Conflict, Queueing, and Deadlocks in Cooperative Transaction Hierarchies

Marian H. Nodine

Technical Report No. CS–91–27
May, 1991

Conflict, Queueing, and Deadlocks in Cooperative Transaction Hierarchies

Marian H. Nodine
Brown University
Providence, RI 02912

May, 1991

Abstract

Cooperative transaction hierarchies are a model for transactions on databases that support design applications. This model relaxes the atomicity constraint, allowing controlled interactions among different cooperative transactions.

This work summarizes earlier work in [NFSZ90], and describes the nature of conflict and queueing in the context of that work. Since queueing can cause deadlocks, we discuss the nature of deadlocks in a cooperative transaction hierarchy, and propose and algorithm for deadlock detection. We give some pointers on how to resolve deadlocks. We also show that determining deadlock freedom, which would be preferable to deadlock detection, is NP-hard.

1 Introduction

Cooperative transaction hierarchies are a model for transactions that support design applications. Design applications place certain requirements on their underlying databases that traditional transaction models do not satisfy. In particular, design applications tend to generate long, open-ended transactions. Also, since people tend to interact during the design process, design applications may need to support transactions that interact, or cooperate. The transaction model must be able to both support this kind of interaction, and prevent unwanted interaction from occurring.

Cooperative transaction hierarchies relax the requirement that transactions be atomic. Instead, design transactions work together in groups, and each group insists that its design transactions interact with the database only in ways it allows. It does this by controlling the order in which operations can occur (*patterns*) and cannot occur (*conflicts*). [NFSZ90] contains a complete description of the cooperative transaction hierarchy model.

Because cooperative transaction hierarchies relax atomicity, there is a sense in which the whole model is more oriented towards operations rather than transactions. With serializable transactions, all the modifications done by a single transaction must appear to occur at the same time. Since cooperative transaction hierarchies allow different transactions to interact, only operations must appear to happen as a single unit. Consequently, it becomes sensible to allow, for example, aborts of single operations to occur without triggering an abort of the whole transaction. This orientation also affects the nature of transaction conflict – how it occurs as well as how the database management system handles it.

In this paper, we examine the nature of queueing and deadlocks in a cooperative transaction hierarchy. Queueing occurs when specific operations are delayed because they conflict at the time they are submitted. In a cooperative transaction hierarchy, operations are queued waiting for other operations to occur. Because operations can be queued, we can also have deadlocks between operations. We examine the nature of deadlocks in a cooperative transaction hierarchy, and give both an algorithm for deadlock detection, and factors that may be important in deadlock resolution.

The rest of the paper is organized as follows: Section 2 briefly describes the cooperative transaction model. Section 3 reviews the synchronization protocols, and reviews intentions. These two sections summarize earlier work. We explore the resolution of conflict by delaying or queueing operations in Section 4. Since queueing schemes are prone to deadlocks, we discuss deadlock detection and resolution in a cooperative transaction hierarchy in Section 5. Finally, we summarize this work in Section 6.

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*. The *Root* transaction group is at the top of the hierarchy. Each 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.

This section describes the basic cooperative transaction hierarchy model, including definitions of transaction groups, cooperative transactions, and operations. The features of each are summarized in Figure 1.

2.1 Transaction Groups

A transaction group is a set of members that are directly cooperating on a specific task. Because of the cooperative nature of the members, we do not assume that the sequence of operations by a single member necessarily leaves the database in a correct state. Instead, each transaction group has its own *correctness specification*, which consists of a set of active *patterns* and *conflicts*. Patterns state ways that the members operations must be ordered complete the task. Conflicts specify how the members operations cannot be ordered, to prevent unwanted side-effects. The transaction group ensures that the members interact only

Transaction Group	• Is not atomic. • Corresponds to some task that is done by its members. • Enforces its own correctness criteria. • Mediates communication among its members. • Operates on its own object copies. • Is a unit of recovery. • Makes own guarantees about the recoverability of its object copies.
Cooperative Transaction	• Is not atomic. • Is the collection of operations by a single member.
Operation	• Is atomic. • Is specified by a tuple $<M,o,O>$, where M is a member. o is an operation specifier. O is an object identifier.

Figure 1: Basic Features.

in the ways allowable by its correctness specification, and in that way guarantees that the operations by its members as a group leave the database in a correct state.

Each transaction group has a local set of object versions. Thus, there may be many versions of an object scattered throughout the transaction hierarchy. For a specific object, the version at the root is the oldest version, and the ones further down are more recent. The version of an object in the group's object set is accessible to any of its descendants in the transaction hierarchy that does not have access to a more recent version.

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 operations must conform to the transaction group's correctness specification.

We allow the individual members to commit or undo parts of their work as it progresses. When some subtask is complete, we allow the member to *checkpoint*. A member checkpoint causes all of its operations that have been neither checkpointed nor aborted to be checkpointed. The effect of the operation checkpoint is to propagate the effects of the operations to the group's parent. This is done by encapsulating sequences of operations by the group's member(s) into a shorter sequence of operations initiated in its parent by the transaction group. This sequence of operations must be equivalent to the sequence checkpointed in the transaction group, in that it has the same effects on the objects. For example, several write operations on a single object version in the transaction group's object set may be collapsed into a single write operation on the object version in its parent's set.

We allow a member to selectively undo certain parts of its work by aborting individual uncheckpointed operations. Aborting an operation may mean that other operations need to be aborted as well, either because they read the effects of the aborted operation, or because the abort caused the history to become incorrect in some way.

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

2.2 Cooperative Transactions

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

During its lifetime, a cooperative transaction issues operations on object versions in its transaction group.

These operations are executed by the transaction group if they conform to its correctness specification. Operations may be refused by the transaction group if they do not conform to the correctness specification. As with transaction groups, as the design task progresses, operations may be either aborted individually or be checkpointed when the cooperative transaction has completed some subtask.

2.3 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. Eventually, the effects of an operation are either propagated towards the *Root* transaction group or invalidated by one of the members of the group.

An operation is a tuple $\mathcal{O} = <M, o, O>$, as specified in Figure 1. In this paper, we examine only ***read***, ***write***, ***increment*** and ***decrement*** operations.

