VLDB Journal, 1, 41-80 (1992) T.L. Kunii, Editor

©VLDB

Cooperative Transaction Hierarchies: Transaction Support for Design Applications

Marian H. Nodine and Stanley B. Zdonik

Received March 25, 1991; revised version received August 21, 1991; accepted October 22, 1991.

Abstract. Traditional atomic and nested transactions are not always well-suited to cooperative applications, such as design applications. Cooperative applications place requirements on the database that may conflict with the serializability requirement. They require transactions to be long, possibly nested, and able to interact with each other in a structured way. We define a transaction framework, called a *cooperative transaction hierarchy*, that allows us to relax the requirement for atomic, serializable transactions to better support cooperative applications. In cooperative transaction hierarchies, we allow the correctness specification for groups of designers to be tailored to the needs of the application. We use *patterns* and *conflicts* to specify the constraints imposed on a group's history for it to be correct. We also provide some primitives to smooth the operation of the members. We characterize deadlocks in a cooperative transaction hierarchy, and provide mechanisms for deadlock detection and resolution. We examine issues associated with failure and recovery.

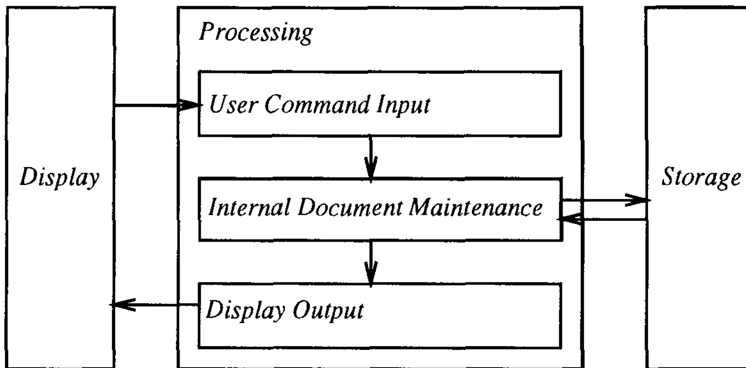
Key Words. Cooperation, design transactions, transaction hierarchies, non-serializability, transaction synchronization, deadlock detection, version management.

1. Introduction

Design applications such as CAD tools generate 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. When using a database, design applications tend to generate long transactions that are not necessarily atomic. Because the design

Marian H. Nodine is a Ph.D. Candidate, and Stanley B. Zdonik, Ph.D. is Professor, Computer Science Department, Brown University, Providence, RI. (Reprint requests to Dr. S. Zdonik, Computer Science Dept., Box 1910, Brown University, Providence, RI 02912, USA.)

Figure 1. Word processing system design



process is interactive, design transactions also interact with each other, specifically through the sharing of data. Because the design process is both interactive and iterative, design transactions are not completely specified at the time they begin. Rather, they are open-ended. Because the design task often decomposes hierarchically, the transactions that support that task also may be structured to reflect that decomposition. All of these properties complicate the issues of synchronizing these transactions and recovering from failure. On the other hand, we can assume that the transactions are controlled ultimately by experienced users who can respond intelligently to problems.

Design applications are an example of *cooperative applications*. 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. Instead, we provide a way to program application-specific correctness specifications for the cooperative transactions.

In this paper, we will use as an example a CASE tool being used to write a simple word processing system. The word processing system has three modules, as shown in Figure 1.

The *Display* module manages the user interface, the *Processing* module maintains the document in its internal format, changing it according to the commands received from the *Display* module, and the *Storage* module deals with the disk I/O. The *Processing* module has submodules that take the commands specified to the display module and compute the appropriate changes in the document itself. Each module and submodule has designers responsible for designing and programming it. The objects in

the database include design specifications for each module, the interfaces between modules, the internal format of the document, and header and source files containing the code.

In this example, the specification for the interface between the *Display* and *Processing* modules may be manipulated by more than one designer. The *Display* module designer may change it as he gives the users new capabilities. For example, if he allows the user of the word processor to specify new commands, new procedures will be added to the interface specification. The *Processing* module designers, specifically the one working on the *User Command Input* submodule, may need to change the specification to define the information that they need to execute the new command.

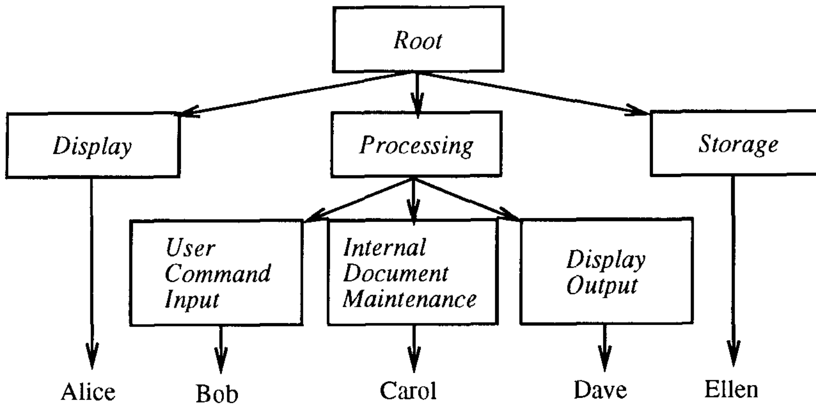
In the word processor design, each designer completes only part of the design task and maintains only partial consistency in the database. For instance, the part of the interface discussed above that is in the *Display* module is maintained by the designer of that module, and similarly for the part of the interface in the *User Command Input* submodule. However, the two parts of the interface design are interdependent, because the various designers corporately implement the interface, and therefore must agree on any changes made to it. The interactions between the designers are structured according to the task they are working on; using the example in the previous paragraph, if the *Display* module designer changes the interface, the *Processing* module designers must ensure that the changes are reasonable from their perspective. The designers also may iterate through several refinements or changes in the design as they work. Because of the complexity of the design process, the individual designers dynamically determine what is necessary to complete their part of the task. Also, all designers involved in a task must agree that it is complete before it should be committed.

From the point of view of the database, each designer starts a *cooperative transaction* for each major design change he participates in. Groups of designers working together join *transaction groups*. In the above example, each of the *Display*, *Processing*, and *Storage* modules is represented by a transaction group. Three transaction groups are nested within the *Processing* group. Figure 2 shows this structure.

Each designer has started up a *cooperative transaction*. For example, Bob works on the *User Command Input* submodule of the *Processing* module. We call the whole tree-like structure a *cooperative transaction hierarchy*.

The cooperative transaction hierarchy for a particular design task is typically structured according to the task's natural decomposition. The internal nodes in the tree represent transaction groups, and the external nodes represent cooperative transactions. For any given transaction group, we call its direct children its *members*. The transaction group itself is its members' *parent*. The *Root* transaction group is at the top of the transaction hierarchy.

Figure 2. Word processing hierarchy



Cooperative transactions are sequences of related atomic read and write operations. Because cooperative transactions in the same transaction group may need to read and modify the same set of objects, they may read or overwrite each others' uncommitted object versions (as in the previous interface example). Consequently, concurrent cooperative transactions may be interdependent, and atomicity may be too strict a requirement for correct execution. Transaction groups correspond to some task, and that task is done cooperatively by its members. Since transaction groups may themselves cooperate, and thus may also be interdependent, atomicity may be too strict a requirement for them as well.

In this paper we define a method for specifying correctness for a cooperative transaction hierarchy. This method, which is related to one originally proposed (Skarra, 1991), allows each transaction group to tailor the correctness specification for its members to the needs of the group's task. Correctness is specified not only by how the different members cannot interact, as in a locking system, but also in how they must interact to complete the task. Thus, we can both support interactions between cooperative transactions that share uncommitted data, and control the extent to which this data sharing is allowed.

Each transaction group has its own local set of copies of the objects that its members are currently accessing. An object is copied into this set the first time a member reads it. An object is removed from the set when all members that have interacted with it have terminated. Each group makes its own guarantees about the *persistence* of the object copies in its set, i.e., their resilience to certain types of failure. For example, a transaction group may have its own backing store where it keeps its object copies in case of system failure.

Each transaction group also has its own correctness specification that defines the order in which its members' operations can interleave. The transaction group is responsible for ensuring that its members' operations follow that order. This includes determining the acceptability of the operations as they are submitted to the transaction group (synchronization), and ensuring that the correctness of the transaction group is upheld even after the failure or abort of a member.

