Cooperative Transaction Hierarchies

Marian H. Nodine, Mary F. Fernandez
Andrea H. Skarra and *Stanley B. Zdonik*

Technical Report No. CS–90–03
February 1990
Revised: February 1991

Cooperative Transaction Hierarchies

Marian H. Nodine, Mary F. Fernandez,
Andrea H. Skarra and Stanley B. Zdonik
Brown University
Providence, RI 02912

February 28, 1990
Revised: February, 1991

Abstract

In this paper, we examine some of the requirements designers place on databases. Based on these requirements, we define a new transaction framework called a *cooperative transaction hierarchy*. This framework allows us to relax the criterion that transactions be atomic, while restricting the effect that this has on synchronization and recovery.

Patterns and *conflicts* specify the constraints imposed on an operation history for it to be correct. We use a type of augmented finite state automaton, called an *operation machine*, to enforce user-defined operation patterns. The patterns are also used in computing dependency chains among operations on the objects in the database. When an operation fails, we use these chains to invalidate any operations which are dependent on the failed operations. We can also define conflicts in the context of these patterns. We present a method for detecting and invalidating conflicts in the new history produced during recovery. We define a set of operations that cooperating transactions can use when recovering forward once the effects of all invalid operations have been removed from the history. We show that the operation history remains correct even after recovery from a failure.

1 Introduction

Databases were originally developed to support on-line data processing for commercial institutions such as banks and airlines. Transaction processing for these applications is characterized as follows:

- High transaction throughput;
- Short, independent, and atomic transactions;
- Serializable transaction histories; and
- Log-based recovery.

With the advent of more semantically-rich database models, the number of applications for database management systems has increased. In particular, design applications such as CAD/CASE tools, hypermedia systems and interactive programming environments all share a common requirement for a good, flexible, underlying database.

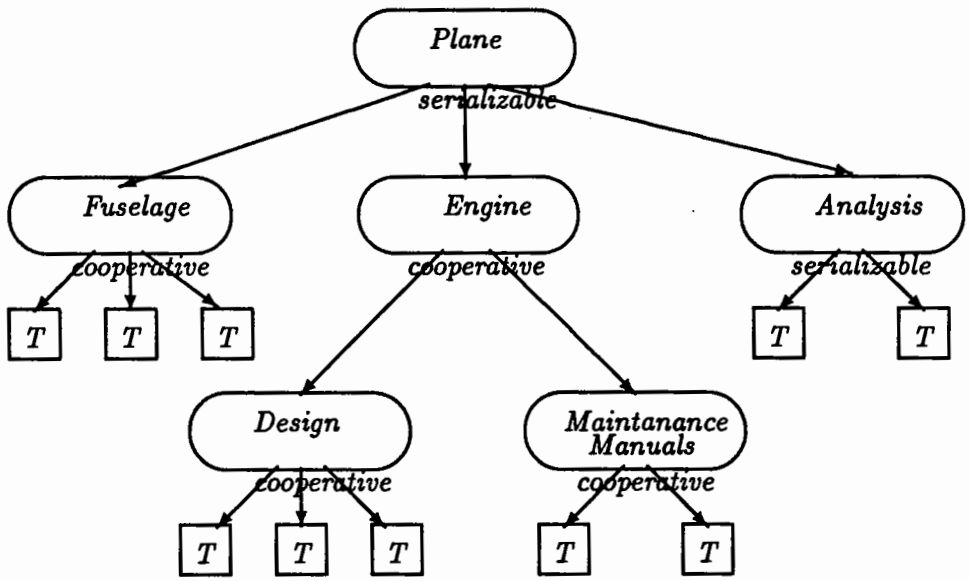


Figure 1: Groups of design applications

As an example of a design application, consider an environment in which groups of people are working on the design of an airplane (see Figure 1). These people naturally divide into three groups. There are two groups of actual designers, working on the engine and fuselage design, respectively. The third group consists of testers who test and analyze the designs. The engine group subdivides naturally into two more groups, one containing engine designers and another containing the technical writers working on the engine maintenance manuals. The fuselage group is composed solely of designers. The analysis group uses engine and fuselage data as input to structural analysis programs.

Depending on the interaction of the design applications, different notions of correctness and operation synchronization may be applicable at different places in the airplane design hierarchy. For instance, the structural analysis programs should not execute until the objects created by the engine and fuselage groups are complete. Therefore at the *plane* level, serializability of the groups' operations should be enforced. In the *Engine Design* group, different designers should be able to see each others' work in progress. This means that they may want to share changes with each other before releasing those changes to be seen by the *Maintenance Manuals* group. Furthermore, more than one designer may need to change a specific part of the engine (e.g. the propeller shaft may be modified by both the person working on the propeller and the person working on its coupling to the engine). Therefore, engine designers may need to allow multiple writers on objects. Also they may require the ability for a designer to read intermediate changes made by other designers. Furthermore, the *Maintenance Manuals* group should be permitted read-only access to objects modified by the *Engine Design* group, but only after the engine design is complete with respect to those objects.

Looking at this example, we can begin to see some of the characteristics of design applications.

- Long, non-atomic transactions;
- Transactions operate on non-uniform objects which are related in complex and arbitrary ways;
- Transactions (and users) form natural, interactive groups;
- Groups of transactions may need to tailor their correctness criteria and synchronization protocols to their own needs; and
- Transaction histories are not necessarily serializable, therefore recovery is more difficult.

The requirement to support non-uniform objects seems to indicate that an object-oriented database would be the most useful in this context. The underlying object server must use a transaction model that supports the requirements for hierarchically-structured, cooperating transactions. The transaction model should capture the hierarchical grouping of design activities and provide the ability for each group to choose its own synchronization mechanisms (including serializability). Furthermore, the model should enable application builders to concisely specify the view of correctness of each group of transactions, and provide mechanisms for mapping between nested sharing protocols so that the overall integrity of the database is preserved.

2 Related Research

The need for a transaction model that supports cooperative design activities was recognized during the development of complex, integrated CAD and CASE tools. Traditional databases supported a model of short duration, serializable transactions. The application of this model to the long lived, interactive and distributed transactions generated by design tools was difficult. This encouraged the development of an extended transaction model to support both types of database transactions.

2.1 Hierarchical Organization of Transactions

There are several transaction models that are based on a three-level hierarchy. These include [KLMP84, KSUW85, LP83]. The three layers are generally defined as follows:

- a *public* layer where objects can be accessed by any application or group,
- a *group* or *semi-public* layer where sets of users can store shared objects, and
- a *user* or *private* layer where individual transactions store their own copies of objects.

In [KSUW85], transactions may only "belong" to one group or have access to one group database. Thus, the structure of the database resembles a tree with the public database at the root, the group databases as children of the root and the user transactions as children of the groups. In [KLMP84], transactions are permitted to "check out" objects from any other transaction thus producing a *dag* of cooperating transactions.

Since design transactions do not always naturally decompose into a three-level hierarchy, models that allow arbitrary nesting may be more useful. Nested transactions [Mos85] allow hierarchies of arbitrary depth. In this model, a transaction may spawn a set of *subtransactions* to help complete its task. Subtransactions in turn can spawn other subtransactions, resulting in a hierarchy. A subtransaction's commit or abort is relative to the parent. Aborted subtransactions may not cause the parent to abort; other options are allowed. However, at each level, the subtransactions obey the basic constraints of atomicity and serializability. Also, in contrast to the transaction hierarchies described above, nested transactions only keep a single copy of every object in the root database. This copy is shared by all the threads of execution corresponding to active transactions and subtransactions.

Some interesting recovery mechanisms have been proposed for nested transactions. Moss originally proposed a version-based recovery method, where each subtransaction saves before-versions of the objects it modifies. If the subtransaction fails, these versions are restored. The ARIES/NT recovery method [RM89] developed at IBM uses write-ahead logging for nested transaction recovery. With write-ahead logging, a record of changes is written to a stable log *before* the actual updates are written to the disk. After a failure, the recovery algorithm first redoes the operations in the log that are not reflected in the object on the disk, then undoes all operations by incomplete transactions. ARIES/NT also supports save points.

Several proposals for relaxing serializability of subtransactions for design applications have been proposed. In their three-level hierarchy, Klahod et.al. [KSUW85] allow serializability to be relaxed among the user transactions cooperating in a single group transaction.

Only one user transaction can be active in a group transaction at a time, but the user transactions can voluntarily suspend operations at points where other user transactions may need to interleave their operations. Thus, a user transaction can access the intermediate work of other user transactions in its group transaction. However, this scheme does not guarantee that the data remains consistent.

Korth et.al. [KKB87] also define a hierarchical transaction structure for CAD databases that allows serializability to be relaxed at lower levels. The top three levels of the hierarchy are similar to the *public*, *group*, and *private* databases from the three-level organization above. Group databases maintain all of the information associated with a specific project. The private databases may be accessed by a set of *client/subcontractor* transactions, which may be nested to an arbitrary depth and have relaxed correctness criteria. These transactions are typically broken up into a sequence of short-duration transactions which must be atomic. The sequence and interleaving of the short-duration transactions must correspond to the consistency constraints defined for the underlying database. Korth et. al also define a new locking mechanism, called *predicatewise two-phase locking* to allow more flexibility.

2.2 Transaction Models with Relaxed Constraints

ObServer [SZR86], supports non-restrictive read and non-restrictive write locks to allow arbitrary interleaving of operations. They do not allow the user to restrict the interleaving of operations in any way. As a result, the visibility of intermediate objects cannot be controlled, so undesirable sharing of intermediate versions of objects can occur. Also, aborting and recovery can affect an arbitrary number of objects in the database. Similar locking techniques are also used in [KSUW85].

Sagas, proposed in [GMS87], provide a two-level breakdown for long-lived transactions. A *saga* is defined as a sequence of atomic transactions $\{T_1, T_2, ..., T_N\}$. Compensating transactions $\{C_1, C_2, ..., C_N\}$ are also defined for each transaction in the saga. Executing a saga means executing each of its transactions in sequence. If transaction T_i fails, the saga recovers by executing the corresponding compensating transaction C_i. The saga then can continue to abort by executing the rest of the compensating transactions in inverse order $\{C_{i-1}, ..., C_2, C_1\}$, or it can attempt to continue by restarting the saga at T_i. Atomicity is not preserved at the saga level, because sagas may interleave their transactions. This scheme assumes that the transaction sequences and the compensating transactions for a saga can be easily defined.

[HR87] mentions some work done at IBM on *conversational interfaces* for nested transactions. A conversational interface allows a subtransaction to interact with, and release information to, its parent. This work attempts to preserve the atomicity of all subtransactions, and thus concludes that too much of the database would have to be undone after an abort for this to be a practical approach.

[Lyn83] contains an abstract model for specifying multiple levels of transactions, called *multilevel atomicity*. Lynch provides a framework in which serializability is relaxed to include a broader set of allowable executions. These executions are specified by a set of k equivalence relations defining how much each pair of transactions can interact, and a k-level breakpoint specification which defines the places in a transaction execution where other transactions can interleave their operations. This model does not require that groups interleave their

operations in a strictly serializable order nor does it restrict the depth of transaction groupings. Lynch assumes that groups of transactions can predeclare a partial ordering of their operations based on the semantics of the application. These partial orders are used to determine the correctness of multiple levels of interleaving. Lynch also assumes that the set of transactions in the system is fairly static, in that adding a new transaction requires a certain amount of manual intervention.

Ellis and Gibbs [EG89] define a concurrency control model for groupware systems such as hypermedia. This model uses neither locks nor nesting, but instead uses operation transformation to preserve consistency at different sites. This mechanism is fast and flexible, and allows for serializable operations. However, defining the operation transformations may become difficult as the underlying operation set becomes more complex.

Skarra [Ska89] defines a concurrency control model for cooperative transaction hierarchies in object-oriented databases. She assumes the underlying operation set is defined by the methods on the object type. Concurrency control is done within a group of cooperating transactions. These transactions define allowable sequences of operations, called *patterns*, and unallowable operations within those sequences, called *conflicts*. Some of Skarra's work serves as a basis for the work done here.

2.3 Communication

A characteristic of design activities not addressed by these approaches is communication. Cooperative behavior requires primitives for communicating about shared objects and for defining extensible access patterns. Because the transactions cooperate by sharing objects, the communication mechanism should provide information on the use of shared objects. ObServer [SZR86] supports *notifications* inherently in its locking model. This allows restricted communication about shared objects. None of the other previously mentioned approaches provide primitives for communicating about shared objects.

3 The Transaction Hierarchy

3.1 Conceptual Overview

There are several characteristics that distinguish design applications from traditional applications. The specific application programs used in design are interactive and heterogeneous. Interactive programs often access and modify a small set of objects multiple times, and these operations often occur in predictable sequences. For example, when editing a file, the text editor first reads the file, then may write it multiple times.

Another difference is that, while the individual designers that manipulate the objects are autonomous, they naturally form groups that interact both within and outside of the database. Interactions within the database usually happen in one of the following ways:

- the designers access the same set of objects,
- the database notifies designers when objects they are interested in have changed, or
- the database interacts with the designers to negotiate conflict.

Interactions outside the database happen through direct or indirect personal communication.

Although designers work cooperatively on a design activity, they are autonomous. They do not necessarily use the same application programs, work on the same machine, or follow the same schedule. The entity that binds together different designers' work is the shared data, and thus the locus of sharing in these environments is the database. By providing an extensible model of transactions at the object server level, the groupings and interactions of the designers can be defined and controlled in a more flexible way than in systems that ascribe to a strictly serializable transaction model.

We propose a hierarchical scheme for specifying the logical grouping of cooperating transactions in an object server. A *transaction group* [FZ89] contains of a set of members that cooperate to do a single task. It actively controls the interaction of its cooperating members. A *transaction group member* may be either an individual transaction or another transaction group. No member may have more than one parent. Thus, transactions and transaction groups may be nested in any tree-like manner to form a *transaction hierarchy.* Figure 1 in Section 1 is an example of a transaction hierarchy.

Because we can not determine the clustering of transactions a priori, the transaction hierarchy can be constructed and modified dynamically. The root transaction group, which maintains the database directly, always exists. Other transaction groups and transactions may join existing transaction groups as a design effort progresses, modifying the transaction hierarchy as needed. However, some sort of authentication procedure should exist for joining a transaction group to prevent unauthorized access to any internal information it maintains.

The properties of transaction groups and transactions are defined in more detail in the sections below, and are summarized in Figure 2.

3.1.1 Transaction Groups

The procedures and rules defining the operation of a specific transaction group are called its *protocols.* Protocols define how the members of a transaction group can interact. A

Transaction Group

- *Corresponds to some task*
- *Enforces correctness among its members*
- *Directs communication among its members*
- *Unit of recovery*
- *Not Atomic*

Transaction

- *Collection of read and write operations*
- *Not Atomic*

Operation

- *Read or write of an object by a member*
- *Atomic*

Figure 2: Properties of transaction groups, transactions, and operations.

transaction group's *internal protocols* specify the allowable interactions among its members. Its *external protocols* specify how the transaction group may interact with its siblings in the transaction hierarchy. At any level in the tree, the external protocols of a transaction group member must be the same as the internal protocols of its parent.