When a member M submits an operation to its parent transaction group P, it must be recognized as correct according to P's correctness criteria. Once we know that its execution at this time does not conflict, it is executed. If M decides later that it is incorrect in some way, it can abort the individual operation. Otherwise, the operation checkpoints the next time M checkpoints, and the process of making any changes resulting from the operation more permanent begins. However, checkpointing an operation does not guarantee that it cannot be undone. For example, it may be undone because it depends on an operation by some other member of P that later is aborted. Once an operation cannot be undone for any reason, it is called ***complete***.

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, internal correctness constraints. A transaction group's synchronization process ensures that the members of the transaction group together 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 necessarily constrain the members' operations to a specific execution; rather they should specify the general form of the allowable member interactions. We use patterns and conflicts to specify required and prohibited operation sequences by the members of a transaction group, as well as the interleaving of the members' operation sequences.

Operation machines are user-definable synchronization mechanisms for specifying patterns and conflicts. They were first proposed by Skarra [Ska91]. An operation machine is a finite-state automaton. Each transition in an operation machine is labeled with the symbol σ that defines the operation associated with it, where $\sigma = <M, o, O, P>$, and

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

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

O is an object identifier, 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. It is indicated diagrammatically by an arrowhead pointing to it. The current state indicates the current place in the pattern, and is represented by a solid token in the circle representing the state. Machine transitions represent operations on an object by some member. A transition is represented as a labeled, directed arc from one state to another state. The transitions from a specific state define how the operation fits into the pattern if that state is the

current state. Transitions are annotated with return values that are either *accept* if the operation conforms to the pattern, *refuse* if the operation conflicts, or *queue* if the operation conflicts now but may conform to the pattern if done later. If a transition is annotated with *reject* or *queue*, it always pointst to a *dead* state. These arcs are never traversed. 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. They are indicated by hollow circles inside the circle representing the state.

In a specific transaction group, operation machines in its 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 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 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 can 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 in its parent. When a member M of a transaction group TG is granted an intention to do an operation, TG's version of the object is restricted in such a way that no other operations can be done that will later cause the intended operation to be rejected. 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 done, provided that it is currently acceptable according to the patterns and conflicts defined in its parent. It also may be queued or refused.

3.3.1 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 set of their respective transaction groups for the duration of the intention.

Figure 2 shows an example of the intention machines that are put in place when the transaction group TG accepts an intention request for the operation $<M, w, O>$. Figure 2(a) shows the machine bound to O at TG's level. It prevents any write of the object by any other member, while allowing exactly one write by M. This machine also allows anyone to read the original version of O 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 2(b) shows the machine specifying the allowable operations by the members of M that can be batched into the single intended write in TG. 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 M has already read the O object.

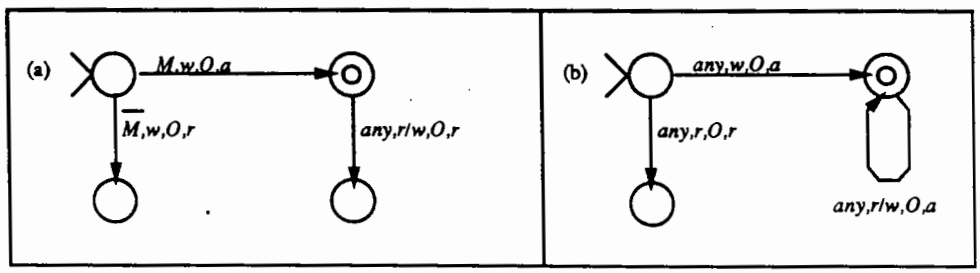

Figure 2: Intention machines: (a) at TG's level; (b) at M's level.

3.3.2 Implementation

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

Intention requests cascade up the transaction hierarchy until 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 and Queueing

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 which prevents the requested operation from taking place immediately. Conflict may also occur with respect to intentions. That is, the member may be requesting an intention instead of actually doing an operation. 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 set of queued requests waiting to be accepted or refused. Queued requests are kept on a *request queue*, with older requests at the head of the queue and later requests towards the tail. When we need to search for queued operations that may be accepted or refused at a given time, we search the request queue from its head to its tail. The set does not mimic a FIFO queue, however; two items in the set may be removed from the set in a different order than the one in which they were inserted. This would occur, for instance, if the two operations op_1 and op_2 were relevant to different patterns. Even if op_1 were requested before op_2, the patterns relevant to op_2 could still make a decision to accept or reject it before a decision could be made about op_1.

When an intention request is queued, a copy of the request is placed at the tail of the queue of requests. An *intention QUEUE* message is sent to the requester, indicating that the intention has been queued. 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 submit the operation or release the intention. These message sequences are shown in Figure 3.

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 4. The purpose of this pattern is to ensure that member M cannot overwrite some other member's changes. Assume we are in the start state. The operation request $<M,w,O>$ is submitted and 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, the member M issues an operation request $<M,r,O>$, which would be accepted. Now the queued request $<M,w,O>$ would be accepted, and the specified version written. This violates the spirit of the pattern, because the new version specified by the write operation 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 $<M,w,O,q>$ 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 placed at the tail of the request queue. Another way to look at queuing is that its purpose is to note that the member wants to do the operation, so that we can inform the member later when the operation can be done.

When a queued intention is accepted, the member that requested the operation is told that it has the appropriate intention. The member then either can 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.

The message sequence for queued operations is similar to the message sequence for queued intentions shown in Figure 3, except that the initial requests are *operation requests* instead of *intention requests*.

