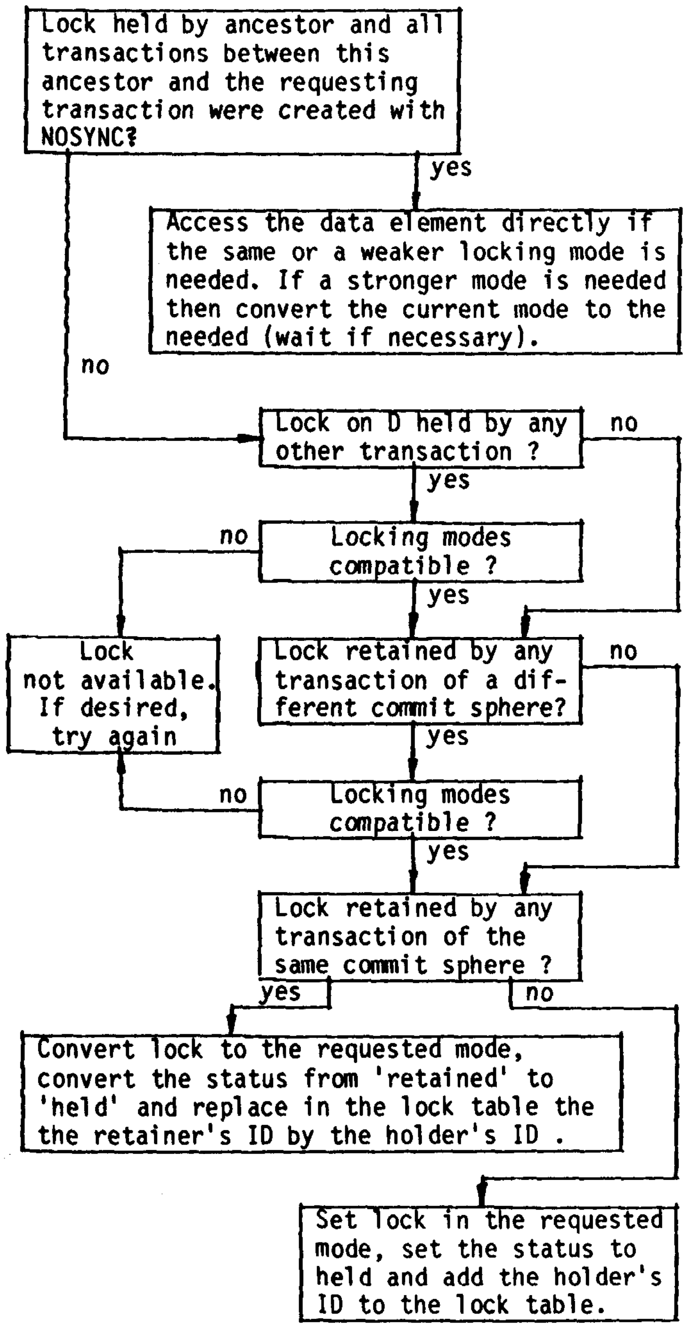
Lock held by ancestor and all transactions between this ancestor and the requesting transaction were created with NOSYNC?
yes
Access the data element directly if the same or a weaker locking mode is needed. If a stronger mode is needed then convert the current mode to the needed (wait if necessary).
no
Lock on D held by any other transaction ?
no
yes
no
Locking modes compatible ?
yes
Lock not available. If desired, try again
Lock retained by any transaction of a different commit sphere?
no
yes
no
Locking modes compatible ?
yes
Lock retained by any transaction of the same commit sphere ?
yes
no
Convert lock to the requested mode, convert the status from 'retained' to 'held' and replace in the lock table the the retainer's ID by the holder's ID .
Set lock in the requested mode, set the status to held and add the holder's ID to the lock table.

Fig. 8: Locking protocol

In /Mo81/ a similar locking scheme was proposed, however, children are not permitted to use the locks of their ancestors and locks that are passed from child to child are forwarded along a path via a common ancestor. In a later proposal /Mo82/ no locks are retained, but children may use the locks of their ancestors. Our proposal includes both methods, supports a direct passing of locks from child to child and supports multiple commit spheres.

