Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications[†]

Marian H. Nodine and Stanley B. Zdonik
Brown University, Providence, RI 02912

Abstract

Traditional atomic and nested transactions are not always well-suited to cooperative applications. Cooperative applications place requirements on the database which may conflict with the serializability requirement. We define a new transaction framework, called a *cooperative transaction hierarchy*, which allows us to relax the requirement for atomic, serializable transactions. Each internal node (*transaction group*) in the transaction hierarchy can enforce its own constraints on how objects can be shared among its children (*members*).

Patterns specify the constraints imposed on an operation history for it to be correct. At a given node in the hierarchy, we use a type of augmented finite state automaton called an *operation machine* to enforce correctness. We provide *intentions* to manage the propagation of object copies and their associated privileges through the transaction hierarchy. We show that using intentions enforces that the overall history of the hierarchy is correct.

Logs record the information required by the cooperative transaction hierarchy for recovery. We specify what must be logged for each transaction group, which includes information about the transaction group's execution and about the dependencies among operations in that execution.

Finally, we show how to use cooperative transaction hierarchies to enforce multilevel atomicity [Lyn83].

[†]Support for this research is provided by IBM under contract No. 559716, by DEC under award No. DEC686, by ONR under Contract N00014-83-K-0146, by Apple Computer, Inc., and by US West.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

1 Introduction

Cooperative applications such as CAD tools have generated requirements for underlying database support that do not conform well to traditional database structure. Traditional databases were developed to support on-line data processing applications and are optimized for short, atomic transactions. Cooperative applications interact with each other. When using a database, cooperative applications tend to generate long transactions that are not necessarily atomic. However, we can assume that the users of such applications are human and can respond intelligently to certain problems.

Transactions that support cooperative applications are correct when they interact and share data only in ways acceptable to the application environment. We do not believe that a single, monolithic correctness criterion such as serializability suffices. We provide a hierarchical scheme for these cooperative transactions that allows different parts of a shared task to use different correctness criteria, and a mechanism by which these correctness criteria can be explicitly programmed.

As an example of a design application, we will use Lynch's Utopian Planning example [Lyn83]. There are several people working on a city design. Objects represent buildings, parks, maps, etc. Each person is responsible for a specific task, though a group of people may be working on the same object. For example, several architects may be working to design City Hall while a landscaper plans the shrubs which surround it. As the design progresses, the members of the group interact naturally, both informally and through the objects in the database. For example, a change in the size or placement of the windows in City Hall may affect the shrubs surrounding it. These new problems must be dealt with by other designers before the initial changes can be "committed".

We see from this example that each designer is responsible for the set of changes to the database associated with his specific task, but these changes may interact with the work of other designers. Therefore, the changes associated with a specific task maintain only partial consistency in the database. The designers also

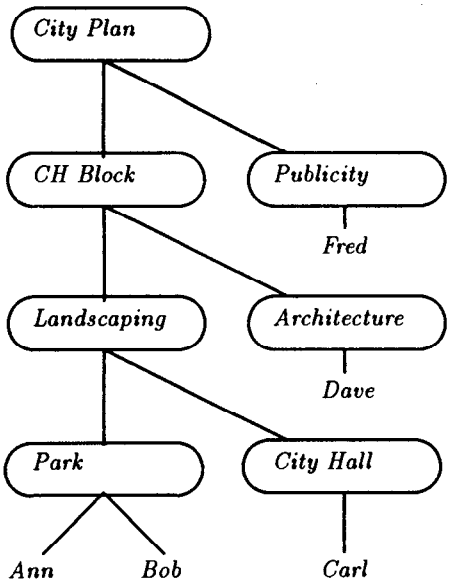

Figure 1: Utopian Planners Hierarchy.

may iterate through several refinements or changes in the initial design as they work.

The design process often decomposes hierarchically. For example, the city planning task breaks up into subtasks such as parks design, traffic design, etc. There are additional subtasks such as publicity which do not relate to the design directly. Each of these subtasks may be constrained in one or more ways, e.g., a park has to fit in the space allocated for it. These subtasks may be divided in turn among several designers.

We propose a structure called a *cooperative transaction hierarchy* for modeling and managing interactive design applications. The hierarchy for a particular design task is typically structured according to the task's natural decomposition. For example, a piece of the city planning task described above is shown in Figure 1. The external nodes (e.g. Ann, Bob) represent the transactions associated with the individual designers. The internal nodes (e.g. Publicity, Landscaping) are called *transaction groups*. The root of the tree is the transaction group which manages the city plan database. Except for this group, each other transaction or transaction group is a *member* of some other transaction group (called its *parent*).

In our model, cooperative transactions are collections of atomic read and write operations by a single member. Cooperative transactions need not be serializable or atomic. Transaction groups correspond to some task, and that task is done cooperatively by its members. The root transaction group, which is at the top of the transaction hierarchy, has the special task of managing the database.