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

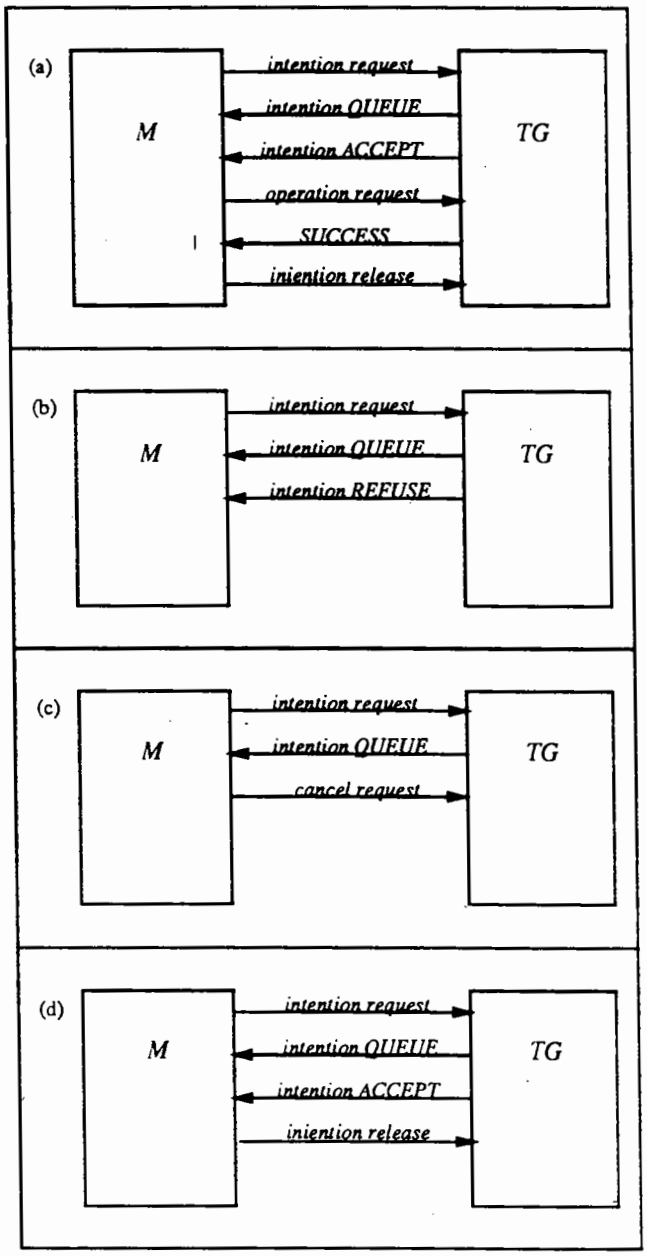

Figure 3: Message sequence for queued operations; (a) Accepted, (b) Refused, (c) Canceled, (d) Intention Released.

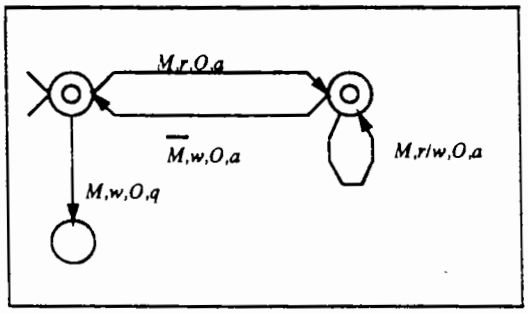

Figure 4: Why write operations cannot be queued as operations.

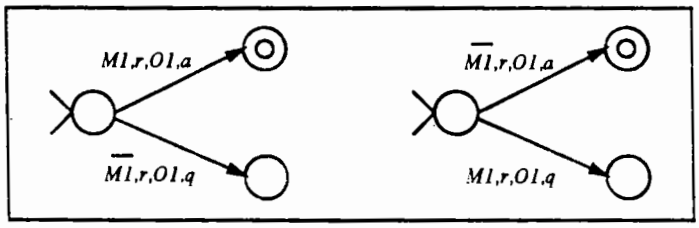

Figure 5: Deadlock example.

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.

5 Intention Deadlocks in a Transaction Group

As with locking systems, a transaction group may have deadlock-type problems. 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 5. If both operation machines are in the start state, then the only operations which allow either machine to reach a final state are $<M1,r,O1>$ for the first machine and any operation specified in $<\overline{M1},r,O1>$ for the second machine. Unfortunately, $<M1,r,O1>$ is queued by the second machine, and any operation in $<\overline{M1},r,O1>$ is queued by the first machine. Therefore, if both these machines are a part of some group's correctness specification, deadlock will occur once intentions for both $<M1,r,O1>$ and some operation in $<\overline{M1},r,O1>$ have been submitted and queued.

5.1 The Waits-For Graph

Each transaction group is responsible for maintaining its own *waits-for graph* that characterizes which intentions for operations are queued 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 has a transition for that operation that currently returns *queue*, and that machine would accept op_2. In the example in Figure 5, if *M2* submits an intention to do the operation $<M2,r,O1>$, the waits-for relationship $<M2,r,O1>\rightarrow<M1,r,O1>$ ("$<M2,r,O1>$ waits for $<M1,r,O1>$") occurs. Similarly, if *M1* submits an intention to do the operation $<M1,r,O1>$, the waits-for relationship $<M1,r,O1>\rightarrow<\overline{M1},r,O1>$ occurs.

At any given time, the waits-for graph contains a node for each operation that is involved in some waits-for relation, and an arc between any pair of operations involved in a specific waits-for relation, indicating the direction of the relationship. 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

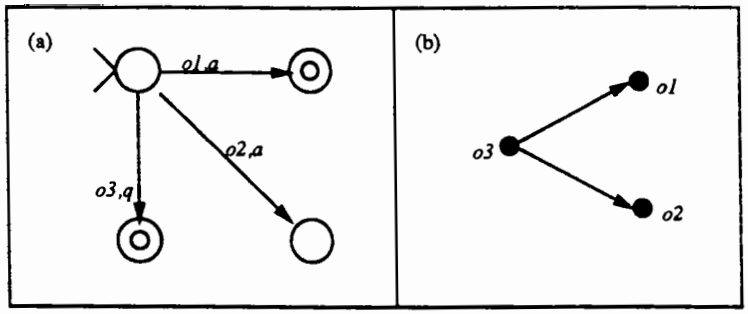

Figure 6: Updating the waits-for graph; (a) An operation machine, (b) Part of the waits-for graph.

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 to the node representing op_2, 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 other machine OM_2, the *waits_for_machines* list on that arc would be augmented to $\{OM_1, OM_2\}$. Also, on the arc $op_1 \rightarrow op_2$ we call op_1 the *tail* and op_2 the *head*.

There are various types of nodes in the waits-for graph. Nodes which are tails of any waits-for relationship represent queued intentions. The corresponding heads represent operations which possibly could allow the queued intention to be resolved. Nodes which are heads may represent any of the following:

- Unsubmitted operations.
- Queued intentions in the transaction group.
- Operations which have intentions granted for them.

We assume that no intention is queued twice. Duplicate intention requests are ignored.