Associated with each transaction group is a *cache* containing the set of objects that are being accessed by its members currently. At the root of the transaction group hierarchy, the cache is the database itself. An object is read automatically to a member transaction group's cache when the member initiates a successful read operation on that object. An operation by a member succeeds when it is allowed by its parent transaction group's internal synchronization protocols. An object is written back to the parent transaction group's cache when the member indicates that it has finished modifying the object. Thus, a transaction group appears to its members to be the database server responsible for storing objects, controlling member access to objects, notifying members of object use and handling recovery.

Because the members of a transaction group cooperate, they are no longer self-contained. This means that they may not individually produce correct operation histories. That is, the sequence of operations by a single member may not leave the database in a correct state. The transaction group must ensure that the combined history produced by all of its members is correct. A transaction group's internal protocols specify the notion of correctness that it enforces, among other things.

A transaction group is also the basic recovery unit in the database. When a member of the transaction group fails or aborts, the transaction group coordinates recovery among the other members. The recovery process ensures that the transaction group's cache is left in a correct state, i.e., one that could be reached by some sequence of operations allowable by the transaction group's internal synchronization protocols. The transaction group's internal

protocols also may be used to tailor the recovery process to the specific needs of the group.

When a transaction group *(M)* is itself a member of another transaction group *(P)*, it must translate the combined history of its members, which conforms to its internal protocols, into an equivalent history[1] compatible with its external protocols. In this way, each level in the transaction hierarchy adheres to its own view of correctness. The translating function also hides the internal operations of the transaction group members from its parent, generally by collapsing sequences of operations by the members into shorter sequences of operations by the transaction group itself. This allows encapsulated groups of non-serializable transactions to be members of other serializable groups.

3.1.2 Transactions

Because transactions may cooperate with other members of their transaction group, they are no longer atomic. Furthermore, they no longer need to enforce consistency within themselves; neither are they directly responsible for recovery. Transactions in this model are viewed instead merely as collections of atomic read or write operations associated with some specific task.

3.2 Implementation Overview

This section describes how members of a transaction group are created, how they make changes to the database, and how they terminate. These operations are similar in spirit to the familiar transaction *begin, commit,* and *abort* operations. However, there are also significant differences caused by the different requirements that cooperative applications such as design applications place on the object server.

A member is created using the *member_begin* command. Once a member has been established, it may operate on the database in any way allowable by the concurrency control mechanism defined by its transaction group's internal protocols. A member may make its operations recoverable[2] using a *member_checkpoint* command. This means that the results of the set of operations are known to the member's parent transaction group, and consequently become visible[3] to the parent transaction group's siblings. A member may revoke one or more of its operations using the *member_abort* command. This means that the effects of the specified operations are removed from the database, potentially causing other dependent members of the transaction group to initiate some form of recovery. Neither the *member_checkpoint* operation nor the *member_abort* operation terminates the member. A member executes one or more several operation sequences, each followed by a *member_checkpoint* or a *member_abort*, before finally terminating. The *member_terminate* command removes the member from the

[1] An equivalent history is either an identical history or one where the operations by the members of *M* on the object copies in *M*'s cache are collapsed into a shorter sequence of operations by *M* on the object copies in *P*'s cache that have the same effect, as defined by the supplied mapping from *M*'s internal protocols to its external protocols.

[2] When applied to an operation, recoverable means that there is some way of retrieving any changes resulting from the operation if the member that did the operation fails.

[3] When applied to an operation, visible means that any results of the operation may be perceived. For write operations, this means that the new version may be read.

transaction group once the member's changes cannot be revoked by some other member's *member_abort* command.

The member operations differ from the traditional transaction operations *begin*, *commit*, and *abort* in two ways. First, *member_checkpoint* and *member_abort* do not terminate the member. This is because we view the member as an ongoing operator doing a long sequence of operations, each of which it may want to selectively commit or abort. A second way they differ is that they may be done by any member, not just by a leaf transaction. This is necessary because we want the operations done within the context of a transaction group to be local to that group, and not affected by members further towards the leaves of the hierarchy that may not concern it.

Using the member operations, a member could simulate a traditional committed transaction by beginning, executing a set of operations, checkpointing, and terminating. A member could simulate a traditional aborted transaction as above, but aborting all of its operations instead of checkpointing them.

3.2.1 Member Begin

Members are created using the *member_begin* command. This command specifies the new member's name and its parent. If the member is a transaction group, its internal protocols are specified as well. These indicate how the members of the transaction group interact, including information such as:

- either an enumeration of its members or a procedure for authenticating new members;
- a synchronization mechanism to control the interleaving of member operations;
- the rules for mapping internal operations (i.e., operations by the members of the transaction group on the objects in its cache) into external operations (i.e., operations by the transaction group on objects in its parent's cache); and
- the rules for handling recovery when an operation by a terminated member becomes invalid.

How to specify these in detail is defined in later sections. Additionally, the new member must authenticate itself to its parent using its parent's authentication protocols.

There are several functions that are not specified during the member definition process, but are inherent in the way the transaction hierarchy is managed. These include:

- the transaction group's external protocols, which are inherited from its parent;
- the rules for passing messages (such as those caused by a *member_abort*) to its parent and members; and
- the rules for managing the objects in the cache.

3.2.2 Member Operation

Individual read and write operations by a member are permitted or forbidden by the transaction group's internal synchronization protocols. Once the transaction group is satisfied that a sequence of operations in some way completes some set of changes that it wishes to release to its parent, it can checkpoint them using a *member_checkpoint* operation. The *member_checkpoint* operation causes the set of internal operations to be mapped into an equivalent set of external operations, possibly introducing new object versions to the parent transaction group's cache as a result. This makes the changes visible to the other members of the parent transaction group. It also makes the changes recoverable from the parent transaction group in the case where the member fails. Since the different groups in the transaction hierarchy may guarantee different levels of permanence,[4] the *member_checkpoint* procedure only guarantees that the new object versions associated with the checkpointed member are as permanent as the parent transaction group guarantees them to be.

Occasionally, a member may wish to abort one or more of its uncheckpointed operations. This means that the operation is no longer a part of the transaction group's history, and that any object version created by the operation is no longer accessible. Operations can be aborted for one of two reasons:

- the member actually failed in some way, causing its transaction group to abort the uncheckpointed operations by the member; and
- the member decided that its changes were inappropriate in some way, and aborted its own operations.

The *member_abort* operation causes a specified set of operations to become invalid. The operations do not need to be contiguous, and aborting an operation does not necessarily mean aborting all subsequent operations by the member. The abort makes it appear to the transaction group members as if those operations had never happened. However, other operations may be dependent on the aborted operations. This could occur, for example, if one of the operations created some version that was read by another member. These operations, too, must become invalid. Thus, the invalidation can propagate through the transaction group's history to all operations that are transitively dependent on the operations that were invalidated initially. Many members' operations may be invalidated, and these members must in turn work to recover as much as possible from the invalidation of their operations.

3.2.3 Member Terminate

When a member is completely finished and all its valid operations are checkpointed, the member may remove itself from the transaction group using the *member_terminate* command. However, some of the operations done by the member may be dependent on other members' operations and consequently may become invalid if one of those members aborts. When a member terminates, it relinquishes the right to recover any of the effects of its operations to its transaction group.

[4] Permanence implies the ability to recover object versions from the object server even in conditions such as system failure or storage media failure.

4 Cooperation and Its Effects

Adding the capability for transactions to cooperate and communicate in a controlled manner through the object server has some interesting effects on some of the basic database operations. For example, a single transaction can no longer be viewed as corresponding to a single task. Instead, several transactions may cooperate to complete a given task. Traditionally we speak of correctness with respect to transactions. The correctness of a task is ensured only indirectly, because the transaction that executed the task is correct. However, correctness is really a property of the task only, so in a cooperative environment it must be specified with respect to the entire set of transactions that can participate in completing the task. Thus, correctness is a property of a transaction group as a whole, not just a single transaction.

The following sections describe how cooperation affects various aspects of transaction and data management in the object server. The areas discussed include atomicity, visibility, correctness, locking, dependencies, transactions, recovery, and permanence.

4.1 Atomicity and Visibility

In a cooperative environment, it is often useful for a group of transactions to be able to see each others' objects and possibly to modify the same set of objects. This is because they are working together to do a single task, and share together the responsibility for doing the task correctly. Because it is desirable for intermediate, uncommitted versions of objects to be visible to other members, we must relax the traditional atomicity constraint on transactions to some extent.

One desirable property of atomicity was that it constrained the visibility of objects in such a way that recovery was simplified. We need to be careful how we relax atomicity in order to keep the recovery process tractable. We need to examine how cooperating transactions interact through the objects, so that we can provide for the needs of the design applications while keeping recovery as efficient as possible.

When a member writes an intermediate version of an object to the database, it is making that version available to the members with which it is cooperating directly. Other members should not be allowed to see this intermediate version, though they should have access to any final, committed version of the object. Furthermore, different versions of the object may be visible at different levels of the transaction hierarchy. For example, in the airplane design hierarchy in Figure 1 in Section 1, different versions of the engine design may be visible to the engine designers, the maintenance manual writers, and the structural analysis programs. If we examine the structure of the transaction hierarchy, we see that visibility is associated with transaction groups. Each object version in the database is associated with some transaction group, and is visible to that group and all of its descendants, unless masked by some other version of the same object further down in the transaction hierarchy. When an object version is declared final by a member, it is written to the next level up in the transaction hierarchy, and thus made visible to more members.

Relaxing atomicity is also desirable because of the length of transactions in the design environment. When transactions are long, aborting transactions is expensive. Thus, it is desirable to minimize the amount of work undone when something fails, and to try to recover in a forward direction by doing compensating operations instead.

4.2 Correctness Criteria

Traditionally, the measure of correctness of a history has been whether or not the history is equivalent to a history that would be generated by some serial ordering of the transactions (serializability) [BHG87]. The basic premise behind serializability is that each transaction should only depend on transactions that commit before it chronologically. Thus, two transactions can never depend on each others' output. A second assumption, implicit in some proofs related to serializability, is that a transaction reads and writes an object at most once during its lifetime. This assumption makes sense in a serializable database. If we look at the equivalent serial execution of the transactions, we can see that nothing can change in between two reads of the same object, and no other transaction can see the results of any write but the last one.

Unfortunately, these restrictions on dependencies among transactions are too restrictive for design applications. The design process is inherently iterative. If two people are working together on the same project, they may do interleaving reads and writes on the same set of objects as they refine the design. Thus, their transactions may depend on each other. Because this type of situation is characterized by transactions working together, we call it *cooperative.*

Rather than always allowing all cooperative transactions to interact freely, we may wish to structure their interactions in a way that makes sense. For example, consider the airplane design hierarchy in Figure 1 in Section 1. We may want to enforce that after the engine designers have finished writing the engine design, the the maintenance manual people must read that design and update the maintenance manuals before the engine design is complete. Thus, we need to restrict interactions among members to ones that reflect the structure of the underlying design process. In [Ska89], a notion of *patterns* and *conflicts* is proposed. Patterns indicate allowable sequences of operations that a set of members can do on a set of objects. Conflicts are usually defined within the context of patterns, and indicate prohibited sequences. During execution, operations requested by the members may be reordered or refused in order to ensure that the resulting history conforms to the allowable patterns and conflicts. This process is called *synchronization.*

4.3 Locking

In a traditional database, transactions must reserve the capability to access or manipulate any object they touch by locking it appropriately. With the cooperative model, it is also true that some members may wish to reserve for themselves the capability to do some operations. However, standard read/write locks are not necessarily the right model because they may interfere with the ability to cooperate in a structured way. Furthermore, using locking as a concurrency control mechanism would be redundant in certain situations.

Consider first what a traditional write lock does. It reserves for the transaction the capability to read and write operations on an object, while preventing other transactions from either reading or writing the object. In a cooperative environment, transactions must be allowed to interact more freely than that. In the example from Figure 1 in Section 1, assume that there are two authors working to update the maintenance manual. They both may need to cooperate in writing the next draft. Their writes may interleave arbitrarily, though

overwriting should be prevented. This is a case where a pattern-based synchronization mechanism can do everything that locks were used for previously, so any use of locks would be redundant.

A second problem with locks occurs because the use of locks may interfere with the natural structure of the underlying task. For example, say that every change by the engine design group should be followed by the production of a new version of the engine maintenance manual. Thus, any two writes of the engine design must be separated by one or more writes of the engine maintenance manual. If the engine design group held a write lock on the engine design after it completed one of its writes, the maintenance manuals group would be prevented from reading the engine design and writing the manual. This misuse of locks would prevent the draft of the manual from being completed.

Unfortunately, it is still often necessary for members of a transaction group to reserve the capability to access or manipulate some object. This is because a transaction group may need to merge the effects of sequences of operations produced by its members into the history of its parent somehow. We need to preserve the capability to do the external operations generated from the group's internal history. We call this an *intention.* When a transaction group receives an intention request, it can check to see that the requested pattern is reasonable before accepting the request. Once the intention is accepted, the member can do the intended operations without fear of other conflicts.

4.4 Dependencies

Cooperation has a strong effect on how dependencies are kept in the database. If you draw the transaction dependency graph for a serializable transaction history, the serializability theorem shows that the graph is acyclic. However, we have already seen that transaction dependency cycles should be allowable in cooperative transactions.

Unfortunately, cycles in a dependency graph are not helpful when trying to analyze dependencies for recovery purposes. However, we can solve this problem by looking at the dependencies at the *operation* level. Operations are still atomic. Furthermore, they can only depend on prior operations. Therefore, if we ignore simultaneous operations, the dependency graph at the operation level should be acyclic. We solve the problem of simultaneous operations by ensuring that they can not depend on each other. We can then use the acyclic operation dependency graph during recovery.

We have defined correctness in terms of patterns and conflicts. The recovery process takes a history that was correct before some failure, and transforms it into a history that is correct after the failure. In the process of doing this, it must find everything that depended on the thing that failed. In the context of cooperative transactions, we see that the failure initially invalidates some number of operations by a single member. Each of these operations is participating in one or more patterns. The failure of an operation should cause the failure of all subsequent operations in each pattern in which it participates. Those operations in turn are participating in other patterns, and so the process continues until all dependent operations have failed.

Thus, we see that the dependencies between operations are defined by the patterns in which they participate. For each pattern that has been active during a particular history, we can isolate the projection of that pattern on the history. Each operation should be

dependent on the previous operation in the projected history. Because these dependencies are associated with the underlying patterns, we call them *pattern dependencies.*

Some of these pattern dependencies can be quite subtle. A transaction group's internal protocols specify how its members interact. For some task, they may require operations to occur in some specific sequence involving several members and several objects. For instance, consider again the consequences of requiring that the maintenance manual document be rewritten each time the engine design group releases a new design. This means that each operation that writes the maintenance manual document is dependent on the previous operation that wrote the engine design information. If the engine design group's write is subsequently aborted, the maintenance manual group's write operation also becomes invalid, even though there is no obvious correlation due to a sequencing of operations by the same member or on the same object.