Because cooperative transactions are not necessarily atomic, we define a correctness criterion for our database different from serializability. Correctness is enforced within a transaction group. That is, each transaction group has a notion of how its members should operate and how their operations should interleave in its own history. We use a notion of *patterns* and *conflicts* to specify correct histories for each transaction group. *Patterns* define acceptable sequences of operations in a transaction group's history. For example, a pattern might say, "Ann must read the *Shrubs* object before writing it." *Conflicts* are defined within the context of patterns, and specify sequences of operations that must not occur. For example, "Once Ann has written the *Shrubs* object, Bob cannot read it until Ann's transaction commits." Every transaction group has many patterns and conflicts defined for the objects and members, and they work together to define correctness.

In addition to enforcing correctness within a transaction group, we coordinate operations by members of different transaction groups on a single object so that the overall set of operations at the level of the root transaction group is correct. Problems occur when some member M tries to read an object O, do several modifications over time, then write the results. When M finishes its modifications its write operations in the member are collapsed *(batched)* into a single write operation in the transaction group. No other operations that conflict with the batched write operation should occur in the transaction group during this time span. *Intentions* (see section 4.2) are used to hold the capability in the transaction group to do the batched write operation while the sequence of modifications is being done in the member.

Because we have defined a new correctness criterion, we need to *log* different information about the history of each transaction group so that the system can restore the database to a correct state after a failure. We specify what information must be kept in the log to ensure that recovery is correct.

2 Related Research

Several proposals have been made for supporting more flexible and long-lived transactions. The approaches related to ours include nesting the transactions [Mos85, KKB87], augmenting traditional locking protocols [SZR86], and specifying a longer transaction as an envelope that contains a sequence of shorter transactions [GMS87].

Nested transactions [Mos85] provide a framework for

decomposing a transaction hierarchically. A transaction may define subtransactions that execute concurrently. The subtransactions must all be serializable with respect to the parent transaction. If a subtransaction fails or aborts, the parent transaction has the option to restart it. Kim et.al. [KLMP84] presents a three-layer hierarchy tailored for design transactions. It differs from nested transactions in that it allows copies of objects to be "checked out" from a parent transaction into a private database.

Nesting works well in design environments where the tasks decompose easily into small, independent subtasks. However, serializability prohibits cooperation and data sharing among the subtasks. Haerder and Rothermel [HR87] examined what was required to allow data sharing while preserving the serializability of subtransactions, and concluded the behavior after a failure was unacceptable due to the large number of dependencies that were formed.

Klahold et.al. [KSUW85] proposed a transaction model that allows cooperating *user transactions* to work together in the context of a *group transaction*. While the group transactions maintain a two-phase locking protocol, the user transactions within a group transaction may share data. The group transactions use a relaxed locking scheme that allows data sharing, but does not guarantee that the data remains consistent.

The constraint-based models [KKB87] allow more cooperation by relaxing serializability at the lower levels of the transaction hierarchy. At these levels, transactions can cooperate as long as each transaction preserves its specified consistency constraint. The constraints are enforced using a modified locking protocol *(predicatewise two-phase locking)*. This model defines the constraints implicitly; the users cannot tailor them to the task at hand. Although it weakens serializability, it does not allow as much expressiveness as our model. For example, it does not allow a designer to read an object while another designer is writing it.

Multilevel atomicity [Lyn83] is a framework for relaxing atomicity. It allows the specification of a hierarchy of breakpoints between operations for a particular transaction execution. The breakpoint specification states how other transactions can interleave their operations with this one (see section 6). Multilevel atomicity assumes that the set of transactions in the system is fairly static; adding a new transaction requires specifying its relationships to all other transactions. It is also not clear how to specify the breakpoint hierarchy for an interactive transaction before it has executed.

Other approaches used to increase the flexibility of design transactions include the NT/PV model of Korth and Speegle [KS90], the flexible transaction model proposed by Kaiser [Kai90] and operation transformation for groupware systems by Ellis and Gibbs [EG89].

An apporach to process synchronization similar to our transaction synchronization mechanism is *path expressions* [CH84]. Path expressions are regular expressions that define how operations on a single module should be synchronized. Our notion of *patterns* is more general, in that a pattern can be defined over multiple objects. Also, patterns can restrict *who* does an operation, as well as when an operation may occur.

In this paper, we have taken the use of *patterns* and *conflicts* to specify correct histories from Skarra's work [Ska89]. Her model uses the methods defined on abstract data types in the database as the underlying operation set, while we restrict our operations to read and write operations. The basic concept of transaction groups that we use was first defined by Fernandez and Zdonik [FZ89].

3 The Model

The design process tends to decompose hierarchically. In our model, we allow cooperative transactions and transaction groups to be nested in a tree-like manner to form a *cooperative transaction hierarchy*, with transaction groups as the internal nodes and cooperative transactions as the leaves. A *transaction group* contains a set of members that cooperate to do a single task. It actively controls the interaction of its cooperating members. A *member* may be either an individual cooperative transaction or another transaction group. No member may have more than one parent.

3.1 Transaction Groups

Each transaction group is tailored to the task its members are working on. The procedures and rules defining the operation of a transaction group (e.g. its definition of correct operation) are called its *protocols*. A 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 identical to the internal protocols of its parent.