5.2 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 are as follows:

- Queueing an intention.
- Accepting, refusing, or canceling a queued intention request.
- Releasing an intention.
- Doing an operation outside of an intention that is represented by some node in the waits-for graph.
- Aborting an operation relevant to some operation machine governing some waits-for relationship.
- Modifying the transaction group's correctness specification by adding or removing an operation machine.

These changes are not necessarily simple. Consider the operation machine in figure 6(a). Once *o3* has been submitted, the construct shown in Figure 6(b) should be present in the waits-for graph. If *o2* is later submitted and accepted, then the machine from Figure 6(a) would no longer queue *o3*, and therefore it could be accepted. Thus, the waits-for relationship $o3 \rightarrow o1$ must also be removed from the waits-for graph.

5.2.1 Basic Changes

Adding a waits-for relationship to the waits-for graph is done according to the following algorithm:

1. Add nodes for the operations in the waits-for relationship that are not already in the waits-for graph.
2. If there is no arc between the two nodes in the direction of the waits-for relationship, add one.
3. Add the operation machine identifier to the *waits_for_machines* list on the arc.

Removing a waits-for relationship associated with some machine is done as follows:

1. Remove the machine identifier from the *waits_for_machines* list on the arc.

2. If the *waits_for_machines* list is now empty, remove the arc.

3. If either end node no longer has any in- or out-edges, remove the node.

5.2.2 Changes Due to Operations or Intentions

Queueing an Intention: When an intention is queued, we update the waits-for graph to reflect the new waits-for relationships. This is done according to the following atomic algorithm:

1. If the intended operation is already represented in the waits-for graph by some node that already has outedges, do nothing. This intention request is a duplicate of some operation or intention request that has already been queued.

2. Let OM_r be the set of operation machines that are both currently relevant to the queued operation and return q for the intended operation.

3. Let S_a be the set of operations or operation sets[1] currently accepted by some $OM \in OM_r$ and that cause a state change in that machine. This represents the operations that potentially could cause the requested intention to be either accepted or refused.

4. For each $op \in S_a$, add a waits-for relationship to the waits-for graph for each $OM \in OM_r$ that accepts op.

The new waits-for relationships are not necessarily between intentions and single operations. At the tail of each waits-for relationship is a node representing a single intention request. However, the intention may be queued by a machine waiting for some other operation which is a read or write on an object 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 $<\overline{M}, r, O, q>$, as in the example from Figure 5. Thus, the node at the head of a waits-for relationship may have a less-specified label, where the member entry may contain some set S of members.

Accepting or Refusing an Intention: When an intention is accepted or refused, there should be no outedges from its node in the waits-for graph. In other words, the graph should reflect that it is waiting for no other operations.

Releasing an Intention or Doing an Operation: When an intention is released or an operation is accepted outside of the scope of an intention, the following is done:

1. Let N be the node in the waits-for graph representing the operation. If there was no node N, return. Otherwise, N should only have inedges, or it would not have been accepted.

2. Remove the node for the operation from the waits-for graph, as well as all of the waits-for relationships it is involved in. Save these waits-for relationships in a temporary list *(wf_list)*.

3. Sort the elements of *wf_list* by the operations in their tail node, according to their order in the request queue. Process the operations in this order, accepting and refusing any operations that can now be accepted or refused. Operations that are still queued maintain their position in the request queue.

4. Let S_o be the set of intentions that are currently queued. Let S_M be the set of machines in waits-for-machines for any element in *wf_list*.

5. For each $m \in S_M$, check each operation $o \in S_o$ to see if m currently would queue o. If so, add waits-for relationships to the waits-for graph associated with machine m, as specified for queueing an intention.

This case is special since new waits-for relationships come into being because one or more operation machines have changed state. Consider the machine shown in Figure 7, with the current state being the start state. If *o1* is submitted and accepted, then *o2* would be queued if it were subsequently submitted. This presents some difficulties. If an intention for *o2* had been requested and accepted previously, but the

[1] such as $<any, w, O, q>$ or $<\overline{M}, r, O, q>$

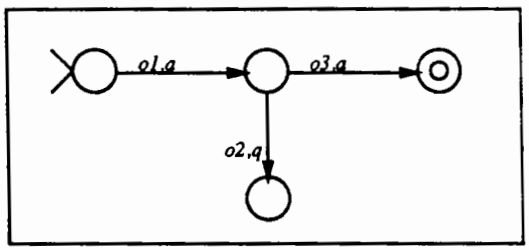

Figure 7: A Problem with Adding Waits-forRelationships.

operation had yet to be submitted, then *o2* could be queued in spite of the accepted intention. If this occurs, we may need to abort *o1* so that *o2* can execute as promised. We want to take an optimistic approach here, since *o2* may not be submitted until after *o3* has executed. In this case, the *o2* would be accepted, as promised.

5.2.3 Changes Because of Operation Abort

Operation abort and the ensuing invalidation phase affect the outstanding intentions as well as rolling back operation machines. In the invalidation phase, once a set of operations has been invalidated, all outstanding accepted intentions need to be re-evaluated against the group's (non-intention) operation machines to see which ones are still accepted. This evaluation is done in the order the intentions were granted. If an intention is still accepted, the intention machine is replaced. Otherwise, the user is notified that the intention was broken by the rollback, and the machine is discarded.

Once the state of all machines is known after the invalidation phase, then the waits-for graph can be adjusted. Any waits-for relationships associated with machines removed during rollback are removed from the waits-for graph. All waits-for relationships associated with machines that were rolled back no longer hold, so they must also be removed. Then the list of queued intentions is processed in the order the intentions were queued, and each intention is checked as follows:

1. If the machines now accept the intention, then accept it and return *intention accept.*
2. If the machines now refuse the intention, then refuse it and return *intention refuse.*
3. If the machines still queue the intention, get the list of waits-for relationships associated with the machines that return *queue* (see the section above on queueing an intention). If any of those relationships are not already represented in the waits-for graph because they are associated with machines that were rolled back, add them to the waits-for graph.

Because operation abort changes the current states of operation machines, it may also be true that the member that requested an intention that was just accepted or re-queued may no longer want to do the operation, and consequently may cancel outstanding intention requests or release outstanding intentions.