Certain types of *reads-from dependencies* also must be maintained between operations. When a member M_1 reads an object O_1 last written by M_2, the read is dependent on the validity of M_2's write operation. Thus, each read of an object is dependent on the previous write of that object in the transaction group's history. However, if M_1 then proceeds to do a write operation on some other object O_2, the write may not necessarily depend on the previous read of O_1. In fact, this dependency will exist only when both the read of O_1 by M_1 participates in the same pattern as the write of O_2 by member M_1. Thus, any dependencies of write operations on earlier read operations are encoded in the patterns already.

4.5 Transactions

In a cooperative transaction hierarchy, many of the properties traditionally attributed to transactions have moved elsewhere. For example, each transaction traditionally defined a coherent unit of work *(task)*, in the database. However, cooperative transactions work together in groups to do a task, so the *transaction group* defines the task in a cooperative database. Thus, when a failure occurs, it is the responsibility of the *transaction group* to ensure that the proper steps are taken for recovery.

We have already noted that it is not logical for a cooperative transaction to be atomic; instead, the property of atomicity is only preserved for operations. Thus, dependencies are kept among operations instead of transactions. Similarly, transactions should be able to commit (checkpoint) and abort *operations*, allowing parts of a transaction to take effect and other parts fail. Furthermore, since the checkpoint and abort requests are oriented towards operations, they should not terminate the transaction.

Cooperation also impacts what it means to checkpoint or abort the operations of a member. Consider first the case where some member thinks that it is complete, or has reached some stable state. This is indicated when all patterns that the member's operations are participating in are complete. At this point, the transaction can checkpoint all of its operations. This does not necessarily mean that the task that it is working on is complete; rather, it means that the transaction believes that its *piece* of the task is complete. The task itself is not complete until all of the members that have cooperated on the task think that they have finished. Thus, the member's checkpoint may be dependent on other members' checkpoints, which can occur either in the past or in the future. The changes made by the member at the time of the checkpoint only become firm once all the operations on which they

depend have been checkpointed as well. It is only when all its members have checkpointed and terminated that the transaction group can consider its task complete. Thus, even though checkpointing and aborting are done at the operation level, it is the transaction group that finally decides whether its task is complete.

4.6 Recovery

Allowing members to cooperate, even in a structured manner, complicates recovery. Say some member of a transaction group wants to abort in the traditional manner, which implies undoing all of its changes and then terminating. Other members of this transaction group may be dependent on the aborted transaction, so the initial impulse is to abort those transactions as well. This cascading may be fairly extreme, especially since the member being aborted may have done many operations that are still valid. Undoing those operations could be fairly drastic. This is especially true in design environments, which are characterized by long, interactive transactions. Therefore, it seems wiser to inform the member of the problem, and allow it to compensate for the failure in some correct manner. One potential way to do this could be to allow the remaining members each to do a sequence of compensating operations that preserves what was still valid in the old operations while correcting what had become invalid. Each compensating operation proposed by a member must be accepted by the transaction group's synchronization protocols before it can be executed. Thus, the transaction group coordinates the recovery process, ensuring that the compensating operations are correct. This also effectively means that all of the remaining members of a transaction group could cooperate to recover from the member's failure.

A second problem associated with aborting or failing transactions is what happens to its members when a transaction group fails. Again, the members' lifetimes are fairly long, so it may not be the best thing to abort all of the transaction group's members as a consequence. Another alternative in the case of failure might be to wait for a replacement transaction group to restart, then have all the members join the new transaction group, and cooperate to recover its history. One potential way of doing this is to keep a log and timestamp each entry. The replacement member then re-executes each operation in the order in which it was executed initially.

4.7 Permanence

The transaction group structure we have described allows the user to define an arbitrary nesting of cooperative transactions. At any given level except the root, a transaction group may make the changes its members have made on some set of objects visible to its siblings by checkpointing. This makes the changes recoverable by the transaction group in some sense, because it releases new object versions with the changes to the transaction group's parent. If the transaction group fails, it can always recover the modified objects from its parent.

Unfortunately, there is a major difference between this operation and a traditional commit operation. There is no guarantee that when a member checkpoints, its changes have been made permanent. There are two reasons for this. The first, discussed above, is that the checkpoint may be dependent on operations by some of the member's siblings. It is not until the siblings have checkpointed those operations that the changes are really valid. Until

then, the transaction group and its members must be prepared to recover if the checkpointed changes become invalid due to the failure of some sibling member or the abort of a sibling's operation. The second reason that the checkpoint does not make the changes permanent is that there is no guarantee that the parent has a backing store to save the objects in its cache unless the parent is the root transaction group. Thus, the failure of the parent may cause the changes to disappear.

What the checkpoint operation does guarantee is that the changes are visible to the group's siblings, and that they are made at least as permanent as the objects in the transaction group's parent's cache. It also guarantees that the changes are recoverable from the parent if the transaction group fails. This is consistent with the notion of transaction groups, in that the group itself is the locus of control for its members, and thus may manage the operations and the consequent changes in its own way.

5 Operation Machines

Transaction groups may need to control not only the concurrent access of objects, but also the sequences in which different objects may be accessed by its different members. None of the locking mechanisms in existence really provide the expressiveness and flexibility required for transaction groups. Using locks, we can only be too restrictive. On the other hand, both using non-restrictive locks as in ObServer [SZR86] and eliminating locks entirely are too permissive. Thus, we can't really control access to objects at the right level. In this section, we present a user-definable synchronization mechanism called an *operation machine* for defining and enforcing the correct interleaving of members' operations and the allowable communications between members.

5.1 Operation Machine Definition

The concept of operation machines was first proposed by Skarra [Ska]. In her model, a transaction group's criteria for determining the correctness of an operation history is based upon operation patterns and conflict specifications. Permissible patterns of object operations invoked by transaction group members are specified formally using augmented finite state automata. We use her definition of operation machines, except we restrict the operation set to {*read, write*}.

Formally, we define an operation machine as a tuple

$$\mathcal{OM} =< N, K, \Sigma, \Delta, s, c, F >$$

N is the name of the machine,
K is a finite set of states,
Σ is an alphabet,
Δ is a transition function from $K \times \Sigma \rightarrow K$,
$s \in K$ is the initial state,
$c \in K$ is the current state, and
$F \subseteq K$ is the set of final states.

Each transition in an operation machine includes the alphabet symbols that define the transition functions. A symbol σ in the alphabet Σ is a quadruple

$$\sigma =< M, OP, OBJ, P >$$

$M \in \{any, m, \overline{m}\}$ is the identifier of some member, where *any* is any member, m is a member identifier, and $\overline{m}$ is any member except m,
$OP \in \{r, w\}$ is an operation, where r is read and w is write,
OBJ is an object identifier, and
$P \in \{a, r, q\}$ is a predicate describing how operation $< M, OP, OBJ >$ relates to the machine's pattern at this time, where a is accept, r is refuse and q is queue.

In an operation machine, the start state represents the beginning of a pattern. Machine transitions represent operations by a member on an object. They are annotated with predicates that check for operation conflict. Based on member identity and the arguments to the operation, the predicates determine whether an operation should be accepted, rejected, or queued. When an operation is executed, the current state of the machine is updated to be the destination state associated with the transition. This is called an *explicit transition.*

The operations explicitly reflected in the operation machines are only those which are relevant to the pattern. If an operation is not specified in an out-transition from a state, that means that the operation is not relevant to the pattern at that point. As a consequence, a requested operation is implicitly accepted if it has no associated out-transition from the current state. This is called an *implicit transition.*

In addition to expressing the permissible orderings of object operations, operation machines capture the "completeness" of a pattern by distinguishing between final and non-final states. If a machine is in a final state, the members have correctly completed the sequence of operations required for a specific task. We can ensure that the database is consistent by allowing members of a transaction group to checkpoint only if all machines relevant to any pattern they participate in are in a final state.

Operation machines may be defined directly, or instantiated from an *operation machine template.* A template is defined in the same way as an operation machine, but the transition functions have object and member variables. Also, a template always has its current state set to the start state. A template is instantiated by making a copy of its definition, and then binding the object and member variables in the copy to specific objects and members.

5.2 Operation Machine Types

There are various types of operation machines that are useful in a cooperative database. *Synchronization machines* define the patterns that enforce the underlying concurrency control protocol (e.g., cooperative, serializable). *Pattern Machines* define the sequences of operations needed to do the task associated with the transaction group. *Protection machines* are used to prohibit specific members from doing specific operations on an object.

5.2.1 Synchronization Machines

Each transaction group has a single *synchronization machine template* that defines the underlying synchronization protocol. This template is specified at the time the transaction group is first set up, and cannot change thereafter.

The synchronization machine template is instantiated once for each member-object interaction. Thus, a synchronization machine is created when the member first interacts with the object copy, either by doing an operation on the object or by declaring its intention to do so. The machine may be bound to some object that has not yet been read into the cache. This is all right, as we are assured that some copy of the object will exist in the cache at some future time.

Synchronization machines are unbound and removed in two situations. The first is when the member that is bound to the machine has terminated, and the machine is in a final state. In this case, the synchronization machine no longer needs to control the member's access to

the object, since no future operations by that member on the object can occur. The machine must be in a final state to ensure that a complete traversal of the machine has occurred.

The second case where a synchronization machine is unbound and removed is when the object is removed explicitly from the cache. This can only happen when all members that have machines bound to the object have either checkpointed or aborted all of the operations on that object, and all of the machines associated with that object are in a final state. The first condition ensures that no operations are lost as a result of deleting the object. All of the operations that the members of the transaction group intended to do on the object have been released to the transaction group's parent. The second condition ensures that the resulting history is correct, in that no incomplete traversals are represented in the transaction group's history.

Figure 3 gives two examples of synchronization machines. Figure 3(a) shows a machine that prevents other members from reading the changes made by M until M terminates. Figure 3(b) shows a synchronization machine that enforces a sort of cooperation. Before member M can write the object, it must have read it. Furthermore, if some other member writes the object, then member M must read the written changes before it can write the object again.

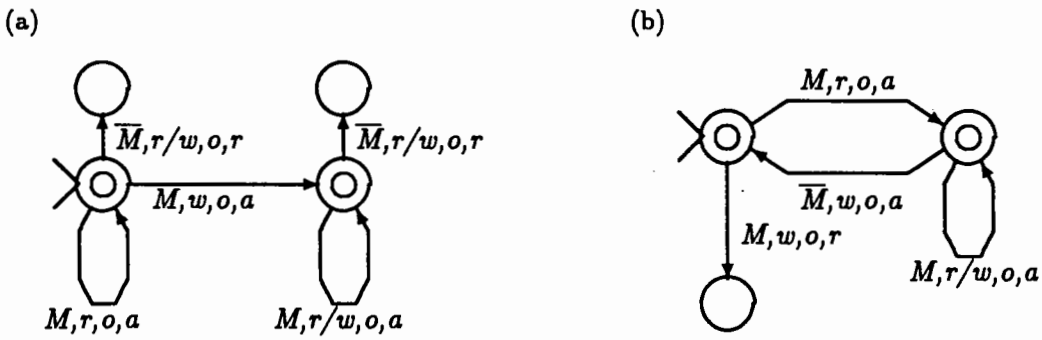


Figure 3: Synchronization machines: (a) Writes not visible until commit; (b) No overwrites.

5.2.2 Pattern Machines

Pattern machines define the overall form of the interactions between the various members and objects in the transaction group. A pattern machine may be bound to many objects and many members. Each pattern machine enforces certain orderings among the operations associated with the transaction group's specific task, and therefore may relate operations that would otherwise be unrelated. For example, a pattern machine may enforce that the last write of the maintenance manuals must follow the last write of the engine design. This machine is shown in Figure 4.

Pattern machines are defined and bound at the time the transaction group is created. This means that the members that are explicitly named in the operation machine must be known at the time the transaction group is created. Pattern machines are removed when the transaction group terminates.

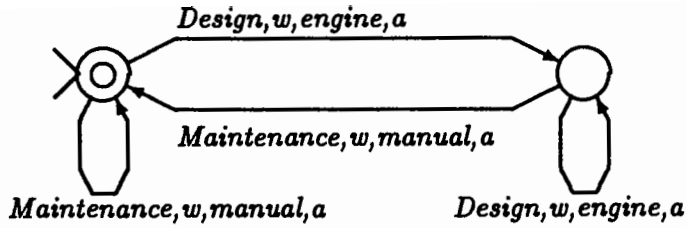


Figure 4: Pattern machine.

5.2.3 Protection Machines

Protection machines prevent unauthorized access to an object. They are implemented using conflict arcs. For example, the machine in Figure 5 prevents the maintenance manuals group from writing to the engine design object. Protection machines are defined for the duration of the restriction.

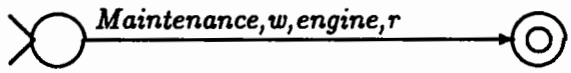


Figure 5: Protection machine.

5.3 Operation Machine Use

Operation machines enforce patterns of interactions among different members on different objects. At any time in the execution of an operation, many machines may be bound to a given object. Each of these machines enforces a different pattern in which the members interacting with the object participate. Together, they are used to enforce the overall correctness of the operations on the objects.

5.3.1 When Operations are Accepted

When a member requests an operation, it must be checked for correctness before it can be executed. This means that we must ensure that each arc traversal caused by the operation is a part of a correct traversal. We specify an operation request as a tuple

$$< M, OP, OBJ >$$

M is the ID of the member requesting to do the operation,
$OP \in \{r, w\}$ is an operation, where r is read and w is write, and
OID is the object identifier of the object.

There may be several operation machines bound to the object *OID*. For each such machine, the arc from its current state associated with the requested operation defines how it participates in the associated pattern. There are four cases:

1. The arc is an accept arc. This operation participates in this pattern, and is correct for this pattern if done now.

2. The arc is a queue arc. This operation cannot be done now, but may be doable later.

3. The arc is a refuse arc. This request cannot be satisfied.

4. There is no such arc in this machine. In this case, this operation does not participate in this machine's pattern, but is acceptable by it.

If all machines bound to the object are in the first or last categories, the operation can be done. If any machine is in the third category, the request must be refused. Otherwise, the request may be queued until it can be satisfied.

Thus, we see that the operation machines bound to an object work together to determine whether an operation on that object is correct. We could do this more efficiently using a single machine that is the intersection of all of the machines bound to the object. However, the number of states in the resulting intersection machine is exponential in the number of states in the largest operation machine ($\leq m^n$, where m is the number of states in the largest machine and n is the number of machines). Since machines can be added and removed dynamically, this optimization is probably not worth it.