A transaction group has a local set of object versions being accessed by its members. There may be several versions of an object scattered throughout the hierarchy. For a given transaction group, the object versions in its set may be accessed (read or written) by any member below it in the hierarchy, but by no other members. The root transaction group has a version of every object in its set, and this version may be

accessed by all members.

An object version is copied automatically into a member transaction group's set from that of its parent when the member initiates a read operation on that object. A new version of the object is written back to the parent transaction group's set when the member indicates that it has finished modifying the object. All read and write operations must be allowed by the transaction group's internal protocols. To its members, a transaction group appears to be the database server responsible for storing objects and controlling member access to objects.

Because the members of a transaction group cooperate, they are no longer self-contained. This means that the sequence of operations by a single member might not leave the database in a correct state. The transaction group is responsible for ensuring that the combined history produced by its members is correct according to its internal protocols. These protocols specify the group's notion of correct interactions among its members, among other things.

When a transaction group M is itself a member of another transaction group TG, 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. This translating function also hides the internal operations of the transaction group's members from its parent, generally by collapsing sequences of operations by the members into shorter sequences of operations by the transaction group itself. For example, a write operation followed by a sequence of read and write operations on a single object in the member may be collapsed into a single write operation by the transaction group. This allows encapsulated groups of non-serializable transactions to be members of other serializable groups.

Because we cannot determine the transaction groups a priori, the transaction hierarchy can be modified dynamically. The root transaction group, whose task is to maintain the database, always exists. Other transaction groups and transactions may join the hierarchy as the design effort progresses.

3.2 Cooperative Transactions

In our model, cooperative transactions represent designers or intelligent design applications. Because the lifetime of a design task is indeterminate, we assume that cooperative transactions are long-lived and open-ended. We also assume that cooperative transactions can interact with each other both externally and through the objects in the database. Thus, they may have a reasonable notion of what the other members in their transaction group are doing.

During its lifetime, a cooperative transaction issues read and write operations on object versions in its transaction group. These operations are executed by the transaction group if they conform to its patterns and conflicts. Operations may be queued or refused by the transaction group. Queueing an operation means that the cooperative transaction is notified when it can resubmit the operation. In the meantime, it is allowed to dequeue its operation and/or continue processing. As with transaction groups, operations are individually checkpointed or aborted by the cooperative transaction as the design task progresses.

3.3 Operational Overview

This section describes how members begin, access the database, and terminate. These operations are similar in spirit to the familiar transaction begin, commit, and abort operations, though there are differences because cooperative applications place different requirements on the object server.

Members are created using the *member_begin* command. The command specifies the new member's name and its parent. If the member is a transaction group, its internal protocols must be 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 transaction group's object versions) into external operations (i.e., operations by the transaction group on its parent's object versions).

The new member also must authenticate itself to its parent, using its parent's authentication procedure.

There are several functions that are not specified during the member definition process, but rather are inherent in the way the transaction hierarchy is managed. These include the transaction group's external protocols, which are inherited from its parent, and the rules for managing object versions.

[1] An equivalent history is either an identical history or one where the operations by the members of M are collapsed into a shorter sequence of operations by M on TG's object versions that affect the data in the same way.

Once a member has been established, it may operate on the database in any way allowable by the synchronization mechanism defined by its parent's internal protocols. When the sequence of operations in a transaction group completes some set of changes, they can be checkpointed using a *member_checkpoint* operation. The decision to checkpoint may be made either manually or automatically. Because the patterns capture the structure of all interactions among the members of the transaction group, all patterns a member participates in must be complete when it checkpoints.

The checkpoint operation causes the internal operation history containing the operations by the members to be mapped into an equivalent external history containing the operations by the group. Since we restrict the operation set in this paper to *(read, write)*, this means that new versions of the objects that have been modified by the members of the group are propagated up to its parent. To do this, the transaction group issues its own (external) write request to the parent for each such object. These requests must be acceptable to the parent. Each write introduces new object versions to the parent transaction group, making the changes accessible by the transaction group's other siblings. The changes also become 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[2] the *member_checkpoint* procedure only guarantees that the new object versions 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 operation history of the transaction group, and that any object version created by the operation no longer properly exists in the object server. Operations can be aborted either because the member actually failed in some way, causing its transaction group to abort the uncheckpointed operations by the member, or because 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 need not 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. Other operations may depend on the aborted operations, for example if one of the operations created a version that was read by another member, or if other operations participate in the same pattern. These operations also must become invalid. The invalidation can propagate through the transaction group's history to all operations that are transitively dependent on the aborted operations. Thus, many members' operations may be invalidated, and these members must in turn recover as much as possible any invalidated changes they made.

When a member is completely finished and all its valid operations are checkpointed, it 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 M terminates, its transaction group becomes responsible for recovery if any of M's operations are subsequently invalidated.

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 selectively commit or abort. The second reason 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 other members closer to the leaves of the hierarchy.

At any time, each member in the transaction hierarchy is in one of the following states:

- *RUNNING* means there may be some outstanding uncheckpointed operations by this member.
- *CHECKPOINTED* means all existing operations are correct and final, according to the member and the transaction group's synchronization specification.
- *TERMINATED* means the member has explicitly terminated itself.