The scheme of /Mo81/ was shown to be correct with respect to serializability in /Be83/. To show the correctness of our proposal we have to show three things:

- The notion of multiple commit spheres for one nested transaction does not influence our locking scheme. In fact, concerning synchronization each commit sphere is handled as a separate transaction, such that we can consider the locking scheme as if we had several transactions with single commit spheres.
- Our passing of locks between children can be mapped to the passing of locks in /Mo81/. In fact, in our proposal the IDs of the retainer and the new holder of a lock contain enough information that permits to look for common ancestors. Only if such a common ancestor is found in the same commit sphere, the lock can be converted. Hence, the semantics are the same in both proposals, however our passing seems to be more efficient to implement (a similar scheme as ours is used in /Li84/).
- The use of locks of the parent does not violate serializability. If a child is called synchronously and if it accesses the same elements as its parent, then this is absolutely the same as if the parent itself had performed this access. If the child is called asynchronously it accesses not the same data elements as its parent, no concurrency exists. So if the parent transaction uses SYNC and NOSYNC properly, then nothing wrong can occur.

Of course, as the scheme proposed in /Mo81/, our locking scheme is subject to deadlock, however, a detection algorithm similar to the one discussed in /Mo81/ would be suitable for our system as well.

Before we can define the various protocols for creating, committing, and backing out transactions, we have to add some details about the maintenance of transaction states. For the purpose of robustness we require the maintenance of a transaction state table (TST) in stable storage. This table contains two entries for each transaction in the system, the transaction's ID (including the creation attributes of its ancestors as described above) and the transaction's current state.

A) Creation of Transactions

When a new top level transaction is created a unique identifier is generated and stored in the TST together with the status ACTIVE.

When a transaction creates a child, then the ID of this child is composed out of the parent's ID, the attribute values defining the relationship between child and parent, and a number that uniquely identifies this child within the set of all children of its parent. The creation of a child with NOBACKOUT extends the parent's backout sphere, whereas the creation of a child with NOCOMMIT extends the parent's commit sphere.

B) Finishing and Preparing Child Transactions

When a child has completed its execution it either 'finishes' or 'finishes and prepares'. To finish means to convert all locks from held to retained. To 'finish and prepare' means to convert the locks and to write all updates into stable storage. A child that finishes, prepares separately at a later point of time (on request of its parent). We have made this distinction for efficiency reasons. If for instance the same data element is subsequently updated by several descendants of the same top level transaction, we do not require that the changes must be stabilized after each update. Of course, if one of the later updaters backs out, this might have the effect, that already finished updaters must back out as well. Our locking strategy guarantees that all such updaters belong to the same commit sphere.

Each transaction can only finish (finish and prepare, prepare) if all its descendants have finished (finished and prepared, prepared) as well. A transaction that has finished forwards the IDs of all its dependent descendants to its parent. A transaction that has finished changes its stable state to FINISH, if it has finished and prepared (prepared), the state is changed to PREPARED.

C) Committing Transactions

As already indicated in a previous chapter, all transactions belonging to the same commit sphere commit atomically together. The commitment of an inner sphere can be done earlier than the commitment of an outer sphere, the commitment of an outer sphere implies the commitment of its inner spheres (if not already done). Note that commitment also might cause a backout sphere to shrink.

The coordinator of the commitment writes a commit record into stable storage and then issues a commit command for all transactions in the sphere to be committed. The coordinator is either the top level transaction or the parent that has created the independent child that is the highest ranking transaction in its sphere to be committed.

D) Backing Out Transactions

Backing out is arranged in the same way as the commitment, i.e. a coordinator sends the backout commands to the transactions to be backed out. The transactions of a backout sphere are either backed out because of a logical error, during restart after a processor failure, or because of a unilateral decision of the coordinator. Note that if finishing and preparing is done separately, the backout of transactions of one backout sphere may imply the backing out of transactions of some other backout sphere as well.

Backing out also means that the data is restored to a previous state. If we assume, that each time a data element is updated, a new version is created, that is written into stable storage when the corresponding transaction prepares, then we always have the appropriate earlier states. Other techniques for doing so are well known and will not be discussed in this paper.