As an example of this, assume that the *Design* transaction group has an *engine* and a *manual* object. The *engine* object has the two operation machines shown in Figure 6 bound to it. These two machines enforce different constraints on a history. The operation machine in figure 6(a) enforces a pattern involving multiple members and objects: The *Maintenance* transaction group must read the last version of the *engine* object written by the *Design* group before redoing the manuals for the new engine design. Figure 6(b) shows an operation machine that enforces that the member *Design* must read the last version of object *engine* before modifying it. It is an example of a *synchronization machine* because it specifies an underlying cooperative synchronization protocol. Consider the situation where both machines are in the start state. The operation <*Design,w,engine*> would be refused by the machine in (b), because the *Design* transaction group must read the *engine* object first. However, the operation <*Design,r,engine*> would be accepted, because it is accepted by the machine in (b) and not relevant to the machine in (a).

5.3.2 Algorithm

This atomic algorithm returns *ACCEPT* if the operation can be done now, *QUEUE* if the operation may be doable later, and *REFUSE* if the operation cannot be done. If it returns *QUEUE* and is requested to do so, it places an intention request for the operation on the queue.

1. For each element m in S, find an arc from the current state for the operation whose predicate evaluates to r. If such an arc is found for some machine, return *REFUSE*.

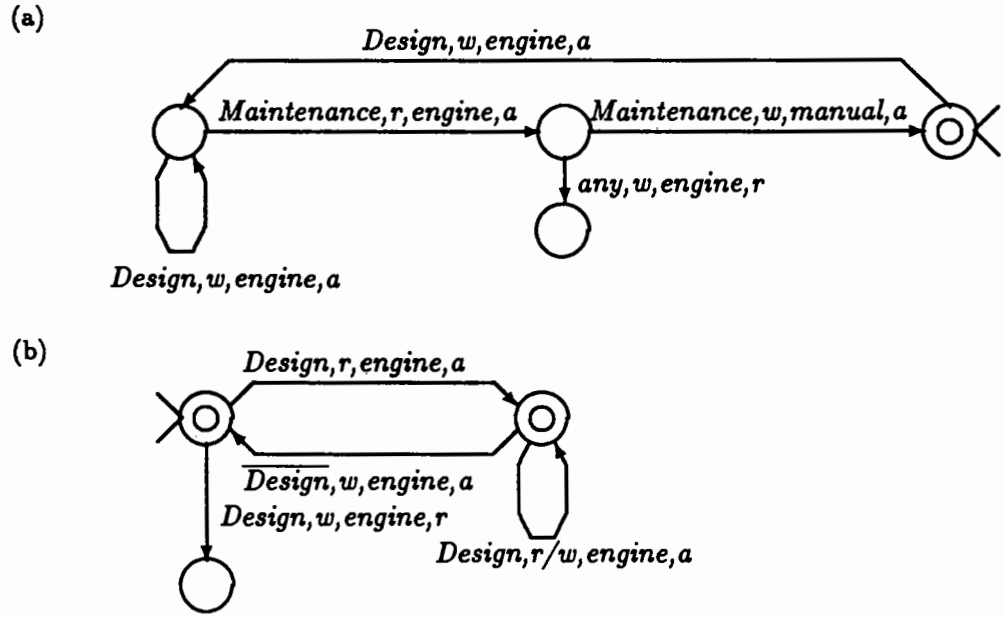


Figure 6: Operation machine examples: (a) Enforces a pattern that spans multiple members and objects; (b) Enforces a cooperative synchronization protocol on a single object.

2. Otherwise, for each element m in S, find an arc from the current state for the operation whose predicate evaluates to q. If such an arc is found for some machine, create an intention for the operation and place it at the end of the queue if requested. Return *QUEUE*. When the intention is removed from the queue, it is either accepted or refused at that time. Return *ACCEPT* or *REFUSE* appropriately.

3. Otherwise, return *ACCEPT*.

6 Concurrency Control Using Intentions

In our definition of the transaction hierarchy, we specified that each transaction group has a private cache containing copies of the objects that it is currently accessing. This means that there may be multiple copies of the object in existence, therefore there is a potential for inconsistency among the copies.

We use *intentions* to control the way members access object copies in the transaction hierarchy so that the overall sequence of operations on that object is correct. Intentions reserve the capability for a member to do a single operation on a single object in its parent's cache. Note that this differs from the original notion of intentions defined in Section 4.3, in that the patterns are restricted to a single operation. This considerably simplifies the process of determining when an intention can be accepted.

6.1 Overview

When a member M of a transaction group TG is granted an intention to do an operation, the copy of the object in TG's cache is restricted so that none of its other members can do an operation that later causes the intended operation to be rejected. Similarly, the copy in M's cache is restricted so that the net effect of the operations of its members is the single operation for which the intention was declared.

When the member M is a transaction group, the intended operation generally represents the combined effects of a sequence of operations by the transaction group's members on the object. For example, many reads and many writes by the members of M can be consolidated into a single write by M to TG. Thus, there may be more complex patterns in M associated with the single operation in TG. We call this phenomenon *batching*.

The sequence of steps required to gain and release intentions is very similar to that of locks. When a member M wants an intention, it makes an *intention request* to the transaction group TG. Once TG ascertains that the operation can be done immediately, it *accepts* the intention. Once an intention has been accepted, TG ensures that M can do the operation at any time by preventing any operations that would later conflict with it from being processed. M may *release* the intention at any time up to the point when the operation completes.

It is possible for a member to request an operation without having acquired an intention. The operation is still done, provided that it is allowable given the specified operation patterns and conflicts. It also may be delayed by operation scheduling code.

6.2 Intention Machines

We enforce intentions using operation machines. *Intention machines* are used for managing operations on multiple copies of an object as if there were only a single copy. In other words, they ensure that the merged operation sequences correspond to a traversal of the operation machines bound to the copy of the object in the root database.

When a member M is granted an intention by transaction group TG, we use operation machines to enforce two things. At the transaction group level *(TG)*, we ensure that no other member does an operation that conflicts with the operation requested by member M. If M is itself a transaction group, then we need to ensure that its members do operations

only according to a pattern that batches to the single intended operation. Thus, we associate two operation machines with each type of intention (read or write), one of which is bound to the object copy at TG's level, and the other to the object copy at M's level.

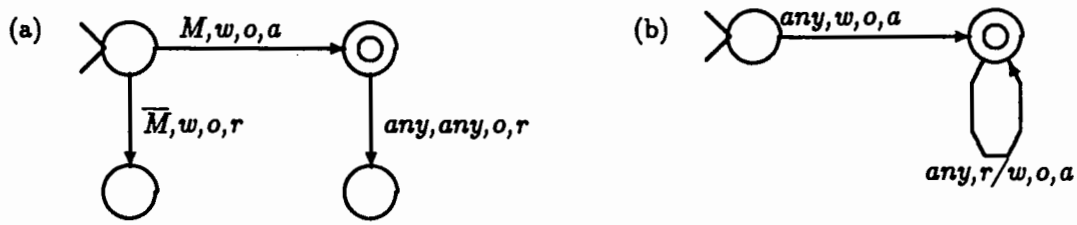


Figure 7: Intention machines: (a) at TG's level; (b) at M's level.

Figure 7 shows an example of operation machines that may be associated with an intention to write. Figure 7(a) shows the machine which is bound to the object at TG's level when TG has a cooperative synchronization protocol and there are no other operation machines associated with the object. It prevents any write of the object by any other member, while allowing exactly one write to the member which granted the intention. While it is not a part of the underlying pattern, this machine also allows anyone to read the object until the write operation by M is actually executed. This is because the read operation causes no change of state in any existing machine bound to the object, nor does the read operation modify the object. Figure 7(b) shows the machine associated with the batched write operation at M's level. It allows many read and write operations by its members to be batched into the single write operation by TG. However, at least one member must do a write operation first.

Intention machines are bound to the object copies in the member and the parent at the time the intention is accepted. They are removed at the time the intention is released by the member.

7 Histories and Correctness

This section formally defines histories, and then defines the correctness of a transaction group in terms of these histories.

7.1 Member Executions

From an operational point of view, the execution of a member is a partial order containing read, write, checkpoint, and abort operations. This partial order is called a *happens-before* ordering (<), where if *op1* and *op2* are operations by a given transaction group or its members, and if *op1* is executed before *op2*, then *(op1 < op2)*.

The member execution conforms to the following constraints:

1. It contains a record of each operation that was requested by the member and actually executed.
2. For any operation *op* in the execution, there is at most one *checkpoint(op)* record or *abort(op)* record.
3. If there is a *checkpoint(op)* record for an operation, then *(op < checkpoint(op))*
4. If there is an *abort(op)* record for an operation, then *(op < abort(op))*
5. Any operation *op1* in the execution that is dependent on some operation *op2* has *(op2 < op1)*.
6. An execution may contain exactly one *terminate* record. For any operation, checkpoint, or abort record *r*, *(r < terminate)*.

7.2 History

A history of a transaction group is a partial order containing the read and write operations by the group and its members that have not been aborted, ordered by the happens-before partial order. The history conforms to the following constraints:

1. It contains an entry for every read or write operation *op* in each of its member's executions that is not followed by an *abort(op)*.
2. It contains an entry for every read or write operation *op* in its own execution that is not followed by an *abort(op)*.
3. For a specific member, if *(op2 < op1)* in the member's execution, then *(op2 < op1)* in the group's history.
4. For the transaction group, if *(op2 < op1)* in the group's execution, then *(op2 < op1)* in the group's history.
5. Any operation *op1* in the history which is dependent on some operation *op2* has *(op2 < op1)*.

7.3 Definition of Correct Histories Using Operation Machines

We use operation machines to enforce the patterns and conflicts defined at any given point in an operation sequence. We formalize our definition of correctness of histories in terms of operation machines.

A *traversal* of an operation machine is a sequence of operations associated with a consecutive, ordered sequence of explicit traversals for the machine, beginning at the start state and ending at the current state. For example, given the operation machine shown in Figure 8, the operation sequence (1) in Figure 9 is a traversal. The operation sequence (2) is not a traversal, because the first operation corresponds to an implicit transition. The operation sequence (3) also is not a traversal, because the third operation corresponds to an arc with a predicate function that always returns refuse. A *complete traversal* is a traversal which ends at a final state. For example, the sequence (4) in Figure 9 is a complete traversal.

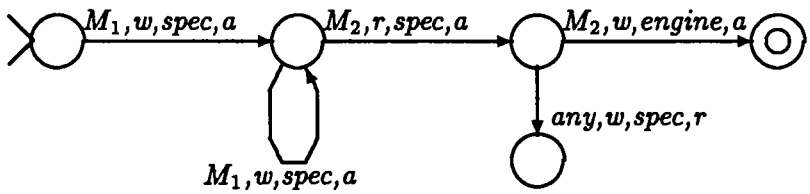


Figure 8: Operation machines and traversals

1. {<M_1,*w*,*spec*>, <M_1,*w*,*spec*>, <M_2,*r*,*spec*>}
2. {<M_1,*r*,*spec*>, <M_1,*w*,*spec*>}
3. {<M_1,*w*,*spec*>, <M_2,*r*,*spec*>, <M_1,*w*,*spec*>}
4. {<M_1,*w*,*spec*>, <M_1,*w*,*spec*>, <M_2,*r*,*spec*>, <M_2,*w*,*engine*>}

Figure 9: Traversal examples.

For each operation in a history, we know which operation machine arcs were followed as a result of its execution. Define the sequence π_{OM} associated with operation machine *OM* as follows:

$$\pi_{OM} = \{O \mid \text{operation } O \text{ caused an arc traversal in } OM\}$$

Then, we can define a *correct history* to be one in which, for all machines *OM* which were active during the history, all projections π_{OM} are traversals; also it must contain no operation that conflicts at the current point in the traversal of any machine active at the time it was executed. A *complete history* is a correct history in which all traversals are complete and all operations are checkpointed.

8 Objects, Copies and Versions

An *object* in the database contains a block of data and the control information associated with it. The data usually corresponds to the instance variables for some persistent object in an object-oriented system. Since information associated with an object may change over time, there may be several *versions* of the data for a single object.

Formally, an object ($\mathcal{O}$) is a tuple defined by:

$$\mathcal{O} = <OID,\ OM,\ NL,\ VS,\ LVV>$$

OID is the object identifier of the object,

OM is the set of operation machines currently bound to the object,

NL is the list of members who have requested notifications, sorted by operation (see Section 12),

VS is the chronological sequence of versions of the object, including both valid versions and recent invalid versions, and

LVV is the last valid version of the object in VS.

8.1 Object Copies in Transaction Groups

The members of a transaction group operate on a local copy of the object. This copy may have several local versions associated with it. The copy has the same data structure as the object in the root database. When a member first reads an object, the transaction group reads its parent's most recent version of the object. If the parent transaction group does not have a copy of the object in its cache, the read request is propagated up the transaction hierarchy until a copy is found. Then, the most recent version in that copy is copied back down the transaction hierarchy. When a member copies an object from its parent, it initializes the data and control structures for its copy as follows:

OID is the ID of the requested object.

OM contains a copy of the synchronization machine defined by the transaction group, bound to the object and to the member which requested the read.

NL is empty.

VS is copied from the parent's most recent version.

LVV points to the version read from the parent.

Sometimes, we need to associate a control structure with an object, even before the data is actually read into the cache. This occurs when the transaction group explicitly defines pattern or protection machines associated with the object, and also when an intention is requested by a member for some operation on the object. At the time the operation machines are defined, a *future copy* is placed in the cache. A future copy is identical in structure to a copy, except that it contains no data. It is initialized as follows:

OID is the object identifier of the object.

OM contains each operation machine explicitly bound to the object when the transaction group was created, and any intention machine associated with the object.

NL is empty.

VS is empty.

LVV is the null pointer.

In this way, we can use the operation machine information in the future copy when determining the correctness of the operation without actually requiring a the data to be in the transaction group's cache.

8.2 Versions

A new *version* of an object is created in a transaction group's object copy either when the object is read from the transaction group's parent or when one of its members does a write operation. Each transaction group numbers the versions its member creates in ascending order based on the time they were written. The version space is assumed to be linear. That is, the versions of an object in a transaction group do not form a version tree.

Formally, a version is a tuple $\mathcal{V}$ defined as follows:

$$\mathcal{V} = <N,\ T,\ LE,\ D,\ OMS,\ S>$$

N is the version number assigned by the transaction group,

T is the version creation timestamp assigned by the transaction group,

LE is a back pointer to the history entry for the operation which created the version,

D is the version data,

OMS is the set $\{<OM,\ S> \mid OM$ is an operation machine bound to the object and S is its state at the time of the write.$\}$

$S \in \{VALID,\ INVALID\}$ is the state of the version.

9 Logs and Operation Dependencies

We defined a *history* for a transaction group TG in Section 7 as the partial order of operations by TG and its members.

A transaction group's *log* records its history and other information required for recovery. *Entries* in the log record the operations in the history. They are fully-ordered in a way consistent with the partial order in the history. In practice, the records of the operations are ordered chronologically by the time they were executed. Operations that were executed simultaneously may be reflected in any order in the log.