A transaction group *history* is the actual sequence of operations by its members on the objects in the database. In this paper, we consider only histories that contain nonnested *read* and *write* operations. Each transaction group's correctness specification defines 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. Each *pattern* defines a set of acceptable orderings of operations in a transaction group's history. For example, a pattern might say, "Alice must read the *Display/Processing Interface Specification* object (*if_spec*) before she writes it." *Conflicts* are defined within the context of patterns, and specify orderings of operations that must not occur. For example, "Once Alice has written *if_spec*, Bob cannot write it until he reads Alice's version." Every transaction group has many patterns and conflicts defined for the objects and members, and they work together to define correctness.

Although we use only read-write semantics in this paper, we expect that this model can support any set of operations that do not nest. This would include methods in simple object-oriented systems.

Once we have defined a notion of correct operation interleaving in a transaction group's history, we need to ensure that the actual history produced by a transaction group corresponds to that notion of correctness. The synchronization process examines each operation submitted to the transaction group in turn, and queues or rejects any operation that cannot occur at the time it was submitted. Unfortunately, once we allow for operation queueing, deadlocks can develop among operations submitted by different members. Thus, the transaction groups are also responsible for deadlock detection and resolution within the cooperative transaction hierarchy. Deadlock detection and resolution here have some similarities to that of a lock-based database. However, we may not want to abort an entire member because one of its operations is deadlocked, if only because the member may be long-lived.

In fact, there are many compelling reasons why a deadlock, failure, or abort should not cause the effects of a whole transaction to be backed out of the database. In addition to the fact that cooperative transactions may be long-lived and open-ended, we also know that they may have complex interdependencies that would cause a severe cascading abort problem (Haerder, 1987). These interdependencies need to be dealt

with during recovery; however, we may want the cascading only to propagate to the *parts* of the other cooperative transactions that actually were affected by the failed or aborted operation.

Transaction groups also provide a few facilities that, while not required for correct operation, do facilitate the process of cooperation. *Intentions* allow a member to reserve the capability to do some specific operation in the future. Once a member is granted an intention to do an operation, any conflicting operation is queued until the intention is released.

Intentions allow members to make either optimistic or pessimistic assumptions about whether other members will interfere with their work. For example, say Alice needs to make a large modification to *if_spec*. The result of this modification will be the writing of the new version of the object at some possibly distant point in the future. If she wants to guarantee that she will be able to do the write, she can first request an intention for it. The granting of the intention by the transaction group guarantees that the write operation will not be queued or refused. On the other hand, if she is optimistic that the operation will be accepted, she can go ahead and do the modification without requesting the intention. However, if the write operation is rejected or queued, she may have to redo her work.

Notification allows a member to be informed of other members' operations that it may be interested in. Notification facilitates the working of the transaction groups because it allows the members to know when they may need to do something in response to a change in some object in the database. For instance, the designers responsible for the *Processing* module in our example may request to be notified when *if_spec* changes.

This paper covers the basic operation of a cooperative transaction hierarchy. In the next section, we define cooperative transaction hierarchies in more detail. In Section 3 we discuss patterns and their associated conflicts, and describe how they are specified *(operation machines)*. We then discuss the basic synchronization algorithm, including how we coordinate the synchronization among the different groups in a cooperative transaction hierarchy to make the database operate more smoothly. In Section 4 we discuss the queueing semantics and the mechanisms for deadlock detection and resolution. In Sections 5 and 6 we discuss what information needs to be kept for recovery, and provide a procedure for backing changes out of a database after a failure or abort. We also provide guidelines for how operations can be redone cooperatively. Finally, in Section 7, we discuss database support for mediated communication among the cooperative transactions. A brief summary of related research is provided in Section 8.

2. The Model

A *cooperative transaction hierarchy* is a structured set of cooperative transactions, where the structure of the interaction among the cooperative transactions reflects the underlying hierarchical decomposition of the task they are working on together. The internal nodes are the *transaction groups*, and the leaves are *cooperative transactions*. 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. The *Root* transaction group always exists at the top of the hierarchy.

2.1 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. The internal protocols include a *correctness specification* that defines the patterns and associated conflicts that must hold in the transaction group's history. 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 are defined to be 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. We do not follow a general versioning scheme; in this scheme a member's version of an object always directly subsumes its parent's version. The object versions in a particular transaction group's set may be accessed (read or written) only by members below it in the hierarchy. The *Root* transaction group has a version of every object in its set, and this version may be accessed by all members.

Each transaction group makes its own guarantees about how resilient its copies are to various failures (*permanence*). Because the *Root* transaction group is at the top of the hierarchy, it contains the most stable versions of the objects. All of the other versions of an object in the hierarchy are either identical to *Root*'s version, or are more recent intermediate versions. Because of this, the *Root* transaction group must make the strongest guarantees about the persistence of its copies. This means that these copies should be rigorously backed up on disk, and routinely archived on tape or some other off-line storage medium.

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 correctness specification. Because of the long-lived and interactive nature of the cooperative transactions, we also allow the individual members to selectively commit or undo parts of their work as they progress.

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 conforms to its correctness specification.

When a member (M) of a transaction group (TG) commits some portion of its work, the effects of that work on the database must be propagated to its parent (P). This involves encapsulating the operations done by M into a sequence of operations initiated by TG on P's object copies. This sequence of operations must be compatible with P's correctness specification, because each transaction group must adhere to its own notion of correctness. It also must be equivalent to the committed operations, in that it leaves P's object copies in the same state as the copies in TG. For example, a write operation followed by a sequence of read and write operations on a single object in *TG* may be encapsulated into a single write operation by *TG* on P's object version. This encapsulation hides the internal operations of the transaction group's members from its parents, allowing groups of non-serializable transactions to be members of other serializable groups, among other things.

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 cooperative transactions may join and leave the hierarchy as the overall task progresses.

2.2 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 know 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 correctness specification. The transaction group's synchronization mechanism examines each operation as it is submitted, and returns *a(ccept)* if the operation can be executed immediately, *r(efuse)* if the operation can-

not be executed, or *q(ueue)* if the operation may be doable later. If *q* is returned, the operation itself is not actually queued. Instead, the intention to do the operation is queued, and the transaction is notified once the operation can definitely either be accepted or refused. In the meantime, the cooperative transaction is allowed to dequeue this intention and/or continue processing. As with transaction groups, operations are individually checkpointed or aborted by the cooperative transaction as the design task progresses.

2.3 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 of the different requirements cooperative applications place on the database.

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 an enumeration of its members or a procedure for authenticating new members and a correctness specification. 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 rules for managing object versions, and the rules for requesting, sending, and receiving notifications.

Once a member has been established, it may operate on its parent's object copies in any way allowable by its parent's correctness specification. Objects are copied into its parent's object set as they are needed by the member. Each operation is checked by the transaction group as it is submitted to see that it conforms to the group's correctness specification. The only operations that are accepted are those that are correctly ordered.

When the members of a transaction group complete some set of changes, the operations that effected that change can be checkpointed (committed) using a *member_checkpoint* operation. The decision to checkpoint may be made either manually or automatically. Because the group's patterns capture the structure of all interactions among its members, all patterns a member participates in must be complete when the checkpoint takes effect.

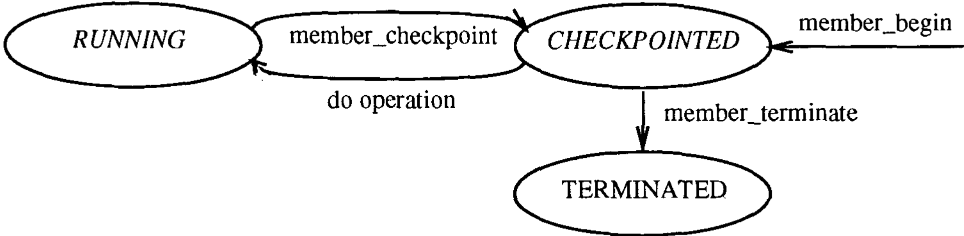
The checkpoint operation causes the internal operations by the members to be encapsulated into an equivalent set of external operations by the transaction group. Since we restrict the operation set in this paper to {*read, write*}, this means that new

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, specifying its latest version as the one to be written. The requests are issued in the order in which the last modification of each object occurred. They must be acceptable according to the parent's correctness criteria. Each write introduces a new object version 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 if the member fails. Since the different groups in the transaction hierarchy may guarantee different levels of permanence, the *member_checkpoint* procedure only guarantees that the new object versions are as permanent as the parent transaction group guarantees them to be.