3.4 Data Structures

A cooperative transaction is a sequence of operations that share some local control structure. It 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 cooperative transaction is a tuple

$$CT =< TID, P, S >,$$

where

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

TID is the unique member ID,

P is the member ID of the parent transaction group,

$S \in \{RUNNING, CHECKPOINTED, TERMINATED\}$ is the member's state.

A transaction group is responsible for a single task within the database, and that task is accomplished through the cooperation of its members. Because of this, it also controls the interaction among its members, only allowing operations that are consistent with its internal synchronization protocols. The transaction group records each of its members' operations, as well as all of its own operations. When one of its members fails or when some operation is aborted, the transaction group also ensures that its object versions are recovered to some consistent state.

A transaction group acts on behalf of its members when submitting operations to its parent. This means that it must map sets of operations by its members which are correct according to its internal protocols into single operations by itself which are correct according to its parent's internal protocols.

A transaction group is a tuple

$$\mathcal{TG} =< TID, P, S, M, IP, EP >,$$

where

TID, P, S are defined as for cooperative transactions,

M contains the member IDs of TG's members,

IP specifies TG's internal protocols,

EP specifies TG's external protocols,

4 Synchronization

Synchronization in this context is the constraining of operations in each transaction group to conform to some correct history. The patterns and conflicts defined in the root transaction group indicate the synchronization constraints on the database. The patterns and conflicts defined in other transaction groups indicate their synchronization constraints. There is also a mechanism to arbitrate permission to do operations on the various versions of an object. This is done using a variant of a checkout/checkin process called an *intention*.

4.1 Operation Machines

Synchronization protocols for cooperative transaction hierarchies need to control not only the concurrent access of objects, but also the order in which different objects are accessed by different members. They need not necessarily constrain the members' operations to a specific execution; rather they should specify the general form of the allowable member interactions. We use patterns and conflicts to specify required and prohibited operation sequences by the members of a transaction group, as well as the interleaving of the members' operation sequences.

Operation machines are user-definable synchronization mechanisms for specifying patterns and conflicts. They were first proposed by Skarra [Ska90]. An operation machine ($\mathcal{OM}$) is a finite-state automaton. Each transition in an operation machine is labeled with the symbol σ that defines the operation associated with it, where

$$\sigma =< M, O, o, P >$$

and

$M \in \{any, m_i, \overline{m_i}\}$ is the *TID* of some member, where *any* is any member, m_i identifies some member i, and $\overline{m_i}$ is any member except m_i.

$O \in \{r, w\}$ is an operation, where r is read and w is write,

o is an object identifier,

$P \in \{a, r, q\}$ is a return value, 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 on an object by some member. They are annotated with return values that are either *accept* if the operation conforms to the pattern, *refuse* if the operation conflicts, or *queue* if the operation conflicts now but may conform to the pattern if done later. The lack of a transition for an operation from some state indicates that the operation is not relevant to the pattern at that time, therefore the pattern cannot cause the operation to be rejected or queued. The final states of an operation machine indicate when its pattern is complete in the history. A database is consistent if every member checkpoints only when every machine associated with the member is in a final state.

At any given time, a transaction group uses its entire set of operation machines together to enforce the correctness of its history. Each operation may participate in one or more patterns and must be accepted by the operation machines enforcing each such pattern.

As an example, let us assume that the members of the *CH_Block* transaction group are currently working on the two objects *CH* and *Park*. The member *Arch* is modifying the design of the facade of City Hall, and the member *Land* is adjusting the shrubs around City Hall to fit the new design. Figure 2 shows the operation machines defined for the *CH_Block* transaction group that enforce the different constraints on the history. The operation machine in Figure 2(a) enforces