For a given transaction group TG, there are entries in its log for:

- All the operations by each member of TG on the object copies in its cache *(member entries)*,
- All of the operations by TG on the object copies in its parent's cache *(group entries)*.

The log is created as the members and transaction groups execute their operations. When there is a failure or an aborted operation, the log is used during recovery. Because of this, a transaction group's log also records the dependencies between the operations. This allows the transaction group not only to determine what is affected directly by the failure, but also to determine what other operations were affected because they depended on some failed operation.

When M does an operation and both M and TG are transaction groups, there are two log entries recorded for that operation. In TG's log, there is a member entry for the operation, and in M's log, there is a group entry. This allows us to determine how invalidations of operations in one transaction group affect its parent transaction group and its members. Figure 10(b) shows correlations between log entries in the transaction groups for TG and M_1 for the transaction hierarchy in Figure 10(a).

9.1 Operation Dependencies

In addition to recording relevant information about the operations in a history, the log records dependencies among the operations in that history. These dependencies include *pattern*, *reads-from*, and *parent-child* dependencies.

In Section 4.4, we noted that dependencies occur among operations in a cooperative transaction hierarchy, rather than among transactions. We defined *pattern dependencies*, which are kept between operations that correspond to a projection of some pattern from a transaction group's history. For each pattern that is enforced on the operations in a transaction group, there is a corresponding operation machine that was enforcing that pattern. The projection of the pattern on the history is a *correct traversal* of the corresponding operation machine, as defined in Section 7.3.

When an operation is invalidated, all subsequent operations in each correct traversal that the operation was involved in also become invalid. Thus, we keep a dependency chain associated with each correct traversal in the log, where each operation in the traversal is dependent on the previous operation.

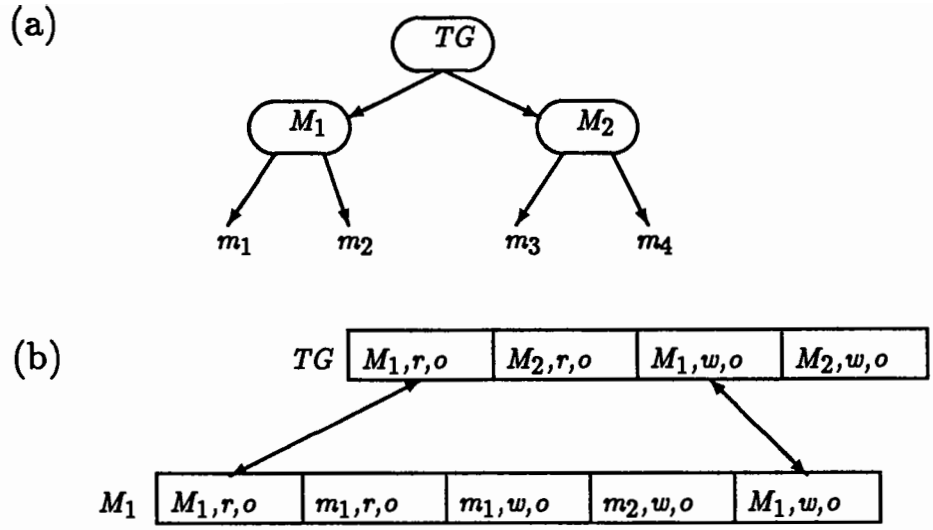


Figure 10: Example of logs of a transaction group and a member: (a) Transaction hierarchy, (b) Logs for TG, M_1.

In addition to the pattern dependencies, dependencies occur among operations at different levels of the transaction hierarchy. For example, when a member M of a transaction group TG first reads an object, the object must be copied into TG's cache first. M's read is correct only if TG read a correct version into the cache. If that version is later invalidated as a result of some abort, then M's read is also invalid. A similar situation exists with writes; when TG writes a version in its parent's cache, the validity of that write is based on the validity of the last write operation by one of its members. We call both of these *parent-child dependencies.*

A third form of dependency among operations is a *reads-from dependency.* These occur because a read of some version of an object depends on the operation that wrote that version.

Figure 11(a) shows a synchronization machine that is bound to the engine design specification for an airplane. Figure 11(b) shows a subset of the log for the *Design* transaction group. In this sequence, the specification is read and modified by member M. To the left of the entries for the operations are two dependency chains. The left chain shows the dependencies associated with the traversal of the synchronization machine. The right chains show the parent-child dependencies.

9.2 Formal Definition

A transaction group's log consists of a fully-ordered sequence of member and group entries for unaborted operations associated with the transaction group. The full ordering is consistent with the partial order in the history.

A member or group entry in a transaction group's log is a tuple

$$\mathcal{LE} =< EID,\ TID,\ OP,\ OID,\ V,\ OD,\ UOD,\ FD,\ S >$$

EID is *TID*'s unique identifier for the operation,

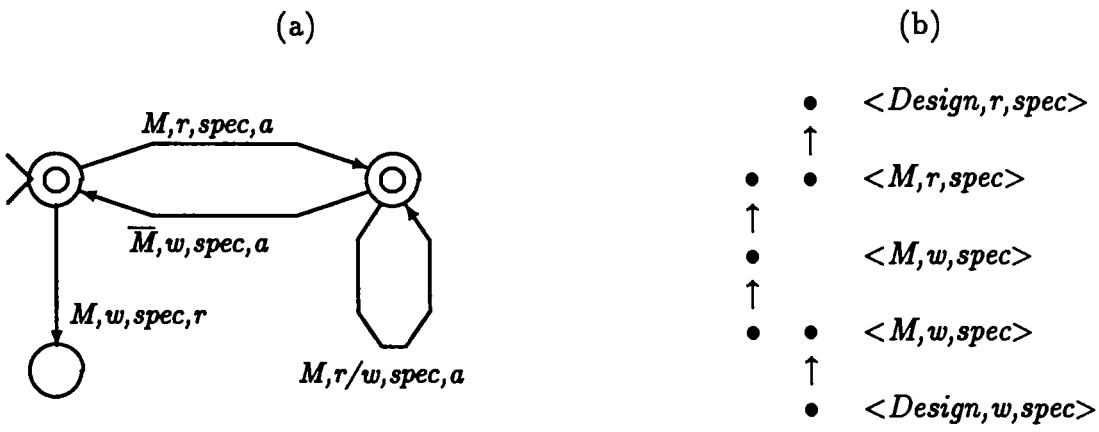


Figure 11: Example dependencies: (a) Synchronization machine, (b) Dependencies.

TID identifies the member that did the operation,
OP $\in \{r,w\}$ is the operation, where r is read and w is write,
OID identifies the object on which the operation was done,
V points to the (possibly null) version created by the operation,
OD is the set of operations on which this operation depends,
UOD is the set of operations in *OD* that could possibly still be invalidated,
FD is the list of operations depending on this one, used for invalidated operations during recovery to determine when the entry can be spliced out of the log, and
S $\in$ {*SUBMITTED, PENDING, COMPLETE, INVALID*} is the state of the operation.

Log entries are identified uniquely by <*EID, TID*> pairs.

Operations to add and remove entries from the *UOD* and *FD* lists must be atomic, because otherwise different recovering members could interfere with one another.

10 Invalidation and Recovery

Recovery is a two-phase process. In the *invalidation phase*, all operations that depend on some invalid operation are also invalidated, as well as any operations that conflict in the remaining history and the operations that depend on them. Any version written by an invalidated operation is also invalidated. In the *recovery phase*, transactions whose operations were invalidated work together to compensate for the failure. The invalidation and recovery phases preserve the correctness of the history.

A transaction may abort any of its uncheckpointed operations at any time. The abort process invalidates the aborted operations by marking the log entries as *INVALID*, and invalidating the versions created by them. When an operation is invalidated, both the *group entry* in the member's log and the *member entry* in the group's log are marked as invalid. It then finds the operations that transitively depend on the aborted operation, both in the current transaction group and in other transaction groups in the hierarchy using the *pattern*, *reads-from*, and *parent-child* dependencies in the logs. These operations are invalidated in the same way.

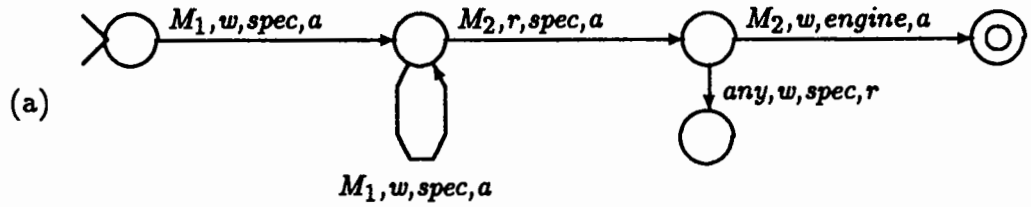
A third category of operations to invalidate is those that conflict in the new history. Conflicts may occur where they were not present in the original history because some machines are not in the same state as they were when the operation was originally accepted. As we process the log, we need to keep track of the "current state" of every machine that has had at least one of its operations invalidated. If an operation now conflicts (is queued or refused), it must be invalidated as well.

Once the invalidation propagation is complete, any object that has had some version invalidated is restored to its state at the time just after the last valid version was written. The invalidation of an operation breaks each traversal that the operation participated in. Thus, each operation machine is restored to whatever state it was in just before the first invalid operation in its traversal was executed.

Figure 12(a) shows an operation machine and, (b) shows an excerpt of the log showing the traversal of that machine. If M_1 aborts the second operation in the traversal, the operations $<M_1,w,spec>$, $<M_2,r,spec>$, and $<M_2,w,engine>$ are invalidated because of pattern dependencies. The operation machine is reset to be in the state just after the first operation, and the versions created by the second $<M_1,w,spec>$ operation and the $<M_2,w,engine>$ operation become invalid. The last valid version of the *spec* object is the one created by the first write. The last valid version of the *engine* object is the one that was current just before the write operation $<M_2,w,engine>$. Note that other operations also may be invalidated because of other pattern or reads-from dependencies.

As a second example, consider what would happen if the operation $<M_2,w,engine>$ were aborted instead in the traversal in Figure 12(b). This would leave the machine in Figure 12(a) in the state before the final state. If the history continues with the operation $<M_2,w,spec>$ in the traversal in Figure 12(c), that operation would now conflict and is therefore invalid.

The database is left in a correct state after the invalidation phase. Assume that the database was correct before the abort occurred. The abort operation breaks some traversals in the history. For each such traversal, the first invalidated operation is called the *break point*. Because of the pattern dependencies, the log entries for all operations "downstream" from the break point also are invalidated. The only part of the traversal left in the log is



(b) $\{<M_1,w,spec>, <M_1,w,spec>, <M_2,r,spec>, <M_2,w,engine>\}$
(c) $\{<M_2,r,spec>, <M_2,w,spec>\}$

Figure 12: (a) An operation machine, (b) A traversal, and (c) A traversal of a different operation machine.

the part preceding the break point. Thus, for each pattern, the log that remains represents an operation sequence containing a correct traversal for that pattern ending just before the break point.

The reads-from dependencies show places where one transaction's read depends on another transaction's write. The read operation becomes invalid when the write operation is invalid. Thus, other traversals may be severed because of the reads-from dependency. Invalidations may also cascade from traversal to traversal when an operation is invalidated that participated in more than one pattern. In the example from Figure 12, the operation $<M_2,r,spec>$ in traversal in (b) was invalidated. But this operation is also the first operation in the traversal in Figure 12(c). Therefore, the operation $<M_2,w,spec>$ must also be invalidated because it follows traversal (c)'s break point.

Conflicts also are correctly dealt with during the invalidation phase. Not all patterns are in the same state at a given point in the history as they were during the original processing. New conflicts are found correctly by maintaining a new conflict list and checking each operation in turn against that list. Recovery does not need to deal with operations that previously conflicted but now do not. This case is indistinguishable from the one where the conflicting operations were never submitted, and thus has no effect on the correctness or incorrectness of the new history.

Correctness is ensured after the invalidation phase because the use of pattern dependencies guarantees that all operations in a traversal that follow its break point will be invalidated as well. The process of checking and enforcing any new conflicts ensures that the new history is not incorrect because it now contains operation sequences that would not have originally been accepted. We limit the performance penalty due to the cascading from pattern to pattern with proper definition of the patterns. In particular, we do not define patterns over a set of operations that are not related by data- or application-specific constraints.

The log entry for each write operation points to the version it created. We invalidate a version when we invalidate the write operations that created it. After the invalidation phase, the latest valid version of each object is the most recent version created by some traversal in the remaining history. Thus, all effects of invalidated operations are purged from the database.

Once the invalidation phase is complete, the members are notified of any of their operations that have been invalidated, and the recovery phase begins. Since the members may cooperate on a task, we allow them to cooperate in the recovery process. This process is cooperative in the same sense as the initial work, and is governed by the same synchronization protocols. However, other members may wish to take responsibility for some of the changes that would otherwise be invalidated. Therefore, we allow members to do the following during recovery:

1. Abort any uncheckpointed operations. A traditional *abort* operation can be mimicked by immediately aborting all uncheckpointed operations and then terminating. This option also is useful when the member that initiated the invalidated operations has failed.

2. Reread any invalid object versions previously read by the member. This allows the member to remember what it has seen, in case it wants to incorporate some of the invalid data into a new version. The read of the old versions bypasses the synchronization mechanism.

3. Request compensating operations. These operations are scheduled and processed in the normal manner, to ensure that the history remains correct.

Given the first example above, M_2 could recover after the invalidation phase by rereading the *spec* object, adjusting the engine design appropriately, and writing the new *engine* object.

The recovery phase maintains correctness. This is because each action a transaction can take during recovery is individually correct.

11 Operations and Intentions

An operation defines an atomic action on a single object by a single member. In this paper, we limit the allowable operations to the traditional reads and writes. Before a member does an operation, it may reserve the capability to do the operation by declaring an intention. Furthermore, it may hold the intention for as long as it wants after the operation, up to the point that the operation completes.

Within the context of an intention, a transaction group may *batch* a group of operations by its members into a single external operation. The allowable ways to batch operations are defined by an association between the intended external operation and its internal operation machine. The batched operation is correct when the internal operations form a complete traversal of the internal operation machine. The intention ensures that the transaction group can legally merge the batched operation into the parent's history once all of the updates have taken place on its copy.

11.1 Operation Phases

This section describes the steps a member takes in executing an operation and making its effects permanent. First, if the operation is batched by the member, it must declare the intention to do the operation. This step is optional if the member's operation is not batched.

Each intention is associated with a potential operation *OP* by some member *M* on some object *OBJ* in the parent transaction group. If this operation were to occur, it could potentially cause an arc traversal on each operation machine bound to the copy of *OBJ* in the parent's cache. If the member is a transaction group, an operation machine corresponding to the intention is also bound to its copy of the object defining which operations can be batched into the intended operation. A template of this machine is associated with the intended operation at the time the member is created.