As an example of how a member operates, consider what happens when Alice, who is a member of the *Display* transaction group in Figure 2, decides to make her changes to the *if_spec* object. When she first tries to read the specification, a copy of the object is read into *Display*'s object set from the *Root* transaction group. Alice's *read* operation is then checked for correctness. Provided it is correct, Alice can then read the object into her editor. As Alice edits the object over time, she issues a series of *write* operations. Provided the *Display* transaction group accepts these operations as correct, new versions of *if_spec* are introduced in *Display*'s object set. When she has completed the edit, she checkpoints. At this time, the *Display* transaction group issues a write operation to the *Root* transaction group to propagate the new version up the hierarchy. The write operation must be accepted by the *Root* transaction group for the checkpoint to succeed.

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. An operation 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. The process of aborting must also leave the transaction group's history in a correct state. This means that any operations that are incorrect as a result of the abort must also be removed from the history, as well as any versions they created.

Figure 3. State transition diagram for cooperative transactions and transaction groups.



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 terminates, its transaction group becomes responsible for recovery if any of its 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* – the member may have some outstanding uncheckpointed operations.
- *Checkpointed* – all existing operations by the member are correct and final, according to both itself and the transaction group's correctness criteria.
- *Terminated* – the member has explicitly terminated.

The state transition diagram is shown in Figure 3.

2.4 Data Structures. 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 correctness specification. 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 a 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 correctness specification into single operations by itself which are correct according to its parent's correctness specification, as described in the previous section.

A transaction group is a tuple $\mathcal{TG} =< TID, \mathcal{P}, \mathcal{S}, \mathcal{M}, IP >$, where

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.

M contains the member IDs of TG's members, and

IP specifies TG's internal protocols.

A cooperative transaction is not necessarily atomic. The sequence of operations for a single transaction does not have to be individually correct and consistent, in that it may not in itself leave the database in a consistent state. However, it must be validated according to its parent transaction group's internal protocols.

A cooperative transaction is a tuple $\mathcal{CT} =< TID, P, S >$, where TID, P, and S are defined as for transaction groups.

2.5 Operations. Members access and update their local object versions using *operations*. An operation defines an atomic action on a single object by a single member of a transaction group. When an operation is checkpointed, its effects are propagated towards the root transaction group. An operation can also be aborted.

An operation is a tuple $\mathcal{O} =< M, o, O >$, where

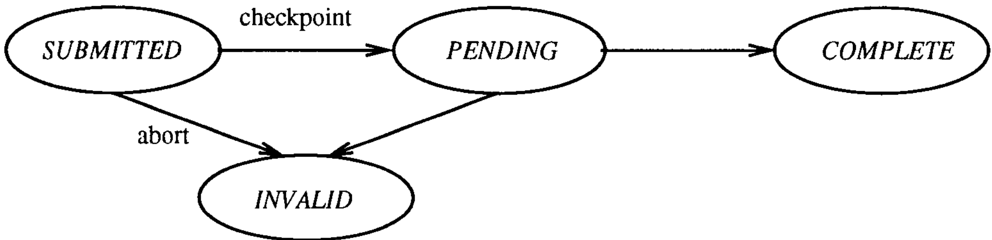
$M \in \{any, m_i, \overline{m_i}\}$ is the ID of some member, where *any* is any member, m_i identifies member i, and $\overline{m_i}$ is any member except m_i.

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

O is an object identifier.

When a member needs to access or update an object, it submits the operation to its parent transaction group. Before an operation can be executed, it must be accepted by the transaction group to which the member that requested the operation belongs. Once the operation has been accepted, we know that its execution at this time does

Figure 4. State transition diagram for operations.



not conflict. Therefore, we can read or write the object as requested. 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, and the process of making any changes resulting from the operation more permanent begins. However, there is a gap between the time the member checkpoints the operation (indicating that it will not abort it later) and the time the operation cannot be undone for any reason (e.g., if some other operation aborts). Once the operation cannot be undone for any reason, it is *complete.*

At any time, and operation is in one of the following states:

- *Submitted.* The operation has been accepted by the transaction group's internal protocols, and executed. Its effects are temporary.
- *Pending.* The operation has been checkpointed by the member that requested it. It cannot be aborted by that member, though it may be indirectly invalidated if some other operation is aborted.
- *Invalid.* The operation has been aborted. Its effects are not reflected in the database.
- *Complete.* This operation has been checkpointed, and all of the earlier operations in the history are *Complete.* It cannot be undone.

The possible state transitions are shown in Figure 4.

2.6 Histories. The history of a transaction group is a partially-ordered sequence of operations. It contains both the operations submitted to the transaction group by its members, and the operations submitted by the transaction group to its parent. In

other words, each operation in the cooperative transaction hierarchy potentially participates in two histories – it always participates in the history of the transaction group the operation is submitted to, but it also participates in the history of the member that submitted it if that member is also a transaction group.

3. Synchronization

Each transaction group in a cooperative transaction hierarchy defines and enforces its own correctness specification using patterns and conflicts. The patterns and conflicts defined in the *Root* transaction group specify the correctness constraints on the database. The patterns and conflicts defined in other transaction groups specify their own correctness constraints. A transaction group's synchronization process ensures that its members generate a history that conforms to group's correctness specification.

3.1 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 constrain the members' operations to a specific execution; rather they 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, 1991). An operation machine is a finite-state automaton. Each transition in an operation machine is labeled with the symbol $\sigma =< M, o, O, P >$ that defines the operation associated with it, where

$M \in \{any, m_i, \overline{m_i}\}$ is the *ID* of a member or set of members any of whom can initiate the operation, where *any* is any member, m_i identifies member i, and $\overline{m_i}$ identifies 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, and

$P \in \{a, r, q\}$ 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 *a(ccept)* if the operation conforms to the pattern, *r(efuse)* if the operation conflicts, or *q(ueue)* 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.

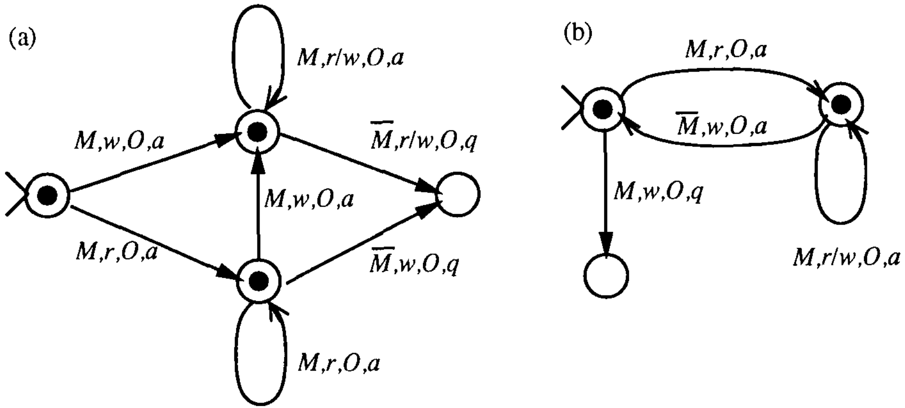
The *active set* of a transaction group is the set of operation machines currently being used to define and enforce correctness in the group. The operation machines in a group's active set may be defined directly or instantiated from *operation machine templates*. An operation machine template is defined in the same way as an operation machine, but the transition functions may have variables for the member and object identifiers. A machine template is instantiated by making a copy of its definition and binding the variables in the copy to specific object and member IDs.

3.2 Synchronization Algorithm. In a given transaction group, the correctness criteria are specified by an active set of operation machines, each of which describes a specific pattern and its associated conflicts. When an operation is submitted by some member to the transaction group, the group first finds the set of operation machines in its active set that are relevant to that operation. A machine is relevant to the operation if there is some arc from the machine's current state that specifies explicitly whether the operation should be accepted, queued, or refused. An operation is only accepted by the group if it is accepted by every operation machine in its active set that considers that operation to be relevant. An operation is refused if any relevant machine specifies the operation should be refused. Otherwise, if any relevant machine specifies that the operation should be queued, then it is queued. Executing an operation causes an arc traversal in each machine in the active set that considers that operation to be relevant. Since the operation machines together specify the correct operation of the members of the group, a database maintains consistency if every member checkpoints only when every machine that considered one or more of its operations relevant is in a final state.

3.3 Operation Machine Functions. Operation machines can be used for various functions in a cooperative database. Example functions include synchronizing the access of an object by multiple members, protecting an object from being accessed by specific members, and enforcing a pattern among a specific set of operations by multiple members. Some of the operation machines are defined manually, either when the transaction group is set up or when new members are added. Some may be instantiated automatically as needed from operation machine templates.

Pattern machines and *protection machines* are examples of machines that are defined manually for specific functions. *Pattern machines* define the sequences of operations needed to do some current task. Consider the machine defined in Figure 7(a).

