Long Atomic Computations

by

Pui Ng

August 1986

This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract number N00014-83-K-0125.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Long Atomic Computations

by

Pui Ng

Submitted to the
Department of Electrical Engineering and Computer Science
on August 26, 1986 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

Distributed computing systems are becoming commonplace and offer interesting opportunities for new applications. In a practical system, the problems of synchronizing concurrent computations and recovering from failures must be dealt with effectively. Atomicity has been suggested as a tool that masks concurrency and failures from the users of a system. With synchronization and recovery mechanisms, atomic computations appear to execute indivisibly. This dissertation addresses the issues in implementing *long* atomic computations, such as computations that last for hours or even days. Long computations make synchronization more difficult because their execution is more overlapped. They are also more likely to encounter failures in their execution.

Three issues are raised:

1. Should long computations be executed atomically? Or should atomicity be replaced with other correctness criteria to increase the concurrency of a system?
2. If long atomic computations can be implemented practically, are there implementation paradigms that application programmers can follow to simplify the programming effort?
3. How can long atomic computations be made resilient to transient failures?

This dissertation shows that by using the semantics of an application, a system that supports atomic computations can be made as concurrent as other systems that do not. Since atomicity is easier to understand than other correctness criteria, systems that support long atomic computations are preferable.

Using the semantics of an application requires application-dependent synchronization and recovery code, which can be complicated and introduce subtle

errors easily. Several synchronization and recovery paradigms are investigated in this dissertation. The paradigms divide synchronization and recovery into levels so that the task at each level is simpler. A programming interface that hides the concurrency control algorithm used by a system implementation is also presented.

Finally, this dissertation discusses the use of checkpoints and buffered messages to increase the resilience of long atomic computations.

Thesis Supervisor: David D. Clark
Title: Senior Research Scientist

Keywords: Distributed Systems, Atomicity, Concurrency Control, Long Computations, Recovery, Fault Tolerance, Reliability, Programming Methodology.

Acknowledgments

First I would like to thank my advisor, Dave Clark, for his guidance. He always provided me with fresh insights and imparted his enthusiasm. In addition, I would like to thank my readers, Dave Gifford and Bill Weihl, for their patience and suggestions that make the thesis more readable. Bill also offered many detailed comments, which forced me to think through many issues more thoroughly.

Numerous people on the fifth floor provided both technical and emotional support. In particular, Jim Gibson, Brian Oki, and Lixia Zhang read drafts of my thesis and gave useful suggestions. There are many others that showed their friendship and concern through their words of encouragement. I thank them all.

Many brothers and sisters in my church had in a sense written this thesis together with me. I cannot say enough to thank their support in prayer and in fellowship. Their support started with my looking for a thesis topic and kept me going.

I thank Elaine for her encouragement, her love, and her patience with me.

Finally, my family members, especially my parents, had shown their unceasing love and faith throughout the many years spent in writing this thesis. I would like to express my deepest appreciation.

May all the glory be to God.

Table of Contents

Table of Figures

Chapter One

Introduction

Distributed systems have become a reality with the increasing employment of workstations, home computers, and different types of computer communications equipment. Distributed computing has offered many opportunities to build new types of applications. These applications are characterized by activities that span multiple sites of a distributed system. For example, a travel agent may make several reservations in different airline, hotel, and car rental reservation systems. A bank customer may withdraw money from his account over a geographically distributed banking network. An employee in an office may schedule a meeting with several of his colleagues using a calendar system that runs on multiple workstations and portable computers.

However, as the number of sites connected in a distributed system grows, it also becomes increasingly likely that some components of the system are broken at any given time. Furthermore, the job of synchronizing concurrent activities becomes more difficult. It is unrealistic to use any centralized scheduler when many users may be initiating activities at the same time.