Whether or not an intention is requested, the next step is to do the operation. First we ensure that the operation is allowed by the operation machines. If an intention has been accepted to do the operation, we know the operation is allowed. Otherwise, the operation may be accepted, refused or queued. The member is notified if the operation is refused. Queueing an operation means that an intention request for the operation is placed on the queue and the member is notified. At some point the intention request is dequeued, either because the operation can no longer be done, in which case it is refused, or because the intention has been accepted. If the intention is accepted, the member can determine in its present context whether or not to resubmit the operation. If the operation is resubmitted, it is always accepted because of the intention. If the member does not resubmit the operation, it should release the intention.

Once the operation has been accepted, we know that its execution at this time does not conflict. Therefore, we can do the operation by executing the following steps atomically:

1. Read or write the object,
2. Update all the operation machines to reflect the current state,
3. Log the operation in case something fails,

4. Set the state to *SUBMITTED*, and

5. Send any notifications pertaining to the operation.

If the member that did the operation decides later that it is incorrect in some way, it can abort the individual operation. Otherwise, the operation checkpoints the next time the member checkpoints. The operation's state in its history entry is set to *PENDING*, and the process of making any changes resulting from the operation more permanent begins. However, the operation may still be undoable because of pattern and/or conflict constraints in the history. Once an operation cannot be undone, its state is set to *COMPLETE*. This occurs (at the latest) once the operation is checkpointed and all other operations before it in the history are complete as well.

The member may hold an intention to do the operation until the operation completes. At that time, the intention is released automatically. While the member holds the intention, it may also *augment* it with another intention request. This gives the member priority in reserving the capability to do its next operation.

11.2 Requesting Intentions

This section describes what happens when a member requests an intention to do an operation on an object, and currently does not hold any intention related to that object.

11.2.1 Overview

A member requests its intention to do an operation by sending an *intention request* to its transaction group. When the transaction group accepts the intention, it ensures that its copy of the object has an additional operation machine bound to it to block operations that conflict with the intended operation. If the member is a transaction group, it ensures that there is an object copy or future copy in its cache with the appropriate intention machine bound to it.

The intention request cascades up the transaction hierarchy until the object copy is found. The transaction group at that level ensures that the operation would be accepted if it were requested immediately. Otherwise, it may queue the intention. When the operation would be accepted, the intention is granted back down the transaction hierarchy to all the transaction groups along the path from the member. At each level, a future copy is created with the appropriate operation machines bound to it.

Occasionally, intentions may be translated into stronger intentions as the intention request cascades up the transaction hierarchy. This translation is governed by a transaction group's internal protocols.

11.2.2 Algorithm

This algorithm must be done atomically. In this and successive algorithm descriptions, M is the member, TG is its transaction group, and P is TG's parent.

1. TG checks for a copy of the object (or a future copy) in its cache. If none exist, it propagates the intention request up the transaction hierarchy to its parent P. When

this intention is accepted, *TG* makes a future copy in its cache and initializes it as follows:

OID is the object identifier associated with the intention request,
OM,NL,Q,VS are empty, and
LVV is the null pointer.

2. If the object copy in *TG*'s cache does not have a synchronization machine for its interaction with *M*, instantiate one and add it to the *OM* list in the copy.

3. *TG* checks with its machines to see if the operation is accepted, requested, or queued, using the procedure outlined in Section 5.3.2.

4. If the intention was refused as a result of the previous test, notify the member of the refusal and return *REFUSE*.

5. If the intention was queued, return *QUEUE*. The member can continue to work while waiting for the intention. When the intention is dequeued, return *REFUSE* or continue, depending on the response from the intention request.

6. The intention has now been accepted. *TG* creates an intention machine as follows:

   (a) Make initial, final, and dead states ($K = \{I, F, D\}$).

   (b) Make an accept arc for the intended operation that begins at state I and ends at state F.

   (c) Make refuse arcs from the initial state I to the dead state D for all write operations and any read operation which would cause a state change in some existing operation machine bound to the object.

   (d) Make a refuse arc from the final state F to the dead state D for every operation.

   TG then binds the new machine to the object, and adds it to its set *OM*.

7. If *M* is itself a transaction group, it instantiates a member intention machine based on the intended operation, binding it to the object and the member which made the request. It then adds this new machine to the object's set *OM*.

8. Return *ACCEPT*.

11.3 Doing Operations

This section describes the process of scheduling and executing an operation.

11.3.1 Overview

This algorithm checks to see whether the operation is currently accepted by all patterns that it could be involved in. If it is not, the operation may be queued or refused, as appropriate. The operation is done if it is accepted, and the operation machines and transaction group's log are updated. If the operation machines return *QUEUE*, an intention to do the operation is queued.

Once an operation has been done, there may be queued intention requests whose fate is now clear. We should process the queue routinely to ensure that the queued operations are done. We currently do this after every operation.

11.3.2 Algorithm

This algorithm is invoked when a member requests that an operation be done. An operation request is a tuple

$$\mathcal{OR} =< EID, TID, OID, OP, DATA >$$

EID is an operation identifier unique to the requesting member,
TID is the member identifier of the member requesting the operation,
OID is the object identifier of the target object,
$OP \in \{r, w\}$ is the operation, where r is read and w is write, and
DATA is the new version if this is a write operation.

All the steps of this algorithm except the processing of the queue at the end must be atomic. The queue processing may be done at any time.

1. *TG* checks to see whether or not it has a copy of the object or a future object in its cache. If not, it does one of two things, depending on the operation:
   (a) If *OP* is a read operation, then the read request must be propagated by *TG* to *P*. When this read succeeds, the object will be in *TG*'s cache.
   (b) If *OP* is a write operation, then the member is creating a new version. It must set up a future object in its cache.
2. If this is the first interaction of member *TID* with object *OID*, instantiate a synchronization machine from the group's template for the interaction and place it in the *OM* list of the object.
3. Let $S = \{m \mid m$ is an operation machine in *OM* in the object *OID*$\}$
   $S' = \{\}$.
4. Check whether the operation should be accepted, queued, or refused, according to the algorithm in Section 5.3.2. If there is an intention held for this operation, it is always accepted. Specify that returning *QUEUE* implies that an intention for the operation is placed on the queue.
5. If the operation was refused, return *REFUSE*.

6. If the operation was queued, wait for it to be dequeued. If the response is now *REFUSE*, return *REFUSE*. Otherwise, the intention was accepted, so continue to process the intention as in Section 11.2.2 and notify the user that the intention was accepted. Return *INTENTION_ACCEPTED*.

7. The operation was accepted, so do it. If it is a write operation, create a new version and add it to the object copy in the cache and reset the *LVV* pointer to the new version.

8. Create a log entry for the operation. Initialize it as follows:

   EID is the operation ID from the operation request,
   OP $\in \{r, w\}$ is the operation, where r is read and w is write,
   OID is the object ID,
   TID is the member ID,
   V is null for a read operation, or a pointer to the new version for a write operation,
   OD is $\{o_1 \mid o_1$ is the operation which got us to the current state for each machine relevant to the operation$\} \cup \{o_2 \mid o_2$ is the operation that created the version being read, if this is a read$\}$,
   UOD is $\{s \mid s \in OD$ and s is not *COMPLETE*$\}$,
   S is *SUBMITTED*.

9. If this was a *write* operation, set *LE* in the new version to point to the log entry created in the previous step.

10. Traverse each accepting arc associated with the operation by updating the current state to the destination state of the accepting arc.

11. Generate and send any notifications which are requested for *OP* on object *OID*.

12. Set the state of the transaction group *M* to *RUNNING*.

13. Return *ACCEPT*.

14. If this operation was not done in the scope of an intention, check if any queued operations can be accepted or refused, as per the algorithm in Section 11.6

11.4 Changing Intentions

Once a member has an intention, it has priority to do the operation. Furthermore, we allow intention requests from members who currently hold intentions to have priority over requests from other members. Thus, members can try to keep their priority by operating continually within intentions.

There are a couple of ways that a member may modify its idea of its intended operation after it is granted the intention. For example, if the member decides to do a different

operation than the one it has an intention for, it may attempt to *change* the intention. If the change fails, the old intention is still held. If it succeeds, the new intention is held instead.

If a member has already done the operation it declared the intention for, it may attempt to *augment* the intention. This means that the old intention is released, because its operation was completed, and the new intention is set up in its place. If the augment request is not granted, the old intention is still held by the member. The augmenting of intentions allows a member to attempt a sequence of operations that is not interleaved with any relevant operations by other members. However, members can still interfere with one another if they are both manipulating intentions on the same object.

11.4.1 Overview

The algorithm is the same for changing and augmenting intentions. If the operation has been done already, it is reflected in the history. Also, the intention machine bound to the transaction group's object copy is in its final state. If the operation has not been done, the intention machine is in its initial state. We can ignore the state of the intention machine, however, because the intention machine is not enforcing a user-defined pattern, but is blocking allowed operations to reserve some capability for some member. Therefore, we do not need to ensure that the pattern defined by the intention machine is complete.

When changing or augmenting an intention, we first check if the new intended operation is currently allowable. If it is not, the request is refused. The new intention cannot be queued. If the new intention could be granted immediately, we remove the old intention machines from the transaction group and the member, and set up the new machines. Requests to change or augment an intention take priority over queued intention requests and operations.

11.4.2 Algorithm

This algorithm must be done atomically.

1. *M* and *TG* find the intention machines for the old intention, remove them from the set of active operation machines *(OM)* associated with their respective object copies, and save them in a temporary location.

2. *TG* checks to see whether the intended operation would be accepted, using the procedure outlined in Section 5.3.2. If the result is *QUEUE*, the operation is not added to the queue.

3. If the intention was refused or queued as a result of the previous test, restore each machine associated with the old intention to the proper operation machine set *(OM)*. Return *REFUSE*.

4. The intention has been accepted. Follow the procedure outlined in Section 11.2.2 to build the new intention machines and bind them to the copies in *TG*'s and *M*'s caches, respectively.

5. Discard the saved machines associated with the old intention.

6. Return *ACCEPT*.

11.5 Releasing Intentions

This section describes how an intention is released.

11.5.1 Overview

Intentions may be released at any time, regardless of whether the operation has been done. If an intention is still in effect at the time the operation reaches the *COMPLETE* state, it is automatically released then.

11.5.2 Algorithm

The steps of this algorithm must be done atomically.

1. *M* and *TG* find the intention machines for the old intention and remove them from the set of active operation machines *(OM)* associated with their respective object copies.
2. Both *M* and *TG* process the queue. This is described in detail in Section 11.6.

11.6 Processing Queued Operations

The intentions on the queue are sorted in the order in which they were requested. When changes are made because an operation is done or an intention released, we may be able to accept or refuse some number of queued operations. Because of fairness, we want older requests to have priority over younger requests. Therefore, we consider the queued intentions in the order they were requested.

Given that the queue is sorted in request order, the simplest algorithm is to process the queue from the head to the tail, evaluating each intention request according to the algorithm in Section 5.3.2, accepting all intentions that can be accepted and refusing all intentions that can be refused. This algorithm is non-optimal, in that there are easy ways of determining the subset of the queued intentions that may have been affected by the finished operation or released intention. We discuss these strategies in a future paper on queueing and deadlocks.

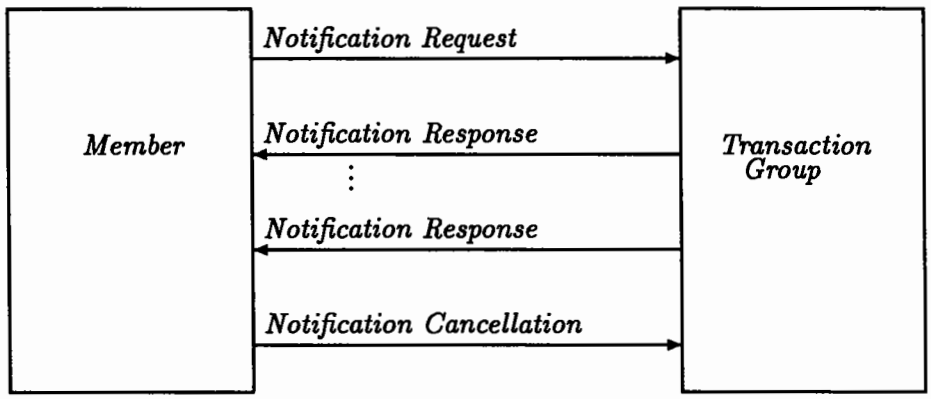


Figure 13: Message Sequence for Operation Notifications.

12 Communication

In a cooperative transaction hierarchy, we not only allow members to share intermediate versions; we also allow them to communicate using *notification* messages. There are two basic types of notifications that can occur, *operation notifications* and *invalidation notifications.* Operation notifications are generated on request, and indicate that some other member has done some interesting operation on some object. Invalidation notifications are generated automatically, and indicate abnormal situations such as aborted or failed operations.

12.1 Operation Notifications

Operation notifications are sent when a particular operation is done on a particular object. They are only generated on request; when a member is interested in knowing when an object has been accessed in a particular way, it can request that a notification be generated on an event-driven basis. Each time the operation occurs, a *notification response* is sent to the member which made the request. When the member no longer is interested in the operation, it can terminate the request using a *notification cancellation.* The sequence of notification messages is shown in Figure 13.

12.1.1 Notification Requests

A member in the transaction hierarchy can request an operation notification for any valid operation whose results would be visible to it. A notification request is a tuple

$$<NID,\ CP,\ OP,\ OID>$$

NID is a unique notification identifier,

CP is the complete path from the member to the root transaction group (using member IDs),

$OP \in \{r, w\}$ is the operation the member wants to be notified about, where r is read and w is write, and

OID is the object ID of the object on which the operation would be done.

Each member can only see the results of operations on object copies in its own cache, or in the caches of its ancestor transaction groups. Therefore, notification requests are propagated towards the root of the transaction hierarchy only. When a transaction group receives a notification request, it does the following processing:

1. If there is no copy of the object in the transaction group's cache, make a future copy. The future copy is created in the same way as they would be if the member declared an intention, except no operation machines are associated with the copy.

2. Given the path specified in the message, compute the relative path from this transaction group to the member that sent the request.

3. Add a tuple <*NID, P*> to the notification list for the operation in the object copy or future copy in the cache, where

   NID is the notification identifier from the request message, and

   P is the relative path to the requesting member.

4. If this is not the root transaction group, forward the request up the hierarchy.

12.1.2 Notification Responses

Notification responses are sent when the specified operation is done on some copy of the object in one of the members' ancestors' caches. A notification response is a tuple

<*PATH,NID*>

PATH is the relative path to the requesting member, for routing.

NID is the notification identifier that was specified in the request message.

The message is routed down the tree. At each level in the tree, the following processing is done:

1. Check *PATH*. If it is null, then this is a response to a request that was initiated locally.

   (a) Using *NID*, find the operation type and the object ID.

   (b) Process the notification as needed.