Figure 5. Synchronization machine templates



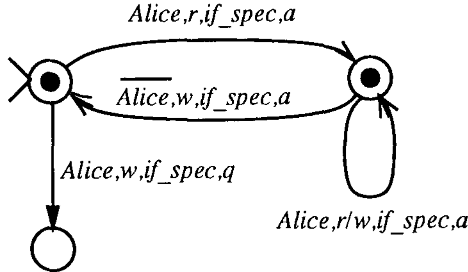
(a) serializable; (b) cooperative.

When operation machines are shown diagrammatically, the start state is indicated by a caret (>), final states have bullets (•) in them, and transitions are represented by labeled arrows (→). This pattern forces *Display* to read the interface specification *(if_spec)* after the last write before it can actually make the corresponding changes in the module *(display.c)*. It also forces *display.c* to be current with *if_spec* before the task is complete. These machines put a general form on the allowable executions consistent with the transaction group's task. They do not necessarily restrict the transactions to conform to one specific execution, though they may. *Protection machines* are used to prohibit specific members from doing specific operations on an object. They are implemented using refuse arcs.

Synchronization machines define the patterns that enforce the underlying concurrency control mechanism (e.g., cooperative, serializable). Since a transaction group usually wants to enforce concurrent access of objects in a uniform way, each transaction group instantiates the synchronization machines as needed from its *synchronization machine template*. The template looks like an operation machine, except that the arcs are labeled with the variables M for a single member identifier and O for a single object. When a member first accesses an object, the transaction group instantiates the template by copying it, binding all instances of the variable M to the member's ID and all instances of the variable O to the object's ID, and placing the new machine in its active set. When a member terminates, its synchronization machines are removed from the active set.

Figure 5 gives two examples of synchronization machine templates. Figure 5(a) shows a machine that enforces serializability in a similar manner to two-phase locking. If the transaction is in the lower middle state the object is basically read-locked.

Figure 6. Synchronization machine governing Alice's interaction with the *if_spec* object.



Similarly, if the transaction is in the upper middle state, it is basically write-locked. The locks persist until the member terminates. At that point, the operation machine is removed. Figure 5(b) shows a synchronization machine template to enforce one type 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. This machine prevents a member from overwriting another member's changes. As an example, say the *Display* transaction group enforces the cooperative synchronization protocol defined by the template in Figure 5(b). When Alice first reads the *(if_spec)* object, the synchronization machine shown in Figure 6 is instantiated and placed in *Display*'s active set of operation machines.

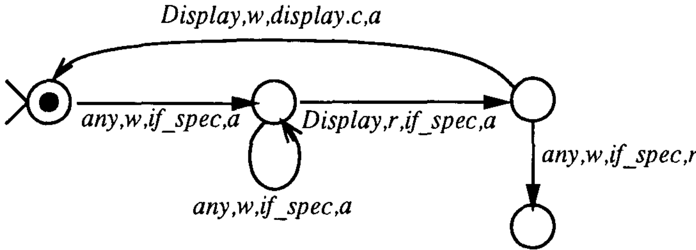
A *traversal* of an operation machine is a sequence of operations associated with consecutive traversals of accept arcs for the machine, beginning at the start state and ending at the current state. If we view the operation machine as a directed graph, a traversal is a path in the graph that begins at the start state, ends at the current state, and traverses only arcs whose return value labels are *a*.

Figure 7 shows examples of traversals. Given the operation machine shown in Figure 7a, the operation sequence 1 in Figure 7b is a traversal. Sequence 2 is not a traversal because the third operation causes the traversal of an arc that returns refuse. Sequence 3 is not a traversal because the first operation does not cause an arc traversal at all, though it would be accepted by the machine. A *complete traversal* ends at a final state, e.g. sequence 4. The null traversal of this machine is correct, though it is only complete because the start state is also a final state.

We can use operation machine traversals to formalize our definition of correctness. For each operation in a history, we know which arcs were traversed when it was executed. Define the sequence Π_{OM} associated with operation machine *OM* as

Figure 7. Operation machines and traversals

(a)



(b)

1. {*<Processing,w,if_spec>*, *<Processing,w,if_spec>*, *<Display,r,if_spec>*}
2. {*<Processing,w,if_spec>*, *<Display,r,if_spec>*, *<Display,w,if_spec>*}
3. {*<Processing,r,if_spec>*, *<Processing,w,if_spec>*, *<Display,r,if_spec>*}
4. {*<Processing,w,if_spec>*, *<Display,r,if_spec>*, *<Display,w,display.c>*}

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

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 it satisfies both the *pattern* and *conflict criteria.*

- *Pattern criterion*
  All sequences Π_{OM} for the machines that were active during the history are traversals.

- *Conflict criterion*
  The history contains no operations that would conflict based on the state of any operation machine active at the point in the history where the operation is recorded.

Correctness is an ongoing notion; it applies to histories of transaction groups that may not have completed their task yet. However, correctness in itself does not guarantee that the work thus far preserves global consistency. A history is *complete* when all its

sequences are complete traversals. A complete history both is correct and preserves global consistency.

3.5 Intentions. Each transaction group in a hierarchy has a local set of versions of the objects that it is currently accessing. Since multiple copies of the same object exist, different members in the hierarchy may be doing operations on different copies of the object that will ultimately conflict. The conflict will be detected eventually, but resolving it requires that some operations be aborted. If a member wishes to ensure that an operation will eventually succeed, it may declare an *intention* to do the operation before any work is actually done. Intentions reserve the capability for a member to do a single operation on a single object. When a member *M* of a transaction group *TG* is granted an intention, *TG*'s version of the object is restricted in such a way that no other operations can be done that will cause the intended operation to be rejected. This also means that new operation machines cannot be added to the transaction group's correctness specification if they would cause the intended operation to be rejected or queued. This could occur, for instance, if the database administrator decided to add a protection machine to the group's active set that rejects the operation.

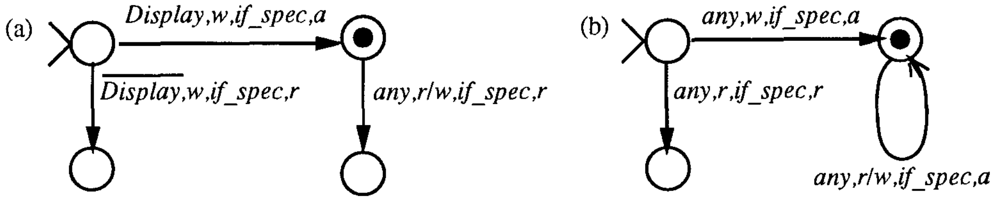
When the member *M* is a transaction group, the intended operation represents the combined effects of a sequence of operations by *M*'s members on the object. For example, many reads and writes by the members of *M* can be consolidated into a single write by *M* to *TG*. Thus, there may be more complex 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 are like locks in that they take a pessimistic approach to concurrency control. However, 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. This is similar to taking an optimistic approach to concurrency control. The operation is still

Figure 8. Intention machines



done, provided that it is currently acceptable according to the patterns and conflicts defined in its parent. It also may be queued or refused.

3.5.1 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 in the active sets of their respective transaction groups for the duration of the intention.

Figure 8 shows an example of the intention machines that are put in place when the *Root* transaction group accepts an intention request for the operation *<Display, w, if_spec>*. Figure 8(a) shows the machine bound to the *if_spec* object at *Root*'s level. It prevents any write of the object by any other member, while allowing exactly one write by *Display*. This machine also allows anyone to read the original version of *if_spec* until the new version is created. If the read operation causes a change of state in any existing machine bound to the object, the intention machine should not allow the read. Figure 8(b) shows the machine specifying the allowable operations by the members of *Display* that can be batched into the single intended write in *Root*. 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 *Display* has already read the *if_spec* object.

3.5.2 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. For example, this machine has the same structure as the one in Figure 8(a) if the intention is for a write operation. 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. For a write operation, this machine would be similar to the one in Figure 8(b). These machines can be constructed automatically from standard read and write intention machine templates.

Intention requests cascade up the transaction hierarchy until a copy of the object is found.

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

4. Conflict, Queueing, and Deadlock

Operation conflict occurs when a member cannot do a requested operation immediately because one of the machines associated with the member returns *queue* or *refuse*. Normally, this situation arises because some other member has done some operation that cannot be followed immediately by the requested operation, or because some member has declared an intention that prevents the requested operation from taking place immediately.

Conflict may also occur when a member is requesting an intention. If the intention cannot be granted, it may also be queued or refused.

Queueing an operation or intention does not block the requesting member. Rather, a message indicating that it was queued is returned to the member, and the member is allowed to continue normally. The member may also cancel the request.