E) Commands

In our system the following commands can be used for handling transactions (TA stands for transaction):

- CREATE-TA (CHILD-ID) returns (STATUS). Remember, that the CHILD-ID contains the PARENT-ID as well as all needed attribute values, STATUS returns whether the call was successfully performed or not.
- FINISH-TA (CHILD-ID) returns (STATUS).
- PREPARE-TA (CHILD-ID) returns (STATUS).
- COMMIT-TA (CHILD-ID) returns (STATUS).
- BACKOUT-TA (CHILD-ID) returns (STATUS).
- REQUEST (CHILD-ID, TASK) returns (RESULT, STATUS). TASK describes the function the child should perform, RESULT includes the returned data.

Additionally these elements can be used in certain combinations. For a transaction consisting of just one request with a separate commit the combination CREATE-REQUEST-FINISH-PREPARE-TA can be used in order to reduce the number of context switches in an implementation and the number of state changes of the child transaction. If the interactions between parent and child are conversational, then several REQUEST-TA would be sent separately. FINISH-PREPARE-TA would cause the corresponding child to finish and prepare as described above.

An interesting combination is the following: CREATE-REQUEST-FINISH-PREPARE-COMMIT-TA (models the single-request interface discussed earlier) reduces the interaction between parent and child to a minimum. The important characteristic of this case is, that a child needs never to wait for its parent. This is important in the context of deadlocks (see also in the next chapter).

We have called the above primitives 'commands' for two reasons:

- Any action a child performs is only done on request of its parent. If the child fails logically, a corresponding code will be returned to its parent, which then can give the BACKOUT-command.
- We do not want to enforce a special implementation by calling the primitives 'calls' or 'messages'. Of course, asynchronous message passing would provide the highest flexibility, however, other implementations are possible as well.

F) Interdependencies between Commit Spheres and Backout Spheres

Since commit and backout spheres are defined independently, two transactions may belong to the same commit sphere but to different backout spheres and vice versa. In the following the various types of interdependencies will be discussed that may occur between an arbitrary parent transaction PT and one of its child transactions, say CT. The discussion is based on the assumption, that whenever PT tells CT to commit, then the effects of CT will never be compensated whatever happens. An extension of our model that additionally supports the opposite strategy, i.e. that compensation may be possible, is discussed later on (in the context of transaction structures for layered applications with synchronisation at multiple levels of abstraction). It should be clear, that CT only performs actions on request from PT.

Before the discussion can be started, it is necessary to remember the various stable states of a transaction:

- ACTIVE: The processing is in progess, updates may already have been done.
- FINISHED: The processing phase is finished, however, the updates may still be in volatile memory and are not available for transactions in other commit spheres. FINISHED can only be reached from state ACTIVE.
- PREPARED: Similar to FINISHED, but all updates have been transferred to stable storage. PREPARED can be reached from state FINISHED or directly from ACTIVE.
- COMMIT: This state can only be reached via the state PREPARED. Its meaning is, that actions can be taken to make the updates available for transactions in other commit spheres.
- BACKOUT: This state can be reached directly from the states ACTIVE, FINISHED, and PREPARED. It's meaning is, that actions must be taken to undo all the transaction's updates.
- UNKNOWN: This state can be reached via COMMIT and via BACKOUT, it is also the state of each transaction before it enters the state ACTIVE. UNKNOWN is an implicit state, i.e. all transactions not explicitly noted to be in one of the previous five states, are assumed to be in the state UNKNOWN.

In principle, four types of interdependencies between PT and CT are possible:

- PT and CT belong to the same commit sphere as well as to the same backout sphere. This is the trivial case, where either both commit or both backout.
- PT and CT belong to the same commit sphere but to different backout spheres. If PT commits (backs out) then CT must commit (backout) as well. However, as long as PT is in the state ACTIVE, CT may be backed out independently of PT. When PT enters the state FINISHED, then CT must be prepared as well and CT may only backout on request of PT. To understand this strategy, it must be considered, that when PT is in the state FINISHED, it has already left its processing phase and hence is not able to redo the original requests to CT any more. So, in the latter case the only possibility to restart CT would be to restart PT.
- PT and CT belong to different commit spheres and to different backout spheres. CT may be committed and backed out independently on request

of PT as long as PT is in the state ACTIVE. Additionally, if PT backs out before issuing the COMMIT-command to CT, CT must back out as well, since in this case no one would exist any more for telling CT what to do. If, however, PT backs out after having issued the corresponding COMMIT-command, it has no effects on CT, no matter whether a pure COMMIT-command or a combination of commands including COMMIT was used. Since CT was created in a different COMMIT-sphere PT must issue the COMMIT-command before leaving the ACTIVE-phase (otherwise it would not have made sense to run CT in a separate commit sphere).
- PT and CT belong to different commit spheres but to the same backout sphere. This case is similar to the previous case with the exception, that a backing out of CT always enforces PT to back out as well.