Atomicity [17, 28] has been suggested as a useful tool that alleviates these *synchronization* and *reliability* problems. Under the atomicity model, the activities in a distributed system are modelled as a collection of *atomic computations*. A computation is a unit of work initiated by a user or by the system itself. Atomic computations are computations that appear to execute serially in a certain *serialization order*. This *serializability* property frees the programmer from worrying about concurrent computations interleaving with one another. In addition to the serial behavior, an atomic computation is either *committed* or *aborted*. The effects of

a committed computation become visible to all computations executed subsequently. The effects are also *permanent* so that they are not lost with transient failures such as power outages. When a computation is aborted, any work performed by the computation is undone and the computation appears never to have executed. This all-or-nothing property is called *failure atomicity*. It lessens the burden on application programmers by undoing computations that are partially done.

In this dissertation we consider how *long* atomic computations can be supported in a distributed system. As the size of a distributed system becomes larger, it is inevitable that the lengths of computations also increase. With a large system, it is unrealistic to expect every component to be highly reliable given the high cost of such components. As a result, communication delays, network partitions, and unavailability of critical resources due to site crashes, are just some of the reasons why computations may execute for a long time. In fact, long computations can be created simply because there is much work to be done as a single unit, or because a computation requires human interaction. Consequently, long computations are not limited to distributed systems.

The increase in computation lengths exacerbates the synchronization and reliability problems. As each computation executes for a longer period of time, there is more overlapping of execution, which increases the likelihood of some of the computations being delayed. It also becomes more likely to encounter a failure during the execution of a long computation. Current distributed systems supporting atomic computations [31, 56] do not provide adequate support to long atomic computations. These systems do not provide any facilities for a computation to make its intermediate state resilient to transient failures. Also, because of an implicit model of short computations, it is considered acceptable to delay one computation pending the completion of another. In a system with long computations, such delays are usually unacceptable.

The rest of this chapter is divided in the following way. Section 1.1 describes our

definition of long computation more carefully and gives examples of such computations. Section 1.2 discusses the major problems in supporting long atomic computations. Section 1.3 summarizes our solutions and contributions toward solving these problems. Section 1.4 presents a roadmap for the thesis. Section 1.5 gives an overview of related work.

1.1 Long Atomic Computations

A computation may execute for a long time because of extensive computing or waiting for I/O events (e.g., waiting for input from keyboard or network). For example, a computation that requires human interaction can last for minutes or even hours. Clearly, the length of a computation is a relative measure. Instead of using an absolute numerical definition for long computations, we concentrate on computations that may require special support due to their length. Whether such support is needed depends on the length of computations *and* on the characteristics of the system on which they are executed. For example, the concurrency control algorithms, the system usage characteristics, and the mean-time-between-failure characteristics of the hardware are some of the factors that affect the response time and resilience of a system. In a typical distributed system that supports atomic computations [31, 56], computations that last hours or days can be considered long because they are prone to be aborted and induce long delays in concurrent computations. Shorter computations that last minutes or even seconds can also be considered long if the hardware is unreliable or the system is heavily used.

In our discussions we will focus on long computations whose lengths can be attributed to long delays in network communication. Several factors can contribute to these long delays:

- mobility of sites, such as disconnection of portable computers,
- unreliable links in the network causing partitions,
- slow links or switches,
- economic reasons: sending messages batched is less expensive,
- security that is enforced by isolation.

We believe that our work is also applicable to other types of long computations because of the similarity of the problems encountered in supporting them.

Many applications require long computations. For example, a computation that schedules a meeting among several personal calendar servers can last for hours or days because some of the calendars reside on portable computers and are disconnected from the system. A replicated database [11] may propagate the updates to a replicated data object over a long period of time. A computation making several airline, hotel, and car rental reservations may last too long compared to the concurrency requirements of an airline reservation system.

1.2 Concurrency and Resilience Problems

In the previous section we alluded to a *concurrency* problem and a *resilience* problem with long atomic computations. Intuitively, a system is bound to create a concurrency problem when it is trying to maintain an image of substantially overlapped computations executing serially. A resilience problem is also to be expected because it is more likely to encounter a transient failure in the execution of a long computation than in a short computation. This section describes these problems more concretely by describing how some systems [31, 48, 56] implement atomicity. We argue that a long atomic computation causes long delays in concurrent computations and is prone to be aborted in these implementations.