5.2.4 Changes Because the Correctness Specification is Modified

Adding a machine: Modifying the correctness specification means either adding new operation machines to the transaction group or removing old ones. Any new machine that is added may be relevant to one or more of the queued intentions. Assuming that adding the machine is allowed (it cannot be allowed if it would cause the operation associated with some intention that has already been granted to be queued or refused), do the following for each queued intention:

1. If the new machine would accept the intended operation, do nothing.
2. If the new machine would reject the intended operation, then refuse the intention.
3. If the new machine would queue the intended operation, add the associated waits-for relationship to the waits-for graph.

Removing a Machine: When a machine is removed, any waits-for relationships associated with the machine should be removed from the waits-for graph.

5.3 The Structure of Intention Deadlocks in the Waits-For Graph

We can now try to characterize the structure of intention deadlocks in the waits-for graph. 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, and the ellipsis indicates a chain of zero or more waits-for relationships.

2. $<S_i, o, O_i> \rightarrow \ldots \rightarrow <M_i, o, O_i>$ where S_i is a set of member IDs (or some representational equivalent like $\overline{M_1}$ or *any*) 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 any node at the tail of a waits-for relationship always represents an intention for a unique operation. 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 nice structures associated with deadlocks.

First, let us deal with the lack of representation of sets of members waiting to do a single operation. 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 8(a)). Here, we create ***generalizes*** nodes and arcs with consolidated representations of each such set of operations. This is shown in Figure 8(b). If the waits-for graph has V nodes, no more than $\frac{1}{2}V$ generalizes nodes are added, because each generalizes node consolidates information from at least two other nodes in the waits-for graph. 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.

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 8(c). The computation and addition of the specializes arcs can be done in $O(V^2)$ time by comparing each pair of nodes in the waits-for graph related to an object and adding the specializes arcs and nodes where appropriate. 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.

For the rest of this section, we will be using a special notation for the nodes and arcs in the augmented waits-for graph. Nodes from the original waits-for graph will be represented as N_a, N_b, etc. The generalizes nodes that are added will be represented as G_a, G_b, etc. An arc in the original graph for some waits-for relationship is written $N_x \xrightarrow{w} N_y$. A generalizes arc is written $G_x \xrightarrow{g} N_y$. A specializes arc is written $N_x \xrightarrow{s} G_y$ or $G_x \xrightarrow{s} G_y$.

Let us examine the structure of the paths in some augmented waits-for graph G'. Choose some path P, and find two consecutive nodes N_a and N_b on it that were present in the original waits-for graph. The path between them can have one of the following forms:

1. $N_a \xrightarrow{w} N_b$. This is an original waits-for relationship.

2. $N_a \xrightarrow{s} G_a \xrightarrow{g} N_b$. This also means that the operation(s) represented by the node N_a are a (not necessarily proper) subset of the operations represented by the generalizes node G_a. The operations represented by the generalizes node G_a are all waiting for the operation(s) represented by the node

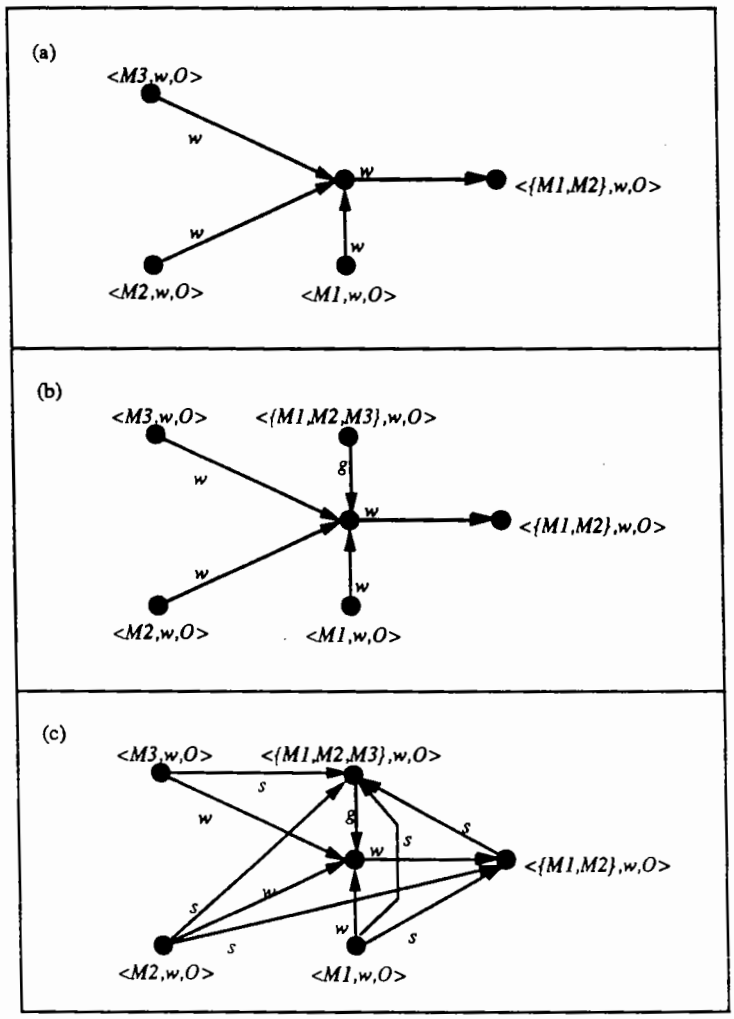

Figure 8: Deadlock situation; (a) As seen in waits-for graph, (b) Waits-for graph after generalizes node and arc are added, (c) Waits-for graph with generalizes node and arc and specializes arcs. The arcs are labeled as follows: original waits-for relationships are labeled w, generalizes arcs are labeled g, and specializes arcs are labeled s.

N_b. Therefore all operations represented by the node N_a are waiting for the operation represented by the node N_b. This is also a type of waits-for relationships, and we can treat such a path segment as if it were of the form $N_a \xrightarrow{w} N_b$.

3. $N_a \xrightarrow{s} G_a \xrightarrow{s} \ldots \xrightarrow{s} G_b \xrightarrow{g} N_b$. In this case, because of the way the specializes arcs are created, we know that if there is a path $N_a \xrightarrow{s} G_a \xrightarrow{s} \ldots \xrightarrow{s} G_b$ in the augmented waits-for graph, there is also a path $N_a \xrightarrow{s} G_b$. Therefore, there is a path $N_a \xrightarrow{s} G_a \xrightarrow{s} \ldots \xrightarrow{s} G_b \xrightarrow{g} N_b$, and this case is identical to the previous one.