Refusing an operation means the operation cannot be done. The transaction group keeps no record of what operations and intentions have been refused.

4.1 The Semantics of Queueing. Each transaction group has a queue of requests waiting to be accepted or refused ordered by the time of the request. That is, the dequeuing of earlier requests is always attempted before the dequeuing of later requests.

This set does not mimic a FIFO queue, however; two requests may be dequeued in a different order than the one in which they were inserted. This could occur, for instance, if the two requested operations were relevant to different patterns.

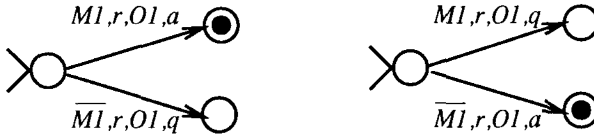
When an intention request is queued, a copy of the request is placed at the tail of the queue and an *intention_queued* message is sent to the requester. The requester may continue processing. At any time, it can cancel its request by sending a *cancel_request* to its transaction group. If the request is not canceled, the transaction group eventually responds by either accepting or refusing the intention request. If the intention request is accepted, then the requesting member can either resubmit the operation or release the intention.

Operation requests are not queued directly, because an operation request contains an entire invocation of the operation. Write operations, for instance, not only specify the member and object to be written, but also the new version. Consider the following scenario derived from the pattern in Figure 6. Recall that the purpose of this pattern is to ensure that Alice cannot overwrite some other member's changes. Assume we are in the start state. Alice submits the operation request *<Alice,w,if_spec>*, which is queued. If it were queued as an operation request, the whole invocation of the operation, including the new version to be written, would be kept. Later, Alice issues an operation request *<Alice,r,if_spec>*, which is accepted. Now the queued request *<Alice,w,if_spec>* could be accepted and the specified version written. This violates the spirit of the pattern, because the new version of *if_spec* was created before the read operation, and therefore cannot depend on the read of the latest version. For the pattern to allow such operations, the arc from the start state labeled *<Alice,w,if_spec,q>* would have been omitted.

We solve this problem by queueing operation requests as intentions. That is, when an operation request returns *queue*, an *intention* for the operation is queued and an *intention_queued* message is returned to the requester. Later, when the request is accepted, the member that requested the operation is told that it has an intention for the operation. The member can then either issue a new operation request or release the intention. The new operation request is generated using information that is currently available, and thus does not inadvertently violate the intent of the patterns.

This solution for queueing intentions only is similar to the approach taken by locking schemes. In a system that uses locking for concurrency control, an appropriate lock on an object must be requested and granted before an operation can take place on it. Lock requests may be queued, but operations are never queued because they are submitted only when the object is already locked for the operation.

Figure 9. Deadlock example.



4.2 Queueing and Intention Deadlocks in a Transaction Group. As with locking systems, a transaction group may have deadlocks. Since we allow members to continue while waiting for queued intentions, the risk in our system is that an intention will never be resolved (accepted or rejected) because it is waiting for other operations that will also never be resolved. We call this situation an *intention deadlock.*

The potential for intention deadlocks is inherent in the structure of the operation machines. Consider the example in Figure 9. If both operation machines are in the start state, then the only operations that allow either machine to reach a final state are <*M1,r,O1*> for the first machine and < *$\overline{M1}$,r,O1*> for the second machine. Unfortunately, <*M1,r,O1*> is queued by the second machine, and < *$\overline{M1}$,r,O1*> is queued by the first machine. Therefore, if both these machines are a part of some group's correctness specification, we know there will be a deadlock once both <*M1,r,O1*> and < *$\overline{M1}$,r,O1*> have been submitted and queued.

In this section, we specify how to construct a waits-for graph within a transaction group, how to detect intention deadlocks given the waits-for graph, and how to resolve them. Further details on conflict and intention deadlocks, including algorithms, theorems and proofs, can be found in (Nodine, 1991).

4.2.1 The Waits-For Graph. Each transaction group is responsible for maintaining its own *waits-for graph* that characterizes which intentions for operations are waiting for which other operations at any given time. A waits-for relationship occurs between two operations op_1 and op_2 when an intention to do op_1 is queued because some machine *OM* has a transition for that operation that currently returns *queue*, and *OM* would accept op_2. In the example in Figure 9, if *M2* (which is in $\overline{M1}$) submits an intention to do the operation <*M2,r,O1*>, the waits-for relationship <*M2,r,O1*>→<*M1,r,O1*> ("<*M2,r,O1*> waits for <*M1,r,O1*>") occurs. Similarly, if *M1* submits an intention to do the operation <*M1,r,O1*>, the waits-for relationship <*M1,r,O1*>→<*$\overline{M1}$,r,O1*> occurs.

At any given time, the waits-for graph contains a node for each operation or set of operations that is involved in some waits-for relationship, and an arc between any pair of nodes representing operations involved in a specific waits-for relationship, indicating its direction. Since waits-for relationships are associated with individual machines, each arc is labeled with a list *waits_for_machines* containing the machines causing that waits-for relationship. That is, if op_1 is waiting for op_2 because of machine OM_1, there is an arc in the waits-for graph from the node representing op_1 (the head) to the node representing op_2 (the tail), and *waits_for_machines* for that arc is $\{OM_1\}$. If an additional intention is later requested that causes the same waits-for relationship because of some machine OM_3, the *waits_for_machines* list on that arc would become $\{OM_1, OM_3\}$.

Different nodes in the waits-for graph can represent different things. Nodes that are tails of any waits-for relationship represent queued operations. The corresponding heads represent anything that could allow the queued operation to be resolved, including unsubmitted operations, queued intentions in the transaction group, operations that have intentions granted for them, or queued intentions in the transaction group's parent.

Waits-for relationships are not necessarily between fully-specified operations. Operations at the tail of a waits-for relationship are those for which specific intentions have been requested, and are therefore fully-specified. However, these operations may be queued by a machine when an operation is waiting for some other operation that could be done by any one of a set of members. In particular, these may be generated by machines with arc labels like $<any, w, O, q>$ or $<\overline{M}, r, O, q>$. Thus, the node at the head of a waits-for relationship may have a less-specified label, with its "member" being some set members.

4.2.2 Updating the Waits-For Graph. The waits-for graph for a transaction group can be maintained incrementally. The actions that change the waits-for graph include executing or aborting operations, changing the set of outstanding intentions, and changing the correctness specification of the transaction group.

When an intention is queued, the new waits-for relationships are added to the waits-for graph for the transaction group. Similarly, when an intention is accepted or refused, waits-for relationships are removed from the graph. When an intention is released or an operation is executed outside of the scope of an intention, any node that represents that operation is removed from the waits-for graph, along with any associated waits-for relationships.

Modifying the correctness specification for a transaction group means either adding new operation machines to the transaction group or removing old ones. When

a new machine is added, we need to see if the machine is relevant to any of the queued intentions. For example, if the new machine would queue the intended operation, then new waits-for relationships need to be added to the waits-for graph. If the new machine would reject the intended operation, then the intention is refused. When a machine is removed, any waits-for relationships associated with the machine are removed from the waits-for graph.

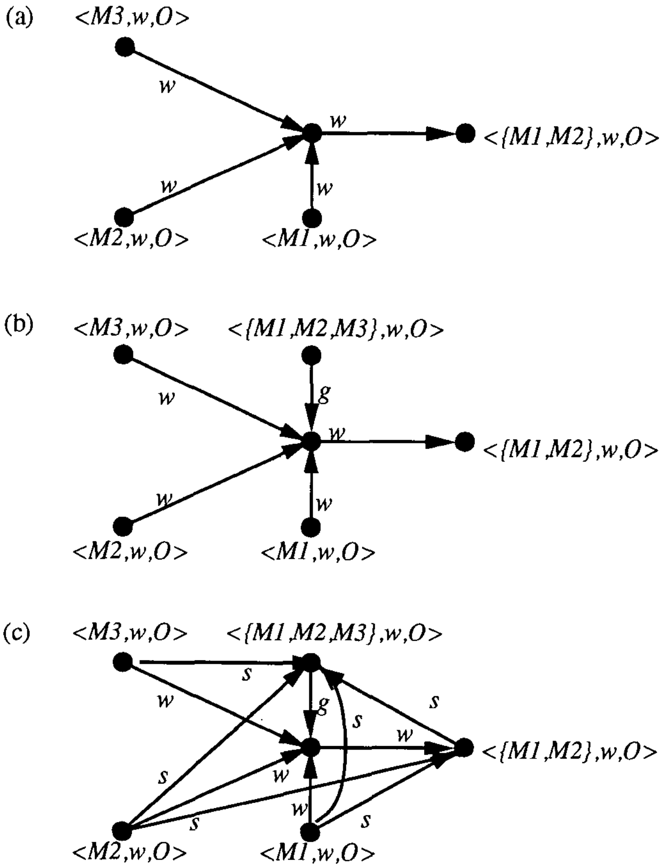
4.2.3 The Structure of Intention Deadlocks in the Waits-For Graph. Intention deadlocks have one of the following forms:

1. $< M_i, o, O_i > \rightarrow \ldots \rightarrow < M_i, o, O_i >$ where M_i is an individual member ID.
2. $< S_i, o, O_i > \rightarrow \ldots \rightarrow < M_i, o, O_i >$ where S_i is a set of member IDs and $M_i \in S_i$.
3. $< S_i, o, O_i > \rightarrow \ldots \rightarrow < S_j, o, O_i >$ where S_i and S_j are sets of member IDs and $S_j \subseteq S_i$.