2. If *PATH* is not null, the next member in the path has the first member ID specified in *PATH*. Remove this member's ID from *PATH* and check to see if it is a valid member of this transaction group.

   (a) If the path is valid, forward the notification response.

   (b) If the path is not valid, build a notification cancellation using the notification ID in *NID*, and forward it back towards the root of the transaction hierarchy.

12.1.3 Notification Cancellations

A notification cancellation by a member terminates the corresponding notification request. A notification cancellation is a tuple

$$<NID, OP>$$

NID is the notification identifier that was specified in the request.

OP is the operation that the notification concerned.

Notification cancellations traverse the same route as notification requests, towards the root of the transaction hierarchy. When a transaction group receives a notification cancellation, it uses the operation *OP* to find the section for the notification list containing its entry for the notification in its object copy. If it is not the root transaction group, it then forwards the notification cancellation message up to its parent.

12.2 Invalidation Notifications

Transaction groups generate invalidation notifications automatically, either when some member fails or when a member aborts one or more operations. They are also generated by a member when its parent fails. They are propagated through the transaction hierarchy to all of the members affected by the failure or abort, to ensure that recovery is done properly. The details of how invalidation notifications are generated and propagated are discussed in Sections 13.2.3 and 13.2.4.

13 Transactions and Transaction Groups

The transaction hierarchy is a tree structure that describes the controlled nesting of cooperating transactions. Transaction groups are the internal nodes (non-leaves). Transactions are the external nodes (leaves). Each transaction group *TG* has a set of children, which may contain both transactions and transaction groups. These are the *members* of *TG*.

This section formally defines the data structures associated with transactions and transaction groups, the operations they may do, and the effects of those operations on the transaction hierarchy.

A member of a transaction group may be either a transaction or a transaction group. A member is created using a *member_begin* command, which initializes the basic data structures associated with the member.

During its lifetime, a member may call *member_checkpoint* one or more times to checkpoint its operations. This makes the results of its uncheckpointed operations visible, and begins the process of making them permanent with respect to the transaction group. The checkpoint of operation *op* does not complete until all operations on which *op* depends have also completed their checkpoints.

A member may also abort operations using a *member_abort* command. Aborting operations invalidates them and causes other members that transitively depend on those operations to recover in some appropriate manner. Also, when a transaction group fails, all of the operations done by its members become invalid. A recovery phase is then entered to allow the affected members to compensate for the failure.

When a member has finished, and all of its operations are either checkpointed or aborted, it may begin the process of terminating using the *member_terminate* command. The member will actually terminate once its operations are all complete.

Roughly speaking, a traditional transaction commit corresponds to a checkpoint followed by a terminate, and a traditional transaction abort corresponds to an abort followed by a terminate.

Each transaction and transaction group has an associated state. The states are defined as follows:

- *RUNNING* means that there are potentially some outstanding uncheckpointed operations by this transaction or transaction group. This occurs when new operations are done. The transaction or transaction group also enters this state when some existing operation is aborted, because it must recover before it can consider all of the existing operations stable again.

- *CHECKPOINTED* means that all existing operations are correct and final, according to the transaction group and the synchronization specification. The transaction or transaction group starts in this state, because there are no outstanding operations. It also enters this state when it checkpoints.

- *TERMINATED* means that the member has explicitly terminated itself.

Figure 14 shows the state transition diagram for the transaction states.

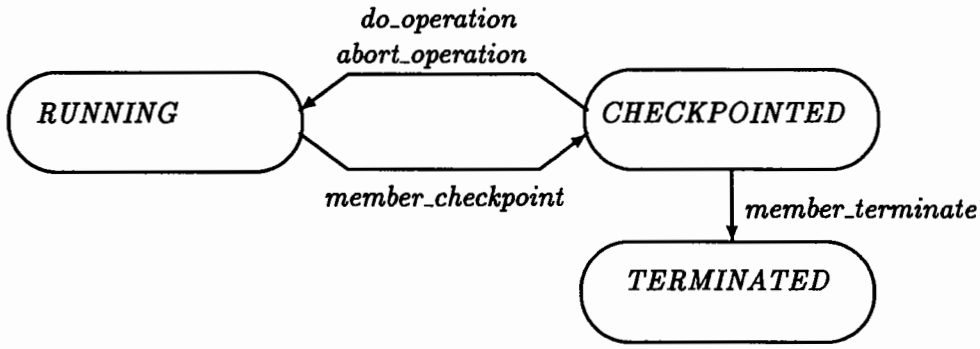


Figure 14: State transition diagram for transactions and transaction groups.

13.1 Data Structures

13.1.1 Transactions

A transaction is a sequence of operations that share some local control structure. A transaction is not necessarily atomic. The sequence of operations for a single transaction does not have to be individually correct and consistent, but must be validated according to its parent transaction group's internal protocols.

A transaction is a tuple $\mathcal{T}$, where

$$\mathcal{T} =< TID,\ N,\ P,\ LC,\ S >$$

TID is its unique member identifier,
N is its (optional) external name,
P is the member ID of its parent transaction group,
LC is its last checkpointed operation,
$S \in \{RUNNING,\ CHECKPOINTED,\ TERMINATED\}$ is its state.

13.1.2 Transaction Groups

A transaction group is responsible for a single task within the database, where the task is accomplished through the cooperation of its members. Because of this, it also controls the interaction among its members. A transaction group only allows its members to do operations as they are consistent with its own internal protocols. The transaction group's log records each of its members' operations. When one of its members fails or some operation is aborted, the transaction group uses the log to ensure that the object copies in its cache are recovered to some consistent state.

A transaction group TG acts on behalf of its members when submitting operations to its parent P. It maps sets of operations by its members that are correct according to its internal protocols into single operations by itself that are correct according to P's internal protocols. When a transaction group's history is correct, its internal protocols should guarantee that,

taken as a whole, the changes to the objects in the group's cache resulting from the operations recorded in the member entries is identical to the changes to the objects in its parent's cache resulting from the operations recorded in the group entries. If this is not true, then the transaction group's internal protocols are specified incorrectly.

A transaction group is a tuple $\mathcal{TG}$, where

$$\mathcal{TG} =< TID,\ N,\ P,\ LC,\ S,\ M,\ IP,\ EP,\ L,\ C,\ Q >$$

TID,N,P,LC,S are defined as for transactions,
M contains the member IDs of the transaction group's members,
IP specifies the transaction group's internal protocols,
EP specifies the transaction group's external protocols,
L is the transaction group's log,
C is the transaction group's cache, and
Q is the set of operations queued for its members.

13.2 Algorithms for Normal Operation

13.2.1 Member Begin

1. Create and initialize the member data structure. This structure varies depending on whether this member is a transaction or transaction group, but the following fields are initialized for both:

   TID is a newly-generated, unique identifier,
   N is the name of the member, as specified in the argument list,
   P is the member identifier of the parent, as specified in the argument list,
   LC is nil,
   S is *CHECKPOINTED*, because there are no outstanding uncheckpointed operations.

2. If the new member is a transaction, the initialization is done. Return.

3. If the new member is a transaction group, initialize the following fields in addition:

   M is the set of members specified in the argument list, or the null set if the members are to be created dynamically,
   EP is generated as specified by the parent *P*,
   L is an empty log,
   C is an empty cache,
   Q is an empty queue.

4. Examine the internal protocol specification found from the argument list. This information is used to initialize *IP*, and includes the following:

- The synchronization machine template for the machines used to enforce the member's internal synchronization mechanism.
- Any pattern machines associated with the transaction group. For each such machine, the transaction group must ensure that a future copy exists for each object named in a pattern machine. Each pattern machine is then bound to all objects relevant to it by including it in the *OM* set in the future copies of those objects.
- A correlation between external operations and the intention machine templates for the corresponding internal batched operations. One <*Operation,Machine*> pair is kept for each operation. If there is no machine specified for some external operation, then the machine in its <*Operation,Machine*> pair is set to only allow the one operation.
- If the members are not explicitly specified, then the internal protocols specify a routine to be used for member authentication.

13.2.2 Member Checkpoint

The following conditions must hold before a checkpoint request can be serviced:

- All operation machines bound to the member are in a final state. This means that all the patterns the member is participating in are complete.
- No operations are currently queued for the member.
- If the member is a transaction group, all of its members currently must be in the *CHECKPOINTED* state. This means that no member is doing something new, and any recovery that has been required of the members is complete.

A checkpoint request must be issued atomically with some operation by the member that causes all the machines bound only to it to enter a final state. If the checkpointing process cannot begin immediately because some pattern machine bound to other members is not in a final state, it waits until the checkpoint can be done. Once the member has requested the checkpoint, it cannot do any new operations until all of the checkpointed operations have entered the *PENDING* state.

The checkpointing process for member M in transaction group TG begins with the following atomic procedure:

1. Let LM be the set of member M's entries in TG's log since its last checkpoint.
2. For each object written by some operation in LM, issue a request to write that object from TG to its parent, with the new version being the last version written by some operation in LM. Note that these write requests need to be accepted by the operation machines in TG's parent.
3. Each write request issued in the previous step causes an entry to be created in both TG's log and its parent's log. In the entry at TG's level, add a parent-child dependency on the last write operation by M to the OD list.

4. Set the state *S* in all log entries in *LM* as *PENDING*.

5. Update *LC* in the member *M* to point to the log entry for the last operation issued in step 2.

6. Set the member's state to *CHECKPOINTED*.

7. Inform the member that the checkpoint is pending, and return.

At this point, the immediate processing for the checkpoint request is complete, so the member can continue with new operation requests. For a checkpointed operation to complete, no dependencies must remain in the *UOD* list in either history entry for the operation. These dependencies are removed as the associated operations complete. Once this condition holds, the following steps are taken to complete the checkpointed operation:

1. Set the *S* field in the log entries for the operation in both logs (*TG*'s and its parent's) to *COMPLETE*.

2. If the member still holds an intention to do this operation, release it.

3. For each valid operation *OP* dependent on this completed operation in either history, remove the identifier for this operation from the *UOD* list in its log entry.

Looking at this algorithm from the point of view of a transaction group coordinating a set of members, the member checkpoint indicates that it is satisfied with its work so far on its piece of the transaction group's task. The *UOD* list in the operation's history entry indicates that members can still interfere with the operation's becoming permanent. The completion of an operation indicates that all of the group's members agree that the operation is correct. The latest time a checkpointed operation should complete is when the last running member in the transaction group does its final checkpoint. At this point, all history entries must have empty *UOD* lists.

13.2.3 Abort an Operation

An abort or invalidation message contains the following information:

- The number of operations being aborted or invalidated.

- For each such operation, the *TID* of the member that did the operation, and the operation ID *(EID)* assigned by the member.

The steps required to invalidate a single history entry are as follows:

1. Set the state *S* in the history entry to *INVALID*.

2. If this is a write operation, invalidate the version it created as follows:

   (a) Set its state to *INVALID*.

   (b) Add the *OID* of the object to an *invalid_objs* set, for postprocessing.

(c) Find the *TID* of the member that has the other log entry associated with the operation (i.e. its parent if it is a group entry or a member if this is a member entry). If there is no message buffer for the member, set one up. Add the tuple <*tid, eid*> to the message buffer, where

- *tid* is the ID of the member that did the operation,
- *eid* is the member's identifier for the operation.

When a transaction group receives an abort or invalidation message, it enters the invalidation phase. In this phase, the following algorithm is done atomically:

1. Initialize the temporary set *invalid_objs* to the empty set.

2. Process the message, invalidating each log entry associated with an aborted operation. Reset the current state of each affected machine to point to the state it was in immediately before the invalid operation was originally accepted.

3. Starting with the log entry associated with the earliest aborted operation, process each log entry in sequence, as follows:

   (a) If this log entry is already invalid, do nothing.

   (b) Otherwise, this entry is initially valid. Search *UOD* in the log entry to see if this operation is dependent on any invalidated operations. If so, invalidate it.

   (c) Given that the entry is still valid, check to see if it now conflicts by running a conflict check using the current machines in the current state. If this operation now conflicts, invalidate it.

   (d) If this log entry is still valid, update the operation machines of all patterns it participated in by traversing the appropriate arcs. If this log entry is now invalid and the invalidation broke any patterns not previously broken, reset the current state in each operation machine to be the state it was in immediately before the invalid operation was originally accepted.

   (e) Add a pointer to the log entry to the *FD* list of all invalid operations on which this operation depends.

4. Again process the log from the first invalidated message. For each invalid log entry, remove the operations in the *FD* list that are now invalid. This must be done as a second pass because the entries in *FD* point forward to operations that depend on the current one. The *FD* list is used during the recovery phase to determine when a log entry can be discarded.

5. For each object in the *invalid_objs* list, set *LVV* in the associated object copy to the last valid version in the copy's *VS* sequence. Set *OM* to the set of operations and their states as saved in the last valid version. If an operation machine is bound to more than one object, ensure that the machine is restored correctly by setting the state only from the object which was operated on by the last valid operation in the machine's traversal. This sets things up to be the same as they were just before the last valid version was created, and ensures that the remaining history be correct even in the absence of any compensating actions due to recovery.

6. Set *LC* in the member to be the last operation in the log that is either *PENDING* or *COMPLETE*. Set the member's state *S* to *RUNNING*.

7. For each message buffer that was built as the operations were invalidated, generate and send an invalidation notification. If every send succeeds, the invalidation phase for this transaction group is complete. However, a send can fail for one of two reasons:

   (a) The destination member has terminated. In this case, the transaction group must decide how to recover from these failed operations itself.

   (b) The destination member has failed. In this case, the transaction group must initiate the processing for a member failure if it is not already in progress.

13.2.4 Member and Parent Failure

Member failures are explicitly detected by their parent. Parent failures are detected by their children. When a transaction group fails, both its parent and its children need to instigate recovery. In either case, when the failure is detected, it is treated as if the failed member aborted all of its operations that were not checkpointed, then terminated. Abort messages are generated appropriately.

13.2.5 Member Terminate

A member terminates when it believes that it has correctly completed its part in the transaction group's task. When a member terminates, it gives up to the transaction group its rights to cooperate in any future recovery operations that might concern it.

For a member to successfully terminate, the following conditions must hold:

- It must be in the *CHECKPOINTED* state, though some of its checkpointed operations may be *PENDING*

- If the member is a transaction group, all of its members must be terminated.

When a member is terminated, its state *S* is set to *TERMINATED*. Once all of the operations by the member complete, the member_terminate completes, and the member's data structures may be deleted.

13.3 Member Recovery

Once a transaction group has received an abort message and completed the invalidation process, its members cooperate in recovery. Members process their invalidated operations in roughly chronological order. Earlier we described the actions a member is allowed to take during recovery. They are restated here for convenience.

- Abort all uncheckpointed operations.

- Read an invalid version that the operation depended on and that the member read previously. These read operations are not arbitrated by the synchronization mechanisms, nor are they recorded in the transaction group's log. This ability to read invalid

versions lets other operations take responsibility for selected modifications made in the failed operations.