No other configurations for the path between N_a and N_b are possible. Chains of generalizes arcs do not occur because the head of a generalizes arc is always some node from the original waits-for graph. All specializes arcs precede the generalizes arc in such a segment because, by construction, any generalizes arc has a node from the original waits-for graph at its head.

For the sake of simplicity, in the remaining discussion we use an equivalent representation of the augmented waits-for graph suggested by the discussion of path structures above. For each segment on some path of the form $N_a \xrightarrow{s} G_a \xrightarrow{s} \ldots \xrightarrow{s} G_b \xrightarrow{g} N_b$, we will consolidate it to the form $N_a \xrightarrow{w'} N_b$, representing an alternative waits-for relationship. The alternative form has no generalizes nodes, nor does it have any generalizes or specializes arcs. We can also treat these alternative waits-for relationships in the same way we treat the original waits-for relationships. Because the two representations show the same relationships explicitly, we know that anything we prove concerning the graph structure in the alternative representation holds true in the original representation of the augmented waits-for graph.

Now we can characterize deadlocks in the augmented waits-for graph.

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.*

We will do this proof using a graph game that emulates the normal operation on the waits-for graph. The graph game allows you to apply the following rules repeatedly to the waits-for graph:

1. You can delete any sink. This corresponds to executing an operation and/or releasing an intention.

2. If you delete a node using rule 1, you must delete all of its inedges. The waits-for relationships no longer hold.

3. If you delete an *outedge* from some node m using rule 2, you can add or delete any outedges from node m. This emulates the change in waits-for relationships because the operation machines changed state.

The objective of the game is to delete as many nodes as possible. Clearly, the number of nodes deleted is maximized by applying rule 1 until there are no more sinks in the graph, always deleting every possible outedge when you apply rule 3, and never adding new edges. If we maximize the number of nodes deleted, it is easy to see that the remaining nodes correspond to operations that are deadlocked.

We can then restate the theorem into an equivalent theorem as follows:

Theorem 2 (Restatement of Theorem 1) *You can delete a node in the graph game described above iff there is a path from that node to a sink in the graph.*

(if) Say there is a path P from node n to some sink s in the graph. If there are multiple paths, choose one with minimal length. We use induction on the length of P. Now, you can delete s and its inedge in P by applying rules 1 and 2. Now, let m be a node on the path. Say you can delete all nodes and edges following it on path P. At this point, m is a sink and you can delete it by rule 1.

(only if) We use induction on the number of nodes in the graph. Say you cannot delete n in the graph game. Then n is clearly not a sink in the original graph; otherwise you could delete it by applying rule 1.

Call an application of the graph game where you choose a sink, delete it using rule 1, and delete the maximum number of edges using rules 2 and 3 an *iteration* of the game. For each node in the graph game, there are two cases:

1. The node is deleted in the iteration. Clearly it could reach a sink.

2. If the node could reach a sink before the iteration, it can still reach a sink after the iteration.

If you maximize the number of nodes deleted in the graph game, then there is no path to a sink for any remaining node, because there are no more sinks in the graph. Since n is in this set (because we hypothesized it could not be deleted), and iterations in the graph game do not change the property of whether or not there is a path from an undeleted node to a sink, you cannot reach a sink from n in the graph. □

5.4 Finding 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'. Let V be the number of nodes in G' and E be the number of edges.
2. Create the generalizes nodes and arcs in G'. This takes $O(V^2)$ time and creates at most $\frac{1}{2}V$ generalizes nodes.
3. Bound the unbounded sets and add the specializes arcs to G'. This takes $O(V^2)$ time because each node may need to be compared with all other nodes to check for specializes relationships.
4. Repeat until nothing is deleted:
   (a) Select a sink node s in G'.
   (b) For each node n that has some 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.

5.5 Resolution of Intention Deadlocks

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 to a member specifies both which of its queued intentions are involved, and which other members are involved. The members can then cooperate to resolve the problem. Actions that can be taken by a member include aborting an operation and introducing a new member.

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.
- 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.

Changing the membership in the group may also help. Introducing a new member may be helpful in situations where there is some intention queued waiting for anyone else to do something; in other words, the head of the waits-for relationship is of the form $<\overline{M}, o, O>$ or $<any, o, O>$. For example, Figure 9(a) shows a deadlock because the transaction group has only two members *M1* and *M2*. If a new member *M3* joins the transaction group there is not a deadlock, as shown in Figure 9(b). Unfortunately, adding a new member is not the sort of thing that can be done arbitrarily within the transaction group; new members must join as they are needed and able to work on the transaction group's task.

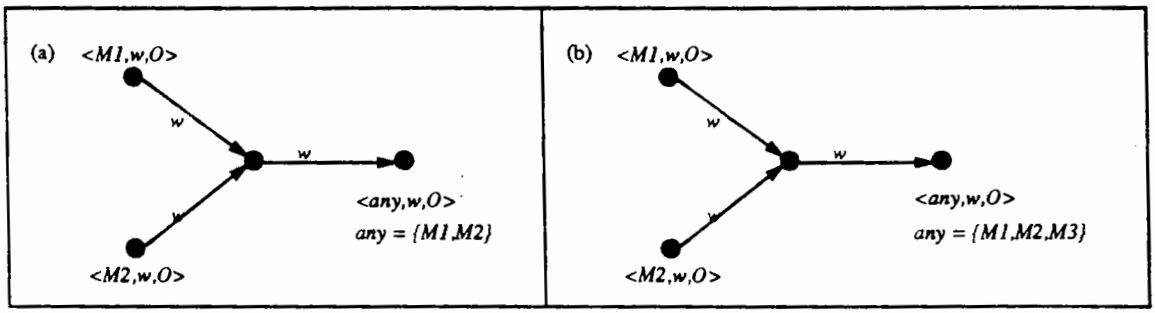

Figure 9: Effects of adding a new member; (a) Deadlock, (b) Not a deadlock.

5.6 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. Intention requests may propagate up the transaction hierarchy, but they never propagate down it. An intention at one level in the transaction hierarchy may be delayed because it has been propagated up the hierarchy, and is deadlocked in one of its ancestor transaction groups. However, that deadlock must involve only intentions within that ancestor transaction group. In fact, any deadlock is resolved within the transaction group that is the least common ancestor of the members that requested the intentions involved in the deadlock. Therefore, deadlock resolution within the ancestor group is sufficient to resolve the problems with the intentions in any of its descendants.