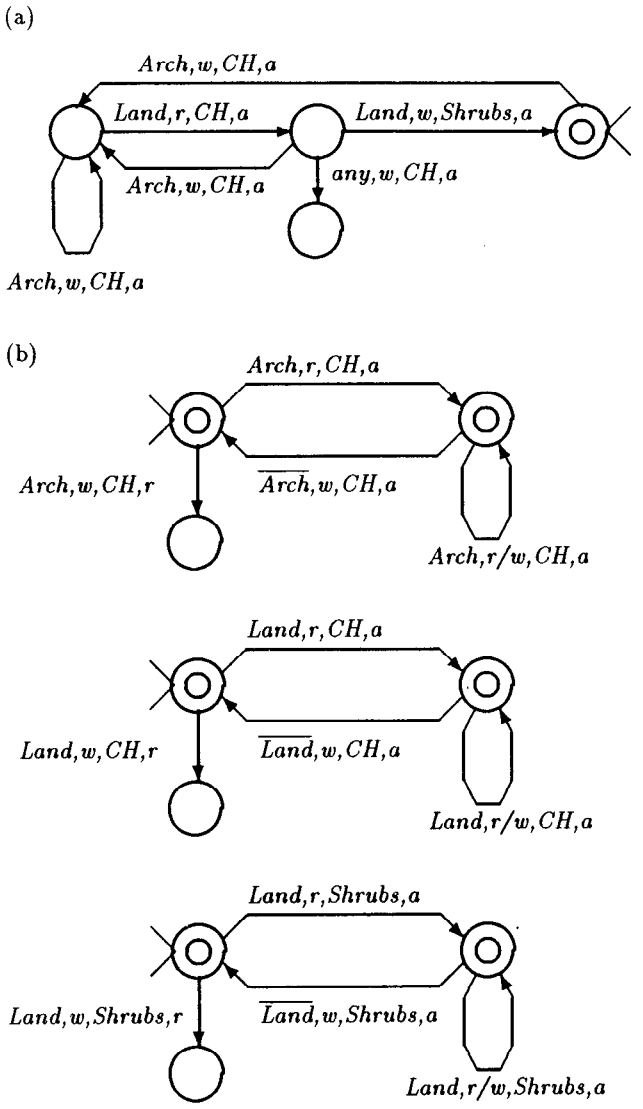

Figure 2: Operation machine examples: (a) Enforces a pattern that spans multiple members and objects; (b) Corporately enforce a cooperative synchronization protocol.

that the member *Land* must read the last version of the *CH* object last written by the architects *(Arch)* before redoing the shrubs around City Hall. It ensures that the *Shrubs* object is current with the latest City Hall design before the modification task is "complete". Figure 2(b) shows the three operation machines that ensure that changes to objects are not overwritten. For instance, the first machine ensures that member *Arch* reads the latest version of the *CH* object before modifying it. These machines specify an underlying cooperative synchronization protocol. In a transaction group, there is one such machine for each member-object pair where the member is currently interacting with the object.

As an example of how operation machines work, consider the machines in Figure 2. If all machines are in the start state, the operation $<Arch,w,CH>$ would be refused by the first machine in (b). However, the operation $<Arch,r,CH>$ would be accepted, because it is accepted by the first machine in (b) and not relevant to the machine in (a) or the other machines in (b).

For comparison, let us assume that we want to enforce serializability at the level of the *City Plan* transaction group. Since serializable transactions are independent, there are no patterns involving multiple members and objects. We only need to define a pattern that mimics read/write locking for each member-object interaction. Since there is one operation machine per pattern, the number of operation machines in a transaction group that enforces serializability is roughly the same as the number of locks there would be if a locking protocol were used.

From these examples we can begin to see how a transaction group's operation machines are created and deleted. Machines such as those in Figure 2(b) all serve the same purpose but for different member-object interactions, and consequently differ only in the member and object bindings. A transaction group administrator can define an *operation machine template* for the specific purpose (e.g., to prevent overwriting), and the transaction group can instantiate operation machines automatically from the template as needed. Operation machines such as the one in Figure 2(a) are one-of-a-kind, and must be defined explicitly by the administrator.

Since we use operation machines to enforce the patterns and conflicts defined at any given point in an operation sequence, we can formalize our definition of correctness within a transaction group in terms of operation machines. We define a *traversal* as a sequence of operations associated with consecutive accept arc transitions in an operation machine, beginning at the start state and ending at the current state. A *complete traversal* ends at some final state. For each operation in a history, we know which transitions occurred as a result of its execution. Define the sequence Π_{OM} associated with operation machine OM as

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

That is, Π_{OM} is the projection of the pattern defined by the machine OM from the history. These operations were acceptable according to the pattern specification

at the time they were executed. A history is *correct* when all projections Π_{OM} for the machines *OM* that were active during the history are traversals. A history is *complete* when all such projections are complete traversals. For more details, see [NFSZ90].

4.2 Intentions for Maintaining Overall Correctness

Each transaction group in a hierarchy has a local set of versions of the objects that it is currently accessing. Since multiple versions of an object may exist, there is a potential for inconsistency. We use *intentions* to control the way members access object copies in the transaction hierarchy so that the correctness of the overall sequence of operations on each object is maintained. Intentions reserve the capability for a member to do a single operation on a single object in its parent. When a member *M* of a transaction group *TG* is granted an intention to do an operation, *TG*'s version of the object is restricted in such a way that no other operations can be done that will later cause the intended operation to be rejected.

When the member *M* is a transaction group, the intended operation represents the combined effects of a sequence of operations by *M*'s members on the object. For example, many reads and 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 conflicting operations from being processed. *M* may *release* the intention at any time.

Intentions differ from locks in that they reserve the capability to do only one operation, while locks reserve the capability to do an arbitrary number of operations from a fixed operation set. Locking is used to prevent transactions from interleaving their operations, and would further restrict the allowable operation sequences in the transaction group. Intentions are more flexible because they do not generate these restrictions.

A member may request an operation without having acquired an intention. The operation is still done, provided that it is currently acceptable according to the patterns and conflicts defined in its parent. It also may be queued or refused.