- Do a compensating read or write operation. These operations are validated by the synchronization mechanism, and are essentially the same as normal operations. Thus, records of the compensating operations are also placed in the transaction group's log.

Once a member has finished compensating for the invalid operation, it can remove all dependencies associated with this operation by calling *remove_dependencies*, which processes each item I in the UOD list for the operation as follows:

1. If I points back to a log entry LE for an invalid operation, find the forward pointer to I in the FD list of LE, and remove it. This removes all records about any dependencies on the recovered operation, and indicates that there is one less situation where a member can request to see what was invalidated. If its FD list is now empty, splice LE out of the log and splice any version it created out of the version sequence in the object copy.

2. Otherwise, do nothing.

Members must recover from all invalid operations before they can checkpoint again.

14 Summary and Future Directions

The design process is an iterative one, and often involves groups of designers cooperating to complete a particular task. The concurrency control and recovery protocols used in traditional databases often get in the way of such iterative, cooperative processes.

We have defined a new structure called a *cooperative transaction hierarchy* that allows us to implement more flexible transactions for design applications, and to support and control their interactions. A cooperative transaction hierarchy has a tree structure that changes as the requirements of the cooperating transactions change. Internal nodes in the hierarchy are called *transaction groups*, and external nodes are called *transactions*. A node M that has a parent transaction group P is called a *member* of P.

In a cooperative transaction hierarchy, only operations are atomic. Transactions are non-atomic collections of operations. A single transaction does not necessarily operate on the database in way that maintains consistent data. A transaction group defines a unit of work, or *task*, associated with the design effort. Its members cooperate to complete the task. The members can share intermediate versions of objects, but the underlying read and write operations that implement the sharing are restricted to conform to user-defined *pattern* and *conflict* specifications. The transaction group also controls the way *notification* messages are passed among its members.

Transaction groups work on local copies of objects in their cache. We define *intentions* to enable members to reserve the capability to do single operations. We use intentions to allow us to control the concurrent access of different copies of the same object so that they do not conflict. We can also use *batching* within the intentions, to allow a transaction group to do several operations and collapse them into a single operation at the next level up in the hierarchy. Batching also allows us to garbage-collect intermediate versions after they are no longer needed.

Operation machines are an augmented form of finite state automata that define the patterns and conflicts specified for a transaction group. We use collections of operation machines to control the interleaving of operations in a transaction group. Correctness can be defined based on these collections of operation machines, because each operation machine enforces a particular pattern and its associated conflicts. Thus, the collection of operation machines associated with a transaction group corporately enforces all of the patterns and conflicts.

Operation machines also define dependencies among the operations, which we maintain in the log. Thus, we can limit the amount of information that is invalidated as a result of a member abort or failure. Only the specific operations that are dependent on other invalid operations and those operations that now conflict in the history must fail. If dependencies were defined at the transaction level, then all operations of all transactions that had some operation that was either invalid or dependent on an invalid operation would fail.

The invalidation of an operation breaks patterns. Each affected pattern must be undone back to the point where the first invalid operation occurred. This process may introduce new conflicts, and the conflicting operations must also be invalidated. Recovery can then be done cooperatively, by simply allowing the members to continue normally from the point where the operation failed. The recovery mechanism preserves as much as possible and encourages the members to recover forwards using compensating actions rather than backwards by aborting.

Cooperative transaction hierarchies are defined in terms of read/write semantics only. They can be used to mimic traditional, one-level, serializable databases. However, they can also allow much more flexibility. In the future, we hope to be able to characterize their behavior more fully, including determining the properties of deadlocks, and determining under what situations operation machines can be dynamically added to a transaction group. We also hope be able to extend this work to allow richer sets of operations and/or version trees.

A Arguments for Correctness

A.1 Correctness of the Log

Theorem 1 *In normal operation, the ordering of operations in the log of a transaction group reflects a correct history.*

The total ordering of operations in the log of a transaction group must be an extension of the *happens_before* partial ordering (<) in the group's history. This happens-before ordering conforms to the following constraints (copied from Section 7.2):

1. It contains an entry for every read or write operation *op* in each of its member's executions that is not followed by an *abort(op)*.
2. It contains an entry for every read or write operation *op* in its own execution that is not followed by an *abort(op)*.
3. For a specific member, if *(op2 < op1)* in the member's execution, then *(op2 < op1)* in the group's history.
4. For the transaction group, if *(op2 < op1)* in the group's execution, then *(op2 < op1)* in the group's history.
5. Any operation *op1* in the history that is dependent on some operation *op2* has *(op2 < op1)*.

Lemma 1 *The set of operations recorded in a transaction group's log satisfies constraints 1 and 2 on the history.*

The only time an operation is logged is during the actual execution of the operation. The steps of doing the operation and placing the entries in the member and the transaction group logs are atomic. Therefore either the operation is done and the entries added to the log, or no operation is done and no entries are added. □

Lemma 2 *A transaction group's history satisfies constraint 5.*

There are three types of dependencies, *pattern dependencies*, *reads-from dependencies*, and *parent-child dependencies*.

Pattern dependencies occur when two operations are relevant to the same pattern. These dependencies are encoded in the operation machines. For each operation at any time, there are a set of operation machines that enforce the patterns relevant to that operation. When the operation executes, all of the arc traversals are done atomically to ensure correctness. This blocks out any simultaneous traversals associated with other operations. Therefore, two arc traversals corresponding to two different operations relevant to the same pattern cannot occur simultaneously, but must be executed in some order allowed by the pattern.

Reads-from dependencies occur when one operation *op1* reads an object version created in the transaction group's cache by some other operation *op2*. Clearly, *op1* occurred before

op2, because *op2* sees the results of *op1*. Therefore, their history entries should be ordered with *op1* < *op2* because the execution order is reflected in the order of entries in the history.

Parent-child dependencies occur between operations representing members at different levels in the transaction hierarchy. A read operation by the transaction group must happen before a read operation by one of its members, because the member's read sees the results of the parent's read. This is the same as the argument for reads-from dependencies above. Similarly, the last write by a member must happen before the transaction group's write, because the transaction group's write sees the effects of the last write by a member. □

Lemma 3 *A transaction group's history satisfies constraints 3 and 4.*

By Lemma 2 and the constraints on ordering in a member execution. □

Lemma 4 *The complete ordering of the log is an extension of the partial ordering of the history.*

If two operations *op1* and *op2* do not depend on each other, then they are unrelated in the partial order, so their order in the log does not matter. If *(op1 < op2)*, then *op1* was executed before *op2* by Lemma 2. Because doing the operation and logging it are atomic, then *op1* occurs before *op2* in the log.

We have shown that the algorithms maintain the log of each transaction group in a way that enforces the formal constraints on histories. As long as the individual log entries are themselves well-formed, all logs reflect a correct history. □

A.2 Correctness of a History After Failure and Recovery

Theorem 2 *The history remaining after operations have aborted is correct once the invalidation phase has completed.*

The invalidation phase removes all operations from all transaction group histories that are transitively dependent on any operation that was aborted. Pattern, reads-from, and parent-child dependencies are all maintained in the *OD* list in the log entry for each operation. The pattern dependencies are determined from the operation machines, and the reads-from and parent-child dependencies are determined using back-pointers from object versions to the history entries for the operations that created them.

For operation *op1*, the *UOD* list in its log entry maintains the dependencies on operations that can still affect the validity of *op1*. If an entry to *op2* is in *op1*'s *OD* list but not its *UOD* list, *op2* is complete and therefore cannot be invalidated. If *op2* is in *op1*'s *UOD* list, *op2* is not complete and therefore can be invalidated, causing *op1* to be invalidated as well.

Because all dependencies are processed in chronological order and all dependencies point backwards in time, we know that all operations on which a given operation depends will be processed during the invalidation phase before that operation. Therefore, at the time the invalidation phase processes an operation, it has all the information needed to determine whether or not it is transitively dependent on an aborted operation.

The invalidation phase also removes all operations from transaction group histories that are incorrect because they now conflict. Conflicts occur at specific points in time, when

patterns are at specific points. Each operation in the history is checked for conflict based on what the current states of the operation machines would be in the new history at the point in the history that the operation's effects would take place. Any new conflicts thus would be correctly detected and the operations invalidated.

The invalidation phase correctly breaks patterns by invalidating all operations following an invalid operation in a traversal. The operations that remain must participate in correct traversals, though some of them may be incomplete. □

Theorem 3 *After the invalidation phase, the log and the database correctly reflect this new history.*

Every operation that is invalidated during the invalidation phase no longer forms a part of the history and is explicitly deleted from the log. Therefore every operation that remains in the history is recorded in the log, and vice versa. Since we neither reorder operations in the part of the log that remains nor reorder the operations in the part of the history that remains, then the total ordering of the log must still be consistent with the partial ordering of the history.

When an operation is invalidated, any version it created is invalidated as well, removing the effects of the operation from the database. Any successive versions are either invalidated because of reads-from or pattern dependencies in the operations that wrote them, or are still valid because they did not depend on the earlier version. Therefore the last valid version written is the last one that has no dependencies on any invalid operations. □

Theorem 4 *After the recovery phase, the log reflects a correct history.*

We show that each action a transaction can take during recovery is individually correct.

If abort requests are issued during recovery, we know from the previous argument that the subsequent invalidation phase maintains correctness. Assume by induction that the recovery phase also maintains correctness. This is acceptable because there are only a finite number of operations. Therefore, even if all members decide to abort, we eventually reach cases where a member that has an operation invalidated has no valid operations left to abort.

If a member reads an invalid version that it has already seen, this is equivalent to the situation where it kept a copy of the invalid version. Therefore, the read does not give the member any new information. Since this read is not a part of the history, it does not affect it.

If compensating operations are issued, they must be accepted by the operation machines. Since the invalidation phase leaves each machine in the state they were in immediately before its traversal's break point, these operations continue existing traversals correctly. Therefore, they maintain correctness as well.

Since each recovery operation individually preserves correctness, the recovery phase as a whole is correct. □

A.3 Operation Completion

Theorem 5 *An operation completes when it is checkpointed and all operations that precede it in the history are complete. Once an operation completes, it cannot be undone by any abort by any member.*

There are three ways that an operation can be invalidated. First, it can be aborted by the member that issued the operation. However, completion happens after the operation is checkpointed, and checkpointing marks the end of the time the member can abort the operation. Therefore, a complete operation cannot be aborted by the member that issued it.

The second way to invalidate an operation is if it is (transitively) dependent on an operation that is invalid because of an abort by some other member. However, all pattern and reads-from dependencies point backwards in the history of the transaction group. Parent-child dependencies occur between operations at different levels in the cooperative transaction hierarchy. A parent-child dependency can occur when the read operation of a member depends on the read operation of the transaction group itself. However, the read operation of the transaction group precedes the read operation of the member in the member's history, and therefore the member's read cannot complete until the transaction group's read completes. A parent-child dependency can also occur when the write operation of a transaction group is dependent on the write operation of a member. In this case, the write operation of the member precedes the write operation of the transaction group in the history, and therefore the transaction group's write cannot complete until the member's write completes. Thus, a complete operation cannot have dependencies on any operation that is not complete. Therefore, it cannot be invalidated because of any dependency.

The third way to invalidate an operation is if it conflicts in the new history generated during the recovery process. This occurs because one or more operation machines are not in the same state in the new history as they were in the old history at the time the operation was executed. If a conflict is specified in the current state of an operation machine in the new history, it must be invalidated for the new history to be correct.

An operation machine can be in a different state in the new history as it was in the old history only if the sequence of operations that precedes it in the new history is different from the sequence that precedes it in the old history. However, if all operations that precede this one are complete, they cannot be undone (we can assume this by induction). Therefore, the part of the new history that precedes this operation cannot be different from the part of the old history that precedes this operation. Thus, this operation cannot be invalidated because of conflicts that appear during recovery. □

Theorem 6 *All operations done by all members in the transaction hierarchy reach the COMPLETE state eventually, provided that all the members checkpoint all their operations eventually.*

Definition A.1 (Combined Log) *The* combined log *of a cooperative transaction hierarchy as the sequence of log entries produced by processing the hierarchy from the bottom up, taking each sequence of operations S by the members that batched into a single operation O by the transaction group, and replacing the entry for O in the transaction group's log with the entries for the sequence of operations SO.*

Operations in the combined log complete in the same sequence as they do in the logs of the different transaction groups in the cooperative transaction hierarchy. Thus, we can now prove the theorem by induction on the combined log.

(Basis) The first operation in the combined log completes as soon as the member that did the operation checkpoints by the definition of completion, because there are no operations preceding it historically.

(Induction) Assume all operations before this one in the log complete eventually. We have two cases:

1. All operations before this one in the combine history complete before this operation checkpoints. In this case the operation will complete at the time it checkpoints.

2. This operation checkpoints before all operations preceding it in the combined history complete. Since we know by the induction hypothesis that all of these operations complete eventually, this operation will complete once the last incomplete operation that precedes it completes.

□

References

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

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *ACM SIGMOD Proceedings.* ACM, 1989.

[FZ89] Mary Fernandez and Stanley Zdonik. Transaction groups: A model for controlling cooperative transactions. In *Third International Workshop On Persistent Object Systems*, January 1989.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *ACM SIGMOD Proceedings*, pages 249–259. ACM, 1987.

[HR87] Theo Haerder and Kurt Rothermel. Concepts for transaction recovery in nested transactions. In *ACM SIGMOD Proceedings*, pages 239–248. ACM, 1987.

[KKB87] H. Korth, W. Kim, and F. Bancilhon. On long-duration CAD transactions. *Information Systems*, 13, 1987.

[KLMP84] W. Kim, R. Lorie, D. McNabb, and W. Plouffe. A transaction mechanism for engineering design databases. In *VLDB Conference Proceedings*, Singapore, 1984.

[KSUW85] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes. A transaction model supporting complex applications in integrated information systems. *ACM SIGMOD Proceedings*, 1985.

[LP83] R. Lorie and W. Plouffe. Complex objects and their use in design transactions. In *Databases for Engineering Applications*, pages 115–121. ACM, May 1983.

[Lyn83] Nancy A. Lynch. Multilevel atomicty - a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems*, 8(4), December 1983.

[Mos85] J. Eliot B. Moss. *Nested Transactions: an Approach to Reliable Distributed Computing.* MIT Press, 1985.

[RM89] K. Rothermel and C. Mohan. ARIES/NT: A recovery method based on write-ahead logging for nested transactions. Technical Report RJ6650, IBM, January 1989.

[Ska] Andrea Skarra. Personal communication.

[Ska89] Andrea Skarra. Concurrency control for cooperating transactions in an object-oriented database. *SIGPLAN Notices*, 24(4), April 1989.

[SZR86] Andrea H. Skarra, Stanley B. Zdonik, and Steven Reiss. An object server for an object-oriented database system. In *Proceedings of the International Workshop on Object-Oriented Database Systems.* ACM/IEEE, 1986.

Contents