5.7 Determining Deadlock Freedom

Deadlock freedom means that, instead of detecting deadlocks as transactions are executed, you check things beforehand to see whether deadlocks can occur, and only allow machines to be defined and added to the correctness specification of a transaction group when they can cause no deadlocks.

Yannakakis [Yan82] has examined basic transaction systems and with the following result:

Theorem 3 (Yannakakis) *"It is NP-complete to decide whether a set of two-phase transactions is not deadlock-free."*

The proof is by reduction from a restricted case of 3-SAT where each variable ℓ_i occurs twice, and the negation $\bar{\ell}_i$ occurs once.

We will show that we can emulate two-phase transactions with our operation machine based correctness criterion, and thus two-phase transactions are a subset of cooperative transaction hierarchies. Based on Yannakakis' result, we then know:

Theorem 4 *Determining deadlock freedom in a transaction group is NP-hard.*

Two-phase locked transactions are a subset of cooperative transaction hierarchies. We will show this by emulate a two-phase locked database DB_{2PL} using a two-level cooperative transaction hierarchy TH and operation machines. In TH, the root transaction group has a correctness specification that emulates two-phase locking in its members. Each transaction T_i in DB_{2PL} is represented by some member M_i of the *Root* transaction group in TH, and that member accesses the objects in the database in the same order and with the same operations as T_i does.

For the purposes of this proof, we avoid recovery issues by assuming that all transactions commit.

First, we define a read-write-lock machine template as shown in Figure 10. The upper middle state represents a "write lock", and the lower middle state a "read lock". The only machines defined for the correctness specification of the transaction group are those instantiated from the read-write-lock machine template. The members never explicitly declare intentions, either. The only time intentions occur is when an operation is queued. In that case, the member blocks until the queued intention is resolved.

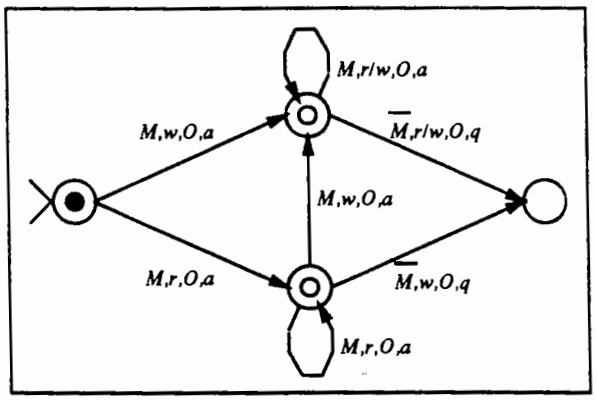

Figure 10: Lock machines for two-phase operation; (a) Read, (b) Write.

When a member M_i requests its first operation on an object O_j, the read-write-lock machine template is instantiated, with M bound to the member's TID (M_i) and O bound to the object ID O_j. If M_i attempts a read operation, we do one of the following:

1. If no other machine is bound to O_j, accept the read operation and update the machine.
2. If other machines are bound to O_j, but all are in the start state or the "read lock" state, accept the read operation and update the machine.
3. If other machines are bound to O_j, and one is in the "write lock" state (with all others in the start state), queue an intention for $<M_i, r, O_j>$.

If M_i attempts a write operation, do the following:

1. If there is another machine bound to O_j and some other member M_k in either of the middle two states, queue an intention to do the operation.
2. If there is no other machine bound to O_j, accept the write operation and update the machine.

The operations by a member are governed by the collection of active operation machines in the *Root* (and only) transaction group in *TH*. The machines associated with a member are removed when the transaction commits. Removing a machine corresponds to releasing the lock that that machine is emulating.

We show an example of some of the operation of the emulation of DB_{2PL} in Figure 11. Here, we have three transactions T_1. T_2, and T_3, emulated by *M1*, *M2*, and *M3*, respectively. In the two-phase locked database, the following operations occur:

1. T_1 gets a read lock on object O and reads it.
2. T_2 gets a read lock on object O and reads it.
3. T_1 tries upgrade its lock on O and fails because of T_2's read lock. The request is queued.
4. T_2 commits, and T_1 upgrades its read lock on O to a write lock. T_1 then does its write operation.
5. T_3 tries to get a read lock on O, but the request is queued until T_1 commits. Then T_3 gets its read lock on O and reads the object.

The emulation works as follows: Member *M1* is emulating transaction T_1, *M2* is emulating T_2, and *M3* is emulating T_3. When *M1* does its initial read in step 1, a read-write-lock machine is instantiated and the read operation is done, leaving the machine in the lower middle state, as shown in Figure 11(a). In step 2, since the read by *M2* is allowed, a read-write-lock machine is also created for that interaction. We now have two machines bound to O, as shown in Figure 11(b). The write operation in step 3 is queued because of the second machine, and *M1* blocks. Once *M2* commits (checkpoints its read and terminates), its machine is