4.2.1 Intention Machines

Two restrictions need to be enforced for intentions to work properly. At the transaction group level *(TG)*, we need to ensure that no other member does an operation that conflicts with the operation requested by member *M*. If *M* is itself a transaction group, we need to ensure that its members do operations only according to a pattern that will batch to the single intended operation. We associate two operation machines with each type of intention (read or write), one to be bound to the object copy at *TG*'s level and one to be bound to the object copy at *M*'s level if *M* itself is a transaction group. These machines are bound to their respective transaction groups for the duration of the intention.

Figure 3 shows an example of the intention machines that are put in place when the *CH_Block* transaction group accepts an intention request for the operation <*Arch,w,CH*>. Figure 3(a) shows the machine bound to the *CH* object at *CH_Block*'s level. It prevents any write of the object by any other member, while allowing exactly one write by *Arch*. This machine also allows anyone to read the original version of *CH* until the new version is created, assuming the read operation neither causes a change of state in any existing machine bound to the object nor modifies the object. Figure 3(b) shows the machine specifying the allowable operations by the members of *Arch* that can be batched into the single intended write in *CH_Block*. It allows many read and write operations by its members, but at least one member must do a write operation first. These patterns assume that *Arch* has already read the *CH* object.

4.2.2 Implementation

A member *M* declares its intention to do an operation by sending an *intention request* to its parent transaction group *TG*. This request is either accepted, queued, or refused depending on whether the intended operation would be accepted, queued, or refused. If *TG* accepts the intention, it associates an additional operation machine with its version of the object to block any operations that conflict with the intended one. If *M* is a transaction group, it associates an operation machine with its object version to ensure that the combined effect of all of its member operations is as intended.

Intention requests cascade up the transaction hierarchy until the object copy is found. Occasionally, the requested intention may be strengthened by a transaction group before it is propagated further. This translation is governed by each transaction group's internal protocols.

Members may wish to *change* their intentions if they decide to do something else or *augment* their inten-

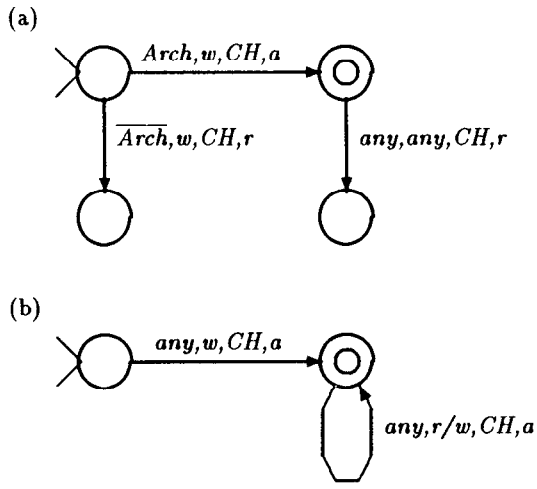

Figure 3: Intention machines: (a) at *CH_Block*'s level; (b) at *Arch*'s level.

tions if they have completed the intended operation and want to do another one. As with intention requests, the database may accept, queue, or refuse these requests. If the change or augment request is refused, the old intention is still retained. Change and augment requests have priority over initial requests.

Intentions may be released at any time, regardless of whether the operation has been done. Once an operation's effects can no longer be revoked by any *member_abort*, its intention is released automatically.

5 Dependencies and Logging

A *log* records the actual sequence of operations on the object copies in each transaction group in the hierarchy. Its main purpose is to keep the information needed for recovery. A mechanism for recovery based on these logs is discussed in [NFSZ90].

5.1 Logs

A transaction group's log records information about the following operations, ordered chronologically by execution time:

- All operations by each member of the transaction group on its object versions (*member entries*).
- All operations by the transaction group on its parent's object versions (*group entries*).

The log is created as the transaction group and its members execute their operations. If a member M of TG is a transaction group, each operation by M is recorded in both logs. In TG's log there is a member entry for the operation, and in M's log there is a group entry. A log entry is of the form

$$\mathcal{LE} =< M, O, o >,$$

where

M is the *TID* of some member,
$O \in \{r, w\}$ is an operation, where r is read and w is write,
o is an object identifier,

The logs in Figure 4 show correlations between log entries in the transaction groups for *CH_Block* and *Land* for the transaction hierarchy shown in the introduction (Figure 1). The history for the transaction group

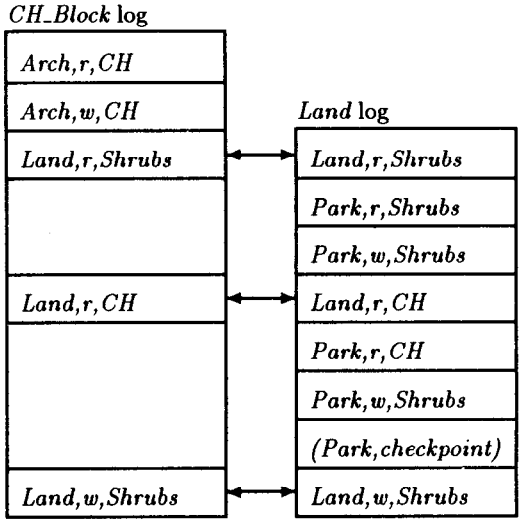

Figure 4: Example logs of a transaction group *(CH_Block)* and its member *(Land)*.