The first situation is detectable as a cycle in the waits-for graph. The second and third situations are harder to detect for two reasons. First, no waits-for relationship of the form $<S_i, o, O_i> \rightarrow <M_i, o, O_j>$ or $<S_i, o, O_i> \rightarrow <S_j, o, O_j>$ is represented explicitly in the waits-for graph, because the arcs in the waits-for graph are all associated with specific requests, and therefore any operation at the tail of an arc is always fully-specified. Second, the nodes representing the operations $<S_i, o, O_i>$ and $<M_i, o, O_i>$ in the second situation, and $<S_i, o, O_i>$ and $<S_j, o, O_i>$ in the third situation are distinct, therefore no easy-to-locate structures such as cycles form automatically in the waits-for graph. Fortunately, we can deal with these two problems separately, building an *augmented waits-for graph* which does have specific structures associated with deadlocks.

First, let us deal with the lack of representation of sets of members queued for the same (read or write) operation on the same object. This situation arises, for example, when more than one member requests an intention for the same operation on the same object, and these intentions are queued. If enough such intentions are queued, we can deadlock (see the example in Figure 10(a)). Here, we create *generalizes* nodes and arcs with consolidated representations of each such set of intentions. This is shown in Figure 10(b). Note also that these new nodes and arcs add no information that is not already in the waits-for graph; they merely consolidate information that is already present.

Figure 10. Deadlock situation



(a) As seen in waits-for graph, (b) Waits-for graph after generalizes node and arc are added, (c) Waits-for graph with generalizes node and arc and specializes arcs.

For the second problem, we add arcs to the graph connecting related nodes. First, for each node labeled with an unbounded member set such as $\overline{M}$ or *any*, compute a bounded member set based on the transaction group's current members. For each pair of distinct nodes $<S_1, o, O>$, $<S_2, o, O>$, where $S_1 \subseteq S_2$, we create a *specializes* arc. An example of this is shown in Figure 10(c). The specializes arcs also add no new waits-for relationships to the waits-for graph, but merely help to relate and consolidate waits-for information pertaining to read (or write) intentions on a single object. We see that when we add both the generalizes nodes and arcs and the specializes arcs

to the waits-for graph, directed cycles are created in the second and third deadlock situations.

Theorem 1. *An intention corresponding to node n in the waits-for graph is not deadlocked if and only if you can reach a sink from n in the augmented waits-for graph.*

The proof for this theorem is found in (Nodine, 1991).

4.2.4 Detection and Resolution of Intention Deadlocks. Given the previous theorem, we can detect deadlocks within a transaction group using the following algorithm:

1. Copy the waits-for graph into a new graph G'.
2. Create the generalizes nodes and arcs in G'.
3. Bound the unbounded sets and add the specializes arcs to G'.
4. Repeat until nothing is deleted:
   (a) Select a sink node s in G'.
   (b) For each node n having an outedge that is also an inedge to s, delete all of its outedges.
   (c) Delete s.
5. Return the set of operations corresponding original nodes in the waits-for graph that remain after the previous step.

The algorithm takes $O(V^2)$ time, where V is the number of nodes in the original waits-for graph.

When an intention is deadlocked, the members of the transaction group whose queued intentions are involved in the deadlock are notified of the problem. The notification specifies both which of the member's queued intentions are involved, and which other members are involved. The members can then cooperate to resolve the problem by selectively aborting operations.

The effect of aborting an operation is to change the current state of one or more operation machines, altering the structure of the waits-for graph as a consequence. Factors involved in choosing which operation(s) to abort include:

- Break as many cycles as possible to disconnect the strongly-connected component.

- Remove arcs associated with the fewest number of intentions (fewest number of machines in *waits-for-machines* on the arc).

- Abort the most recently accepted operation, so as to minimize the number of operations aborted consequently in the invalidation phase.

If a new member joins the transaction group, this may also resolve the intention deadlock, especially in situations where there is some intention queued waiting for anyone else to do some operation; in other words, the head of the waits-for relationship is of the form $<\overline{M}, o, O>$ or $<any, o, O>$.

4.3 Global Detection of Intention Deadlocks. Detecting and resolving intention deadlocks locally within each transaction group is adequate to ensure that no global intention deadlocks occur. This can be seen by examining the structure of the local waits-for graph. We noted in Section 4.2.1 that operations may be queued waiting for other operations either within the same transaction group or within the parent. No waits-for relationships point "down" the cooperative transaction hierarchy. Thus, no waits-for relationships that point "up" the hierarchy can be involved in a cycle associated with some intention deadlock. Thus, any waits-for relationships that form a cycle must be associated only with local intention deadlocks, and detecting intention deadlocks locally within each transaction group is sufficient to detect all intention deadlocks in the hierarchy.

5. Histories and Logging

A *history* is the partial order of operations associated with a transaction group. The operations in a transaction group's history include those executed in the group by its members, and those executed by the group in its parent. The operations in the history, along with other information needed for recovery, are recorded in a *log*. The log for a transaction group is updated as the members execute their operations. Operations that execute simultaneously may be recorded in any order. The log also records the dependencies between operations. When an operation fails or aborts, the log is used to determine which operations are affected by the failure, both directly and indirectly.

5.1 Histories. A history is a partially-ordered sequence of operations. An operation is represented in the history as a tuple $<M,o,O>$ with the fields as in the operation request. The history is partially ordered according to a *happens-before* ordering $(<)$, where if op_1 and op_2 are operations, and if op_1 executes before op_2), then $(op_1 < op_2)$.

The history conforms to the following constraints:

- It contains exactly one entry for every operation that has executed but not aborted.
- It contains no operations that either have not executed or that have been aborted.
- Any operation op_1 in the history that depends on some operation op_2 has $(op_2 < op_1)$.

5.2 Operation Dependencies. Dependency information in a cooperative transaction hierarchy is recorded at the granularity of operations rather than transactions. We use these dependencies among operations in a history to determine what is affected, both directly and indirectly, when an operation aborts. This information is needed to ensure that the database is recovered to a correct state.

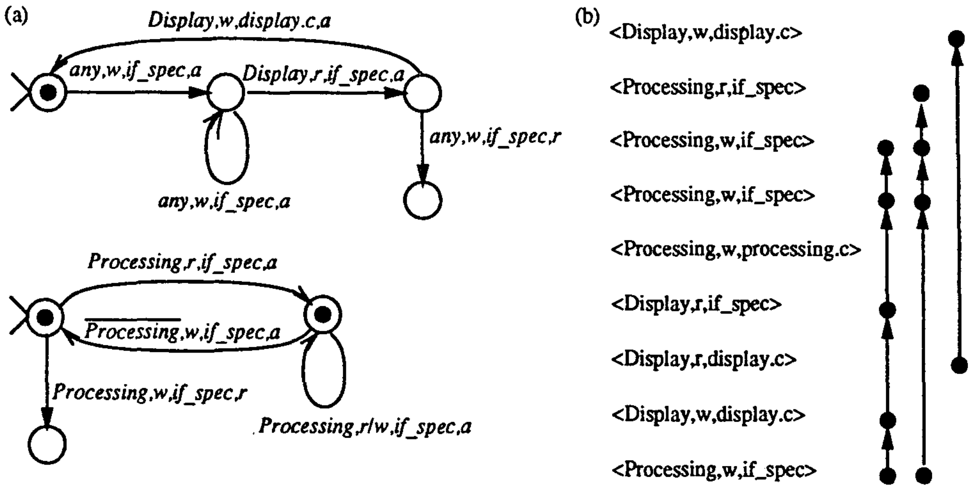
For each defined pattern in a transaction group, *pattern dependencies* are formed among operations that participate in that pattern. Since patterns define allowable operation sequences, each operation in the sequence relevant to some pattern depends on the correctness of the previous operations in that sequence. However, since the sequence of operations associated with a pattern is fully ordered in the history, the definition of a pattern dependency can be simplified to say that each operation *op* is pattern-dependent on the previous operation in each pattern in which *op* 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 adjacent 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 most recent write operation by one of its members.