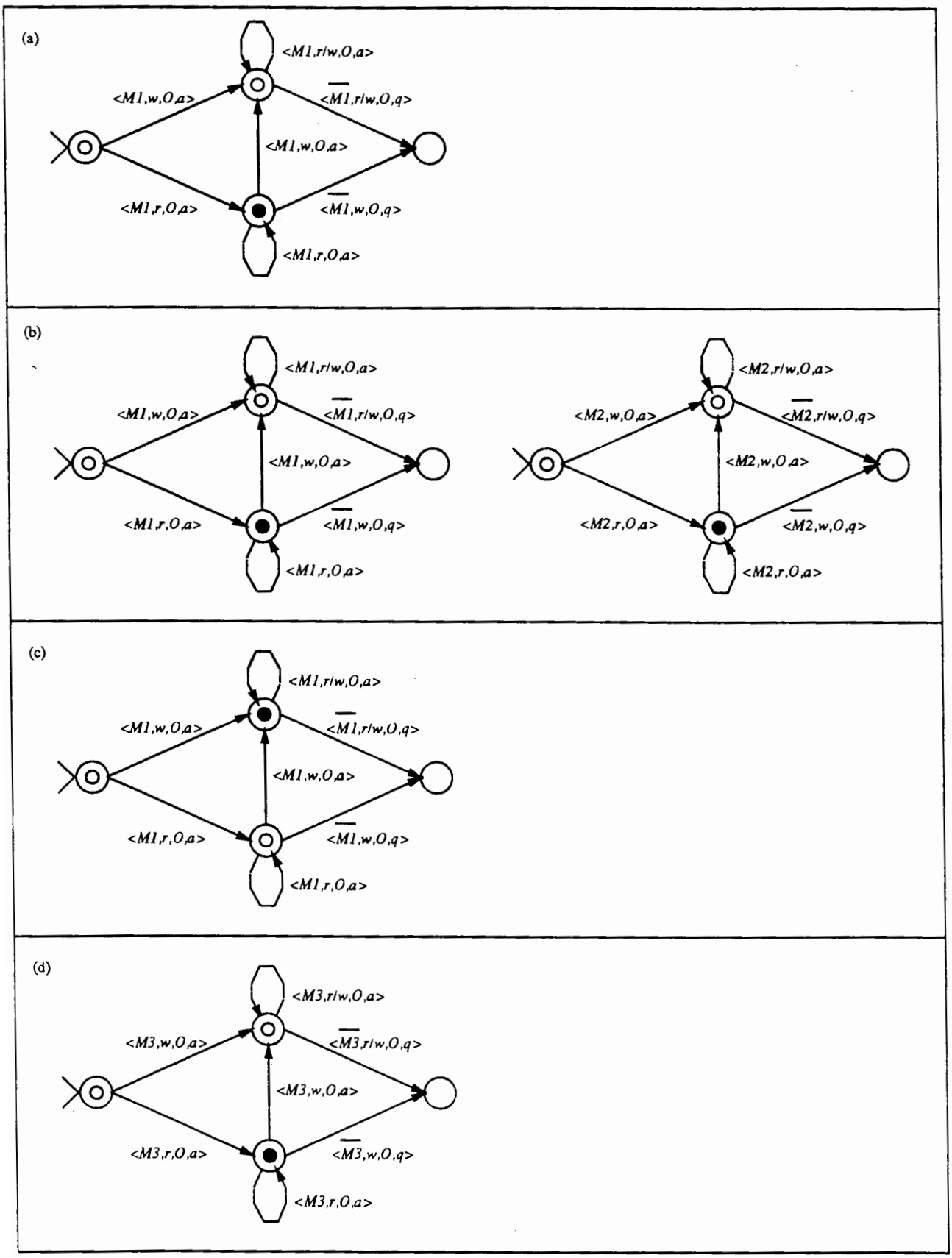

Figure 11: Example emulation of a two-phase-locked database.

removed. *M1*'s intention to write is then accepted, the operation is reissued, and the first machine transitions to the upper middle state. This is shown in Figure 11(c). The first machine now returns q when *M3* requests to read O in step 4, so *M3* blocks itself. In step 5, *M3*'s intention is accepted after *M1* checkpoints its operations and terminates, and *M3* reissues its read request. A new read-write-lock machine is set up and updated appropriately as shown in Figure 11(d).

Because we have shown that two-phase transactions are a subset of cooperative transaction hierarchies, we know that deadlock detection in cooperative transaction hierarchies is NP-hard. □

6 Summary and Related Work

We have defined a new transaction model for design transactions, called *cooperative transaction hierarchies*. Cooperative transaction hierarchies are not serializable, but rather rely on *patterns* and *conflicts* to specify orderings of operations that must and cannot appear in the history, respectively. The synchronization mechanism in the cooperative transaction hierarchy uses these patterns and conflicts to examine operations as they are requested by the members of the cooperative transaction hierarchy, and to determine whether the operation can be executed immediately, or whether it conflicts.

Cooperative transaction hierarchies do not use locks, but instead rely on conflicts and intentions to prevent unwanted interleaving of operations by the members in the database. Since conflict occurs among operations in cooperative transactions, conflict is resolved by either refusing or delaying operations as specified by a transaction group's patterns and conflicts. To prevent an operation from conflicting at the time it is requested, a member can predeclare an *intention* to do the operation single operation. This intention may be queued. Once a queued intention is eventually accepted, the synchronization mechanism guarantees that the corresponding operation will be accepted immediately.

When an operation cannot be executed or an intention cannot be granted because the synchronization method returns *queue*, an *intention* is queued for the operation, which will be accepted or refused once the operation would be accepted or refused. Operations are not queued directly, but instead an intention is queued for the operation so that the member that submitted the operation can re-evaluate it later when the operation can be executed.

When queueing is used during a scheduling process, deadlocks can occur as well. Deadlocks and deadlock detection have been discussed extensively for traditional databases. A good overview is provided in [BHG87]. We have adapted a waits-for graph approach to deadlock detection within a transaction group. We have given some pointers for deadlock resolution. We have also shown that deadlock detection and resolution within the individual transaction groups in a hierarchy is sufficient to ensure that all deadlocks in the hierarchy are resolved.

Deadlock freedom states that a specific set of transactions will never deadlock. We have shown that deadlock freedom in a cooperative transaction hierarchy is NP-hard. This proof is based on Yannakakis' theorem [Yan82] that deadlock freedom in a 2PL database is coNP-complete, and the fact that 2PL transactions are a subset of cooperative transactions.

References

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

[NFSZ90] Marian H. Nodine, Mary F. Fernandez, Andrea H. Skarra, and Stanley B. Zdonik. Cooperative transaction hierarchies. Technical Report CS-90-03, Department of Computer Science, Brown University, February 1990. Revised February, 1991.

[NRZ91] Marian H. Nodine, Sridhar Ramaswamy, and Stanley B. Zdonik. A cooperative transaction model for design databases. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan-Kauffman, 1991. to appear.

[NSZ90] Marian H. Nodine, Andrea H. Skarra, and Stanley B. Zdonik. Synchronization and recovery in cooperative transactions. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice*, pages 329–342. Morgan Kaufmann, 1990. also Proceedings of the 4th International Workshop on Persistent Object Systems.

[NZ90] Marian H. Nodine and Stanley B. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In *VLDB Proceedings*, pages 83–94, 1990.

[Ska91] Andrea Skarra. Localized correctness specifications for cooperating transactions in an object-oriented database. *Office Knowledge Engineering*, 4(1), 1991.

[Yan82] Mihalis Yannakakis. Freedom from deadlock of safe locking policies. *SIAM Journal on Computing*, 11:391–408, May 1982.