1.2.1 Concurrency Problem

In most earlier work [46, 40, 48, 26, 7], a (distributed) system is modelled as a collection of *objects* with read/write *operations*. A computation is modelled as a sequence of read/write operations on the objects accessed by the computation. In order to guarantee serializability and failure atomicity of atomic computations, each object is implemented to behave "atomically:" the values returned by the read operations should be identical to those returned had the committed computations been executed in some serial order common to all the objects.

In general, two different types of algorithms are used to ensure such atomic behavior. In a locking algorithm, an object is associated with a *read/write lock* [31, 56, 17]. A read (write) lock is acquired before a read (write) operation is executed. Two locks conflict with each other unless they are both read locks. When a computation requests a lock, it is delayed until all other computations that had previously acquired conflicting locks are completed. This locking algorithm is called *2-phase locking* [17]. In a timestamp algorithm, computations are assigned timestamps when they are started [48]. A computation is aborted and restarted if it tries to write an object that had already been read by another computation with a larger timestamp. If a computation with a timestamp t tries to read an object, it is delayed until the computation that has the largest, yet smaller than t, timestamp among all the computations that had written that object is committed.

When long computations are executed, neither type of algorithm results in a satisfactory level of concurrency. In the locking algorithm, a long computation causes other computations that attempt to acquire conflicting locks to be delayed until it is completed. Worse yet, computations can be deadlocked with one another, so that one of them has to be aborted. When a deadlock occurs, there is the cost of detection, which usually involves passing messages among sites [43], and the cost of restarting the computation. Although there is no empirical data on the frequency of deadlocks in a system with long computations, one can expect deadlocks to be more frequent than in a system with only short computations, as locks are held for longer periods of time.

The long delays caused by incomplete computations are also possible in a timestamp algorithm. In addition, a long computation can be aborted due to other computations with larger timestamps reading the objects that it is going to write. Normally, to make sure that computations are serialized approximately in the order that they are invoked, timestamps are assigned from real-time clocks. Consequently, a computation becomes more likely to be aborted when it gets longer, because more computations are started while it is being executed.

The following example illustrates the concurrency problem. Consider a personal calendar application that consists of many personal calendars, each owned by a different user. Each user can read his own calendar (*read_calendar*), reserve a time slot in his calendar (*mark*), and un-reserve a time slot (*delete*). *Read_calendar* returns a list of slots, some of which are reserved by previous *mark* operations. The *mark* operation can return *okay* or *slot_filled* depending on whether the proposed slot has already been reserved. *Delete* un-reserves a slot and returns *okay* if the user is permitted to do so. Otherwise *cannot_delete* is returned. All of these operations, except *read_calendar*, require updating a calendar. On top of these operations, the calendar application can construct computations that set up a meeting among several calendars (*set_up_meeting*), or computations that cancel a meeting (*cancel_meeting*). For example, *set_up_meeting* would invoke a *mark* operation at each of the calendars involved.

A word of terminology is needed before we proceed with the example. In this thesis, a computation is modelled as a sequence of *operations* on some *objects*. It should be emphasized that these objects are *abstract* objects supporting *abstract* operations, such as the calendar object described above. A simplified view of the system is to regard each operation, such as a *mark* operation, as relatively short, while a computation, such as a *set_up_meeting* computation, spends most of its time delivering messages across a network to invoke operations.

A computation that involves multiple calendars may span a long period of time because some of the computers involved may be disconnected from the system either physically (because they are portable) or functionally (because they are not running the calendar software). *Set_up_meeting* and *cancel_meeting* computations belong to this category.

Obviously, if each calendar object is implemented with a single read/write lock, the level of concurrency would be unacceptably low. For example, it is unacceptable to render all the calendars of a meeting's participants inaccessible until a disconnected

participant is reconnected to complete a *set_up_meeting* computation. The timestamp algorithm has similar problems. We will omit it in the discussion below unless it offers interesting alternatives to the locking algorithm.