Note, that we do not permit that CT commits later than PT (provided, that PT commits at all). Of course, it would technically be possible, that PT just tells CT what to do and to commit before CT has finished. However, in this case there would be no possibility for detecting a failure of CT.

If we also want to permit compensation in our model, i.e. that transactions might be backed out after commit, then anything would be the same as above, with the exception, that the backout semantics of a parent transaction would be different. As discussed earlier, in this case a backing out parent transaction must create new child transactions for compensating the effects of earlier child transactions. So, the above strategies are also suitable for systems with synchronization at various levels of abstraction.

G) Restart

Since backout and commit spheres can overlap, a restart takes place in two phases:
- In the first phase all transactions are checked for commitment. As indicated in a previous chapter, the coordinator of a commitment writes a commit record into stable storage before issuing the commit command. After restart for each commit sphere it is checked whether the commit record was already written before the crash. If this is the case, the transactions in the corresponding commit sphere may commit. Remember, that commitment might cause backout spheres to shrink.
- In the second phase all other transactions not in the state PREPARED are backed out and restarted. If the time between crash and restart was too long all transactions can be backed out (e.g. on request from the enduser).

IMPLEMENTATION

Due to space limitations we are not able to give a complete description of the implementation of the above concepts, however, we will discuss a few more important questions:
- Should any application use its own transaction management? In theory this might be possible, provided that all private transaction managers understand the same language. In practice this solution would be rather inefficient:
  -- Each new application would require the reimplementation of practically the same set of management functions.
  -- During a restart complex interactions between the various transaction managers would be necessary in order to coordinate their actions.

  For these reasons we favor a solution where all application use a common so-called transaction kernel (the design of a very basic kernel was for instance discussed in /Ro84/).
- Can the various applications use their own locking and scheduling strategies? At a first sight, this question seems to be answerable in the same way as the previous, especially, since the handling of deadlocks in a decentraliced environment is considerably more complex than in case of a centralized lock management. However, it has been demonstrated, that concurrency can be increased if more semantic information is used for scheduling (see for instance Sc83/) and semantic information is only available in the application system itself. Our opinion is, that this is still a subject of further research, however, it might be a good idea to follow the concept of the transaction kernel in /Ro84/ and to include the basic primitives in a kernel, but to let all strategies and hence the semantics in the application. Of course, it must be guaranteed, that each application provides the same scheduling strategy or that some global deadlock detection and handling mechanism exists.

A further interesting question is, whether it would be possible to integrate existing applications systems, i.e. a database system into a structured application system. The main problem is, that an existing system is in most cases something like a black box, i.e. we are not quite sure about its scheduling strategies etc. So if a higher level transaction starts a child transaction in this existing system, then this can lead to deadlock situations.

A deadlock can exist if there is the possibility of a circular wait. Assume two transactions T1 and T2 belonging to the same higher level application a_1. Assume further that both create a child on the lower level application a_2, then, if both use private schedulers, a_1 might schedule T2 after T1 and a_2 might schedule T1 after T2. Now, if T1 waits for a response from its child in a_2 and T2's child waits for the next request from T2, then a deadlock could occur in this situation.

If, however, transactions in a_1 only use calls of the type CREATE-REQUEST-FINISH-PREPARE-COMMIT-TA for running child transactions in a_2, then, as already mentioned earlier, no deadlock can occur.