Figure 11(a) shows two operation machines associated with the *Root* transaction group, and Figure 11(b) shows an excerpt from the history of the *Root* transaction group containing traversals of those machines. To the right of the operation sequence in (b) are three columns of dependencies. The leftmost column shows the pattern dependencies associated with the pattern machine (upper machine in (a)). The middle column shows the pattern dependencies associated with the synchronization machine (lower machine in (a)). The rightmost column shows the read-write dependency associated with the *display.c* object. This set of dependencies is by no means complete, in that there are other machines associated with the members and objects in the history that we have not shown, and therefore we have not shown the dependencies associated with their traversals as well. An example of a machine not shown is the synchronization machine for the *Display* transaction's interaction with the *display.c* object. Also, since this is an excerpt from the *Root* transaction group's history, no parent-child dependencies are shown.

Figure 11. Operation machines and associated history.



(a) Two operation machines. (b) Associated history excerpt with dependencies.

The dependencies associated with a specific operation are computable. Each operation participates in some subset of the patterns defined in the transaction group's active set of operation machines. Since each operation has at most one pattern dependency per active machine, the number of pattern dependencies is upper-bounded by the number of operation machines active in the transaction group. Also, since each operation touches a single object, it can have at most one reads-from dependency and at most one parent-child dependency.

5.3 Logs. A log records the history of a transaction group and the associated dependency information needed for recovery. The log is created as the transaction group and its members execute their operations. If a member *M* of *TG* is itself 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.

An entry in the log is a tuple $\mathcal{LE} =< I, M, o, O, S, V, D >$, where

I is M's unique identifier for the operation as specified in the request.

M is the identifier of the member that did the operation.

$o \in \{r, w\}$ is the operation, where r is read and w is write.

O is the object identifier of the target object.

$S \in \{SUBMITTED, PENDING, COMPLETE, INVALID\}$ is the operation state.

V points to the (possibly null) version created by the operation. Versions are used for recovery only.

D is the set of log entries for operations on which this one depends.

Log entries are identified uniquely by $<I, M>$ pairs. They are totally ordered in the log, with their order consistent with the happens-before order in the history.

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.

6. Invalidation and Recovery

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

6.1 Invalidation

6.1.1 Overview. The invalidation phase begins when some member aborts one of its operations. The abort process begins by marking the log entries for the aborted operations as *INVALID*. It then finds the operations that transitively depend on the aborted operation using the *pattern*, *reads-from* and *parent-child* dependencies in the log, and invalidates them in the same way. It also invalidates any operations that conflict in the new history and the operations that transitively depend on them. Note that because of the parent-child dependencies, invalidations not only cascade from member to member in the transaction group, but also may cascade up and down the cooperative transaction hierarchy.

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 history. Thus, all effects caused by invalidated operations are purged from the database.

Once all affected operations are invalidated, 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 11 from the previous section showed an excerpt of a history containing dependencies associated with the traversal of the pattern machine from Figure 11(a). Consider what happens if *Processing* aborts its second *<Processing,w,if_spec>* operation. The operations *<Display,r,if_spec>*, *<Display,w,display.c>* and *<Processing,w, if_spec>* are also invalidated because of pattern dependencies. The pattern machine is reset to be in the initial state, and the versions created by the two write operations in its traversal are invalidated. The synchronization machine is backed up one transition because the last operation in its traversal is invalidated. The last valid version of the *if_spec* object is the one created by the first write. The last valid version of the *display.c* object is the one that was current just before the write operation *<Display,w,display.c>*. No other operations are invalidated because of dependencies shown in Figure 11, but there may be other operations invalidated because of pattern or reads-from dependencies not shown in the figure.

As a second example, consider what happens if the last *<Display,w,display.c>* operation is aborted. Then the last *<Processing,w,if_spec>* operation must also be invalidated because the upper machine in (a) is in a different state, and now with any write of *if_spec* conflicts.

6.1.2 Correctness. The database is left in a correct state after the invalidation phase. Assume that the database was correct before the abort occurred. The abort breaks some traversals relevant to the history. For each such traversal, the first invalidated operation is called the *breakpoint.* Because of the pattern dependencies, all operations in the traversal after the break point also are invalidated. The only part of the traversal left in the history is the part preceding the break point. Thus, for each pattern in the history, we are left with a correct traversal ending just before the break point.

New conflicts appear because not all patterns are in the same state at a given point in the history as they were during the original processing. An operation at a specific point may conflict in the new history even though it was accepted in the old history. In the invalidation process, we maintain a new conflict list and check each operation in turn against that list. Any operation that conflicts in the new history is correctly found because it is present in the new conflict list at the time it is processed in the invalidation phase. Note that the invalidation of conflicting operations may cause invalidation to cascade to other traversals.

The reads-from dependencies show places where one member's read depends on another member'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. Similarly, the parent-child dependencies indicate where an operation

participates in traversals in more than one history (the history of the member and the history of the parent). Invalidations may also cascade from traversal to traversal within a single transaction group's history when an invalidated operation participates in more than one pattern.

Correctness with respect to the patterns is maintained because the use of pattern dependencies guarantees that all operations in a traversal that follow its break point are 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 this cascading 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.

6.2 Recovery

6.2.1 Overview. Once the invalidation phase is complete, 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 fails.

2. Reread any invalid object versions previously read by the member. This allows the member reread what it has seen, and use that information to determine what operations to submit for recovery. Since the read of old versions involves allowing the member to remember only what it is already seen, it bypasses the synchronization mechanism.

3. Request new operations to recapture any work that was lost inadvertently as a result of the abort. This assumes that the application is either driven by a user or is intelligent, in that it can analyze what needs to be done to recover. During recovery, correctness is ensured because operations are scheduled and processed in the normal manner.

Given the first example above, *Display* could recover after the invalidation phase by rereading the *if_spec* object, adjusting the code for the part of the display module that implements the interface appropriately, and writing the new *display.c* object.

6.2.2 Correctness. The recovery phase maintains correctness. This is shown by proving that each action a member 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 it was in immediately before its traversal's break point, these operations continue existing traversals correctly. Therefore, they maintain correctness as well, so the recovery phase as a whole is correct.

7. 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, or intention deadlocks.

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 that made the request. When the member no longer is interested in the operation, it can terminate the request using a *notification cancellation*.

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 $\mathcal{NR} = <N, CP, o, O>$, where

N is a unique notification identifier,

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

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

O 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 associates the notification ID and the path to the member that sent the request with the object ID specified in the request. It then forwards the request up the hierarchy.

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 $\mathcal{NA} = <RP, N>$, where

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

N is the notification identifier from the notification request.

Each notification message is routed down the tree, following its specified path *RP*, until it reaches the member that made the initial request.

A notification cancellation by a member terminates the corresponding notification request. A notification cancellation is a tuple $\mathcal{NC} = <N, o, O>$, where *N*, *o*, and *O* are as specified in the notification request. 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 notification ID *N* to remove the notification request from the list associated with object *O*.

8. Related Research

Several proposals have been made for supporting more flexible and long-lived transactions. The approaches related to ours include nesting the transactions (Moss, 1985; Korth, 1987), augmenting traditional locking protocols (Skarra, 1991), and specifying a longer transaction as an envelope that contains a sequence of shorter transactions (Garcia-Molina, 1987).

Nested transactions (Moss, 1985) 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. (Kim, 1984) 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. Because these transaction models require (at some level) that child transactions of the same parent are serializable, they work best in design environments where the tasks decompose easily into small, independent subtasks.

Klahold et. al. (Klahold, 1985) 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, for example NT/PV (Korth, 1987, 1988) 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 2PL)*. This model defines the constraints implicitly; the users cannot tailor them to the task at hand. This work was later extended to define a weaker notion of correctness *(entity-wise serializability)* (Korth, 1990). This model also includes work on compensation-based recovery.

Sagas (Garcia-Molina, 1987) define a way of breaking up a longer transaction into shorter ones, and using compensation for recovery when the longer transaction fails. Recently, sagas have been generalized to *nested sagas* (Garcia-Molina, 1990). Both schemes allow arbitrary interleaving of the shorter transactions or nested sagas within a specific saga.

Multilevel atomicity (Lynch, 1983) 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. 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 the flexible transaction model proposed by Kaiser (1990) operation transformation for groupware

systems by Ellis and Gibbs (1989) and the flexible consistency model of Sutton, Jr. (1990).