Concurrency can be increased by dividing a calendar into slots and associating a read/write lock with each slot. However, the concurrency of the implementation may still be unacceptably low. For example, consider the situation in which the owner of a calendar is trying to read his calendar when the calendar is the participant of an incomplete *set_up_meeting* computation. Following the read/write lock algorithm, the *read_calendar* operation will be delayed until the *set_up_meeting* computation is completed. This is clearly unacceptable.

One may argue that a timestamp algorithm offers a solution in this situation. By choosing a smaller timestamp for the computation that invokes *read_calendar* than that of the *set_up_meeting* computation, the *read_calendar* operation can return the state of the calendar *before* the *set_up_meeting* computation is executed. However, this solution is not without its problems. Suppose the owner of the calendar decides to reserve the slot occupied by the *set_up_meeting* computation for some other purpose. The request cannot be accepted because the slot had already been promised to the *set_up_meeting* computation, albeit tentatively[1]. On the other hand, the request cannot be delayed or rejected either because an inconsistent picture will be presented: by observing the state of the calendar before the *set_up_meeting* computation is executed, the user is led to believe that the slot is empty and expects the request to readily succeed.

One may consider this example as an argument against having long atomic computations. Arguing intuitively, we cannot expect an implementation to hide the

---

[1]Depending on how different sites of a distributed computation decide whether the computation should be committed or aborted, a site may be able to abort an incomplete computation unilaterally [17]. However, there is also a window of vulnerability in which such unilateral aborts are not allowed. This window can span a long period of time if communication delays are long. In any case, it is rather counterproductive to abort any incomplete *set_up_meeting* computations whenever a calendar is read.

fact that there are multiple users using the system in substantially overlapped periods of time. Hence, atomicity may have to be replaced with some other correctness criterion. One of the contributions of this thesis is to show how atomicity can be employed even with long computations. Section 1.3 will describe how atomicity can be used in conjunction with non-determinism to solve the concurrency problem. Since atomicity is not abandoned, the simplicity offered by atomicity is preserved.

In conclusion, the concurrency problem is caused by the uncertainty of whether an incomplete computation would eventually commit, and also the requirement that computations should appear to execute serially when in fact they are invoked concurrently. The problem is more serious in a system with long computations because long computations take a long time to complete and overlap substantially.

1.2.2 Resilience Problem

In addition to the concurrency problem, one also needs to deal with a resilience problem in implementing long atomic computations. For a system with long computations, the failure atomicity requirement is both a blessing and a curse. On the one hand, the increased likelihood that a long computation would encounter some transient failure[2] heightens our need for recovery mechanisms. Failure atomicity provides a simple interface to the application users because a computation is executed either in entirety or not at all. On the other hand, satisfying failure atomicity requires aborting computations interrupted by transient failures unless sufficient intermediate state of the computations has been saved. Some systems preserve the intermediate state of a computation through the use of replicated processors and memories [3, 13]. However, these systems require a degree of replication that may be too expensive for many applications.

When a long computation is aborted, potentially much time and work can be wasted. The decomposition of a computation into a nested tree of *actions* [40, 48] provides a

[2]Sources of transient failure include site crashes and deadlocks.

partial solution: an action can be aborted without undoing the effects of its sibling and ancestor actions. It is inadequate since actions near the top of the tree are still vulnerable. Transient failures that happen while these actions are waiting for their descendant actions to complete can cause most of the action tree to be aborted. For example, a *set_up_meeting* computation can be implemented with a parent action at the originator of the meeting, which creates a child action at each of the participant calendars. Although the computation is insulated from transient failures at the participants, it is still vulnerable to failures at the originator site. We will describe the nested action model in greater detail in Chapter 2.

1.3 Contributions and Solutions

Collectively, our contributions can be viewed as an argument for the feasibility of long atomic computations. More specifically, they can be viewed as solutions to the concurrency and resilience problems. We will start with an enumeration of our major contributions, then we will give a more detailed summary of the solutions presented in this thesis.