CH_Block, as reflected in its log, is correct according to the operation machines defined for it in Figure 2. The history of the *Land* transaction group, as reflected in its log, is correct according to the machines defined below in Figure 5(a). The double-headed arrows correlate entries in the two logs for the same operation. Note that each initial read of an object by a member of *Land* is preceded by a read by *Land* itself to get a current copy of the object in its local cache. Also, the checkpoint of the *Park* transaction group causes the modified version of the *Shrubs* object to be written up to the *CH_Block* transaction group.

When a transaction group's history as recorded in its log is correct, its internal protocols guarantee that

the effects of the changes done by its members on its object copies are identical to the changes done by its operations on its parent's copies. Otherwise, the transaction group's internal protocols are incorrect.

5.2 Dependencies

We use dependencies among operations to determine what is affected directly and indirectly when an operation aborts, so that we can recover the database to a correct state. These dependencies are recorded in the transaction groups' logs.

For each defined pattern in a transaction group, *pattern dependencies* are formed among operations that participate in that pattern. Since patterns define allowable operation sequences, each operation in the sequence relevant to some pattern depends on the correctness of the previous operations in that sequence. Pattern dependencies are recorded for each operation on the operation directly preceding it in each pattern in which it it participates.

Reads-from dependencies occur because a read of an object version by member M is only correct if the operation that wrote that version is also correct. If the write operation later becomes invalid, then M's read operation read incorrect information, and is also invalid.

Parent-child 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 first. M's read is correct only if TG read a correct version. 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 to its parent, the validity of that write is based on the validity of the last write operation by one of its members.

Figure 5 shows synchronization machines bound to the member *Park* of the *Land* transaction group, and the piece of the log for the *Land* transaction group copied from Figure 4. The dependency chains are shown at the left of the log. The left chain shows the dependencies from the traversal of the cooperative synchronization machine bound to member *Park* and object *Shrubs*. There are no dependencies related to the other machine. The right chains show the parent-child dependencies.

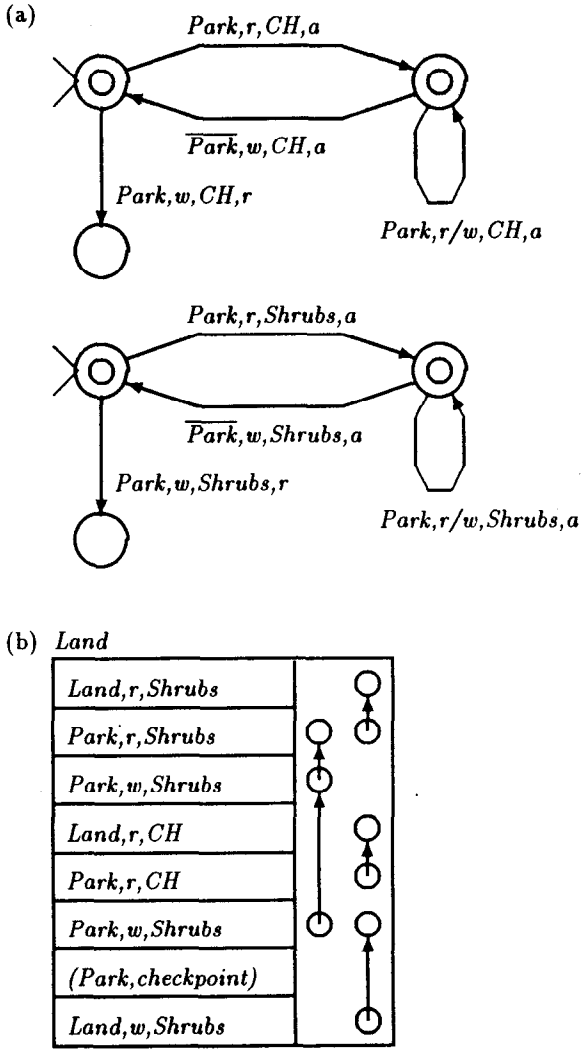

Figure 5: Example dependencies: (a) Synchronization machines, (b) Dependencies.

6 Implementing Multilevel Atomicity

Lynch's *multilevel atomicity* [Lyn83] provides a framework for relaxing atomicity in transactions that naturally decompose into a hierarchy of tasks. Multilevel atomicity allows the specification of allowable interleavings of transactions in an execution. Lynch defines an application database that uses a set of equivalence relations π and a breakpoint specification $\mathcal{B}$ to determine which executions are multilevel atomic.

The equivalence relations π define for each pair of transactions *(t, t')* the set of places t can interrupt t', i.e., how much they trust each other. Figure 6 shows the equivalence relation for the subset of the Utopian Planning example. This equivalence relation corresponds to the transaction hierarchy in Figure 1, with the different equivalence classes in the relation named as transaction groups. Two transactions share an equivalence class at some level if they have the same transaction group as an ancestor at that level in the

$\pi(1)$: {Ann, Bob, Carl, Dave, Fred}
$\pi(2)$: {Ann, Bob, Carl, Dave} {Fred}
$\pi(3)$: {Ann, Bob, Carl} {Dave} {Fred}
$\pi(4)$: {Ann, Bob} {Carl} {Dave} {Fred}
$\pi(5)$: {Ann} {Bob} {Carl} {Dave} {Fred}

Figure 6: Equivalence relation π for the Utopian Planners example.

hierarchy.