An approach to process synchronization similar to our transaction synchronization mechanism is *path expressions* (Campbell, 1974). Path expressions are regular expressions that define how operations on a single module should be synchronized. It is like our notion of *patterns* (without conflict), except that patterns are more expressive, in that they can be defined over multiple objects. Also, patterns can restrict *who* does an operation, as well as *when* it can occur.

Version control systems such as RCS (Tichy, 1982) are normally used by groups of designers to control concurrent access to files. Our work is more expressive than these, both in that it allows other types of protection than just preventing the overwriting of files, and that it allows explicit cooperation.

Transaction groups as used in this paper were first defined by Fernandez and Zdonik (1989).

We have also taken the use of *patterns*, *conflicts*, and *operation machines* to specify correct histories from Skarra's work (1989). 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. Her work touches mainly on methods of representing correctness, especially when the methods associated with the objects can be nested. While she examines the synchronization problem, she does not address recovery or deadlocks at all. Also, she keeps only a single copy of each object at the root of her transaction hierarchy, while we keep private copies for each transaction group that is accessing the object. Her model is more complex than ours, in that her patterns allow arbitrary variables on their transitions; thus, patterns can be used to emulate a Turing machine. Based on our work in (Nodine, 1991), we feel that deciding online whether or not a history is correct in her scheme may not be possible, because it potentially requires looking ahead in the history.

9. Conclusion

Cooperative transaction hierarchies are a new framework for providing database support for cooperative applications such as design applications. Serializability is not necessarily an appropriate requirement for transactions in cooperative applications, because these transactions tend to be open-ended, long-lived, and interactive.

A cooperative transaction hierarchy reflects the structure of the underlying design task. Each node in the hierarchy is responsible for some specific subtask, and has its own private set of versions of the objects it is currently using in accomplishing that

subtask. The internal nodes (*transaction groups*) accomplish their tasks through the cooperation of their children (*members*). The leaf nodes (*cooperative transactions*) represent individual designers or individual design applications.

We have defined a programmable synchronization method based on *patterns* and *conflicts* that structures the interactions among cooperative transactions and allows for controlled data sharing. Conflicts are like locks in the sense that they specify when certain operations cannot occur. Patterns specify operation sequences that must occur in a history for it to be correct. *Operation machines* are used to specify the patterns and conflicts. We also provide *notifications* so that different transactions can receive messages relevant to their work.

Cooperative transactions are not atomic. The largest unit in a cooperative transaction that is guaranteed to be atomic is an operation. Because of this, deadlocks occur among operations rather than transactions. This means that deadlock detection and resolution occur at the level of operations. Deadlock resolution is also affected by the queueing semantics of cooperative transactions, which dictate that a cooperative transaction may continue to operate even if it has queued operations. We have specified how deadlocks can be detected in the cooperative transaction hierarchy, and also given guidelines for deadlock resolution.

Because cooperative transactions are inherently long-lived and interactive, we also allow them to abort individual operations (as opposed to entire cooperative transactions). We maintain operation dependencies in the log of each transaction group. When an operation is aborted, we use these recorded dependencies to limit the operations that are undone to those dependent on the aborted operation and those that conflict in the new history. Thus, while the effects of an abort may cascade from one cooperative transaction to others, we restrict that cascading to the *operations* affected by the original abort.

Once a failure or abort has occurred, and the effects of all of the dependent operations are removed from the database, we allow the members of a transaction group to cooperate in recovering from that point. The operations that can occur during recovery must conform to the same synchronization specifications as the original operations.

Applications that require interaction, such as design applications, may place nontraditional requirements on any underlying database support. Patterns and conflicts provide a useful correctness specification for these applications. In this paper, we have summarized how they can be used, and how they affect basic database operations such as synchronization, deadlock detection, and recovery. These schemes are meant to replace the more traditional schemes used in centralized databases. Extensions to other types of architectures, such as distributed architectures, should be feasible. However, their implementation is a subject of further research.

Acknowledgments

Support for this research was provided in part by an IBM Graduate Fellowship, and in part by IBM under contract No. 559716, by DEC under award No. DEC686, and by ONR under contract N00014-83-K-0146. The authors also would like to thank Andrea Skarra and Mary Fernandez for their helpful insights.

References

Campbell, R.H. and Habermann, A.N. The specification of process synchronization by path expressions, *Lecture Notes in Computer Science*, 16:89-102, 1974.

Ellis, C.A. and Gibbs, S.J. Concurrency control in groupware systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Portland, OR, 1990, pp 399-407.

Fernandez, M. and Zdonik, S. Transaction groups: A model for controlling cooperative transactions. *Third International Workshop On Persistent Object Systems,* Newcastle, Australia, 1989.

Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., and Salem, K. Coordinating multi-transaction activities. *Technical Report CS-TR-247-90,* Princeton, NJ: Princeton University Department of Computer Science, 1990.

Garcia-Molina, H. and Salem, K. Sagas. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* San Francisco, CA, 1987, pp 249-259.

Haerder, T. and Rothermel, K. Concepts for transaction recovery in nested transactions. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* San Francisco, CA, 1987, pp 239–248.

Kaiser, G.E. A flexible transaction model for software engineering. *Proceedings of the 6th International Conference on Data Engineering,* Los Angeles, CA, 1990.

Kim, W., Lorie, R., McNabb, D., and Plouffe, W. A transaction mechanism for engineering design databases. *Proceedings of the 10th International Conference on Very Large Databases,* Singapore, 1984, pp 355-362.

Klahold, P., Schlageter, G., Unland, R., and Wilkes, W. A transaction model supporting complex applications in integrated information systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Austin, TX, 1985, pp 388-401.

Korth, H.F., Kim, W., and Bancilhon, F. On long-duration (CAD) transactions. *Information Sciences,* 46:73-107, 1988.

Korth, H.F., Levy, E., and Silberschatz, A. A formal approach to recovery by compensating transactions. *Proceedings of the 16th International Conference on Very Large Data Bases,* Brisbane, Australia, 1990, pp 95-106.

Korth, H.F. and Speegle, G.D. Formal model of correctness without serializability. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Chicago, IL, 1988, pp 379-386.

Lynch, N.A. Multilevel atomicity–a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems,* 8:484-502, 1983.

Eliot, J. and Moss, B. *Nested Transactions: An Approach to Reliable Distributed Computing.* Cambridge, MA: MIT Press, 1985.

Nodine, M.H. Conflict, queueing and deadlocks in cooperative transaction hierarchies. *Technical Report CS-91-27,* Providence, RI: Brown University Department of Computer Science, 1991.

Nodine, M.H., Ramaswamy, S., and Zdonik, S.B. A cooperative transaction model for design databases. In: Elmagarmid, A., ed. *Database Transaction Models for Advanced Applications.* San Mateo, CA: Morgan Kauffman, 1992.

Nodine, M.H., Skarra, A.H., and Zdonik, S.B. Synchronization and recovery in cooperative transactions. In: Dearle, A., Shaw, G.M., and Zdonik, S.B., eds.*Implementing Persistent Object Bases: Principles and Practice,* San Mateo, CA: Morgan Kaufmann, 1990, pp 329-342. Also:*Proceedings of the 4th International Workshop on Persistent Object Systems,* Martha's Vineyard, MA, 1990.

Nodine, M.H. and Zdonik, S.B. Cooperative transaction hierarchies: A transaction model to support design applications. *Proceedings of the 16th International Conference on Very Large Databases,* Brisbane, Australia, 1990, pp 83-94.

Skarra, A.H. Concurrency control for cooperating transactions in an object-oriented database. *SIGPLAN Notices,* 24:4 (1989).

Skarra, A.H. Localized correctness specifications for cooperating transactions in an object-oriented database. *Office Knowledge Engineering,* 4:1 (1991).

Skarra, A.H., Zdonik, S.B., and Reiss, S.P. An object server for an object-oriented database system. *Proceedings of the International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, 1986, pp 196-204.

Sutton, S.M., Jr. A flexible concurrency model for persistent data in software-process programming languages. Dearle, A., Shaw, G.M., and Zdonik, S.B., eds. *Implementing Persistent Object Bases: Principles and Practice,* San Mateo, CA: Morgan Kaufmann, 1990, 305-318. Also: *Proceedings of the 4th Int'l Workshop on Persistent Object Systems,* Martha's Vineyard, MA, 1990.

Tichy, W.F. Design, implementation, and evaluation of a revision control system. *Proceedings of the 6th International Conference on Software Engineering,* Tokyo, Japan, 1982.