There are four major contributions in this thesis:

1. We show that an application can trade off functionality for more concurrency. By functionality we refer to the behavior of the application when computations are executed serially in an environment without failures. Our approach, like other proposals [1, 25, 38, 50, 51], uses application semantics to increase concurrency. However, our approach, similar to [33], goes a step further and raises the possibility of "*decreasing*" functionality to increase concurrency. The decrease in functionality is achieved by introducing non-determinism. *Our contribution is to show that this approach of decreasing functionality while maintaining atomicity is as "powerful" as other correctness definitions that have abandoned atomicity*, such as the *input consistency* criterion described in [50]. We will show that the exact gain in concurrency through the use of these other correctness definitions can be realized through decreasing the functionality of the application. On this basis, we will claim that our atomicity definition is preferable, since in comparison it is equally powerful and easier to understand.

2. *Our second contribution is the development of a* ***conflict model*** *that allows the programmer to determine an approximation of the level of concurrency achievable with a particular functionality of an application.* The level of concurrency is expressed as conditions under which *conflicts* occur. A conflict is created when an implementation is uncertain of how computations are serialized or whether a computation will eventually commit. When conflicts occur, computations are either delayed or restarted, depending on how the serialization order is determined. The model is useful in that it abstracts away the details of how to deal with a conflict and how the serialization order is determined. For example, the programmer can design the functionality of an application without worrying about whether a timestamp or locking algorithm is used.

3. Our third contribution relates to the study of *concurrency control algorithms*, which determine the actions that are taken when conflicts arise and how a serialization order is determined. Although the concurrency of an application is significantly influenced by its functionality, we argue that the concurrency control algorithm still has an effect on the overall level of concurrency of an implementation. For example, the cost of a conflict is relatively insignificant if it causes a long computation to be delayed until the completion of a short computation. The same is not true if the situation is reversed. *Our contribution lies in the design of novel concurrency control algorithms that can substantially reduce costly conflicts under certain conditions.*

4. Finally, this thesis also discusses how applications can be implemented such that the concurrency of the implementations would improve with the relaxation of the application functionality. *Our contribution is the design of a programming interface that allows application semantics to be utilized without exposing the concurrency control algorithm underneath.* Our programming interface allows a programmer to write programs for systems using different concurrency control algorithms without having to be familiar with all of the algorithms. The programs are also portable so that no modifications are necessary when the underlying concurrency control algorithm is changed.

Having enumerated the major contributions, we now proceed to give a more detailed description of the solutions to the concurrency and resilience problems proposed in this thesis.

1.3.1 Functionality - Concurrency Trade-Off

Consider the *read_calendar* operation discussed in section 1.2 again. Although we have described the concurrency problem using the locking and timestamp algorithms, the problem lies in fact in the functionality of the operation. The problem exists regardless of how atomicity is implemented. The functionality of the *read_calendar* operation that we described in section 1.2 is to present an up-to-date view of the state of a calendar. In addition, we also require the view to be accurate such that it reflects only committed computations. This is clearly unachievable given that a *set_up_meeting* computation had visited the calendar and the calendar has no knowledge as to whether the computation will be committed eventually. An implementation must either risk presenting an inaccurate view or choose an outdated one.

The solution that we propose in this thesis is not to abandon atomicity, but rather, to change the functionality of the *read_calendar* operation. For example, one can incorporate non-determinism in the functionality of the *read_calendar* operation such that the set of reserved slots in the list of slots returned is required to be only a *superset* of the set of reserved slots in the accurate view. By allowing non-determinism in the result returned by *read_calendar*, *read_calendar* does not have to be delayed until all incomplete *set_up_meeting* computations are completed. *Read_calendar* can simply return all the slots reserved by incomplete or committed computations as reserved. The result returned by *read_calendar* is acceptable even if some of the incomplete computations turn out to be aborted later. The semantics of *read_calendar* does not require the result to contain only committed slots. We will define atomicity such that it allows a non-deterministic functionality of an application to be incorporated in the definition. Liskov et al. proposed the same solution in [33].