CONCLUSION

Nested transactions with multiple commit points have been presented as a general method for structuring advanced database oriented applications in a centralized environment. Because of space limitations the version of this concept for distributed database applications is discussed in a separate paper /Wa84/. The main differences of the distributed version are:

- Child transactions can execute at local as well as at remote sites, hence, a fourth attribute with the values LOCAL and REMOTE is necessary in order to describe the relationship between parent and child transactions.
- Applications can be distributed over multiple sites of a network, hence we have a further nesting of transactions within an application.
- The sites of a network can fail independently and hence also the transactions of a distributed nesting hierachy. Hence, the protocols for commitment and recovery are more complex than in the centralized case. Additionally, the orphan problem must be considered. An orphan is a child transaction, whose parent transaction has backed out without being able to inform the child (see also /Al83, Li84/).

The proposed model of nested transactions is more general than any existing proposal. Without considering distribution aspects, the existing models can be modelled in our concept as follows:

- The proposal in /Mo81/ included the attribute values BACKOUT, NOCOMMIT, SYNC. A latter proposal of the same author /Mo81/ included NOSYNC instead of SYNC. The proposals are not suitable for conversational interfaces.
- The ARGUS system /Li83, Li84/ additionally supports COMMIT.
- In /Mu83/ the nested transactions of the LOCUS system are presented. The parent/child relationship is described by either BACKOUT, NOCOMMIT, SYNC (the normal call of a child transaction in LOCUS) or NOBACKOUT, NOCOMMIT, SYNC/ASYNC (a child transaction call based on the 'fork'-primitive of the underlying operating system). The first type of call is not suitable for conversational interfaces.
- In /Al83/ a more general model was presented for transactions over arbitrary objects, however, it does not include the attribute value COMMIT.
- In /We84/ a system with synchronization at multiple levels of abstraction is discussed, which supports the combination COMMIT, BACKOUT, NOSYNC.

Our proposal supports the construction of transactions with arbitrary internal parallelism.

REFERENCES

/Al83/ Allchin, J.E., "An Architecture for Reliable Decentralized Systems", Ph.D. Thesis, Georgia Institute of Technology, GIT-ICS-83/23, 1983.

/Be83/ Beeri, C., P.A. Bernstein, N. Goodman, M.Y. Lai, D.E. Shasha, "A Concurrency Control Theory for Nested Transactions", Proc. 2nd Symp. on Principle of Distributed Computing, 1983.

/Da73/ Davies, C.T., "Recovery Semantics for a DB/DC System", Proc. 28th ACM Nat. Conf., 1973.

/Es76/ Eswaran, K.P., J.N. Gray, R.A. Lorie, I.L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System", CACM 19:11, 1976.

/Ko83/ Korth, H.F., "Locking Primitives in a Database System", JACM 30:1, 1983.

/La81/ Lampson, B., H. Sturgis, "Crash Recovery in a Distributed Data Storage System", Technical Report, XEROX PARC, 1979.

/Li83/ Liskov, B., R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", ACM TOPLAS 5:3, 1983.

/Li84/ Liskov, B., "Overview of the ARGUS Language and System," Programming Methodology Group Memo 40, MIT, 1984.

/Mo81/ Moss, J.E.B., "Nested Transactions: An Approach to Reliable Destributed Computing", Ph.D. Thesis, MIT-LCS TR-260, 1981.

/Mo82/ Moss, J.E.B., "Nested Transactions and Reliable Distributed Computing", Proc. 2nd Symp. on Reliability in Distributed Software and Database Systems, 1982.

/Mu83/ Mueller, E.T., J.D. Moore, G.J. Popek, "A Nested Transaction Mechanism for LOCUS", Proc. 9th ACM Symp. on Operating Systems Principles, 1983.

/Ro84/ Rothermel, K., B. Walter, "A Kernel for Transaction Oriented Communication in Distributed Database Systems", Proc. 4th Int. Conf. for Distributed Computing Systems, 1984.

/Sc83/ Schwarz, P.M., A.Z. Spector, "Synchronizing Shared Abstract Types" (Revised Issue), Technical Report, Carnegie-Mellon University, CMU-CS-83-163, 1983.

/Sc83/ Schneider, F.B., "Fail-Stop Processors", Proc. COMPCON'83, 1983.

/Wa84/ Walter, B., "Nested Transactions with Multiple Commit Points for Structuring Advanced Distributed Applications", Working Paper (to be submitted for publication), University of Stuttgart, 1984.

/We84/ Weikum, G., H.-J. Schek, "Architectural Issues of Transaction Management in Multi-Layered Systems", Proc. 10th VLDB, Singapore, 1984.