The k-level breakpoint specification $\mathcal{B}$ is defined for a system of transactions S as a list of individual breakpoint specifications for each possible execution of S. The individual breakpoint specifications define hierarchically for each transaction t the points at which each other transaction t' can interleave its operations. These points naturally depend on the lowest level at which t and t' share an equivalence class in π.

Each transaction group in a cooperative transaction hierarchy has a set of operation machines that specify the allowable executions for the transaction. Because each level in the transaction hierarchy corresponds to a level in the breakpoint specification, we can modify these operation machines to define and enforce the breakpoint specifications for its members. We do this by inserting conflict arcs into the operation machines to enforce atomicity. For every two consecutive operations that must form an atomic sequence at the current level, insert a refuse arc from the state between the two operations in the the operation machine associated with the corresponding transaction group. The intermediate state must not be a final state. Every two consecutive operations that have a breakpoint between them at this level have no conflict arc between them. These two situations are shown in Figure 7. We can use this technique to define allowable sequences and breakpoints at each level for the set of transactions in the form of operation machines. We can also use batching to simplify the definition of the operation machines while still ensuring that operations which are atomic at one level are atomic at all higher levels in the hierarchy.

Given these machine definitions, we can use the synchronization mechanisms for cooperative transaction hierarchies to enforce that the resulting executions are *correct* multilevel atomic executions.

7 Summary

In this paper, we presented a scheme that allows us to program the correctness criteria for a database to

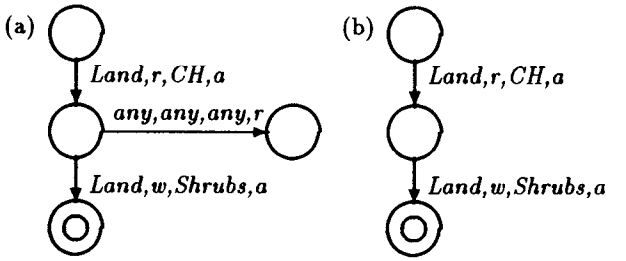

Figure 7: Multilevel atomicity examples: (a) two operations which execute atomically, (b) two operations with a breakpoint between them.

fit specific needs. We have shown how to use cooperative transaction hierarchies to support interactive design transactions. However, we feel that this scheme can also be used to support a wide variety of other cooperative applications.

Cooperative transaction hierarchies allow the decomposition of a task into a hierarchy of subtasks. Internal nodes in the hierarchy, called *transaction groups*, manage the interactions among their children and propagate the effects of their children's operations up the hierarchy in a controlled manner. The leaf nodes are the *cooperative transactions*, and themselves may not be atomic. The transaction hierarchy may be modified as the design task progresses.

Operation machines are used to define task-specific patterns of interaction among the members of a transaction group. The set of correct histories for a transaction group contains exactly those which are allowable by its set of operation machines.

Operations done by members of a transaction group are done on *versions* of the object in the group's local set. A version of an object is copied into a transaction group's set when one of its members first reads the object. A member may protect its ability to do an operation on an object using an *intention*. Intentions ensure that the overall stream of operations on the object remains consistent. Special operation machines, called *intention machines*, are used like locks to enforce the intentions.

In cooperative transaction hierarchies, the effects of operations done at one level may propagate to other parts of the hierarchy before the changes become irrevocable because of an abort or failure. Also, the use of patterns and operation machines causes new types of dependencies among operations. A *log* associated with each transaction group records the execution of its members and the dependencies among the opera-

tions in that execution, for recovery purposes. A member's log is linked to that of its parent because there is an entry for each of its operations in both logs.

8 Acknowledgments

The authors would like to thank Andrea Skarra and Mary Fernandez for many helpful discussions.

References

[CH84] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science*, volume 16, pages 89–102. Springer-Verlag, 1984.

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

[FZ89] Mary Fernandez and Stanley Zdonik. Transaction groups: A model for controlling cooperative transactions. In *3rd International Workshop On Persistent Object Systems*, January 1989.

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

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

[Kai90] Gail E. Kaiser. A flexible transaction model for software engineering. In *Proceedings of the 6th International Conference on Data Engineering*, 1990.

[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 Proceedings*, Singapore, 1984.

[KS90] Henry F. Korth and Gregory D. Speegle. Long-duration transactions in software design projects. In *Proceedings of the 6th International Conference on Data Engineering*, 1990.

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

[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.

[NFSZ90] Marian H. Nodine, Mary F. Fernandez, Andrea H. Skarra, and Stanley B. Zdonik. Cooperative transaction hierarchies. Technical Report CS-90-03, Brown University Computer Science Department, February 1990.

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

[Ska90] Andrea Skarra. Localized correctness specifications for cooperating transactions in an object-oriented database. *Office Knowledge Engineering*, 4, 1990.

[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*, 1986.