Our example can also illustrate why atomicity, coupled with the functionality of the applications, is as powerful as some other correctness definitions. For example, consider an alternative in which *set_up_meeting* is implemented as a collection of atomic computations [15], one at each participant calendar of the meeting. If

set_up_meeting is to be abandoned, compensating atomic computations can be executed at each of the participants already visited. Concurrency is not a problem in this implementation because each of the atomic computations is short. Interestingly, the behavior of this implementation is the same as the one with the relaxed functionality of *read_calendar* described above. Because *set_up_meeting* is implemented as a collection of atomic computations, the atomic computation that executes *read_calendar* can be serialized between the atomic computation that reserves the slot for the meeting and a later compensating atomic computation. The result returned by *read_calendar* is just as up-to-date and tentative as that implied by the relaxed functionality. The difference is that our approach provides an abstract specification of the behavior of the implementation, defined by atomicity and the relaxed functionality of the application. The abstract specification allows the users of the application to understand the behavior of the implementation more easily.

1.3.2 Implementation Paradigms

Relaxing the functionality of the application by itself is not sufficient to solve the concurrency problem. For example, if an implementation of the calendar application uses read/write locks, relaxing the functionality of *read_calendar* does not change the fact that a *read_calendar* operation trying to acquire the read lock would still be delayed by a *set_up_meeting* computation that is holding a write lock. In this thesis, we are also interested in how an application can be *implemented* such that the relaxed functionality of an application can be utilized. To provide a summary of our programming paradigms, we will describe how System R, a relational database management system that supports atomic computations [18], increases its concurrency with the semantics of its index objects. We will draw analogies between System R's approach and our paradigms.

There are two levels of objects in System R. At the upper level, there are *RSS objects*, such as an index to a relation. At the lower level, there are *page objects*. Invoking an operation on an index object involves accessing one or more page

objects. Accesses to page objects are synchronized with *page locks*, which can be viewed as read/write locks of the page objects. Because a page object that implements an index object may be accessed by many concurrent computations, locking a page for the entire duration of a computation is unacceptable. To increase concurrency, page locks are released at the end of an operation on an index object, instead of at the end of a computation. To preserve atomicity, an additional level of "logical locking" is implemented. Information about an index operation is recorded when the operation is executed. By examining the history of past index operations, "conflicting" index operations that may lead to non-atomic behavior, such as inserting and reading from the same key value, are delayed. Furthermore, because the relevant page locks have been released, aborting an index operation cannot be achieved by restoring the previous contents of the modified pages. Rather, a logical undo operation is invoked during recovery.

Our approach to implementing atomicity is similar to System R's in many ways. Moreover, we are interested in the following questions:

1. Can System R's approach of utilizing the semantics of an index object be applied to other kinds of application-level objects? In particular, can the programs that perform the "logical" synchronization and recovery be made easier to write and understand by following a general implementation paradigm?

2. Can a concurrency control algorithm akin to a timestamp algorithm, or some other hybrid algorithms [7], substitute for the locking protocol used in the page lock level or the logical locking level or both? Can a programming interface be designed such that an application programmer is not aware of the concurrency control algorithms used in the system implementation?

The rest of this section gives a summary of our answers to these questions.

1.3.2.1 Level Atomicity

Similar to System R's approach of implementing atomicity, ours also divides objects into multiple levels. This division is more than just a division of levels of abstraction. As will be described in this section, the division is a partitioning of the synchronization and recovery code of an implementation. For simplicity's sake, we will limit the discussion in this thesis to systems with only two levels. An object in the higher level is implemented using the objects in the lower level. For example, an index object is implemented using page objects.

To simplify the programs that access the higher-level objects, all the operations on the objects in the higher level are made to appear instantaneous to one another. For example, because of the page locks acquired by an index operation, index operations appear to be instantaneous to one another even though an index operation may access more than one page object. The logical locking in System R is simplified because index operations can be treated as instantaneous. The atomicity concept can be applied *again* to present this image of instantaneity. In other words, there are two kinds of atomic computations in our implementations. The first kind of atomic computations are the computations that we have been discussing in this chapter. They *access* the higher-level objects and can last a long time. In System R, they may be queries or updates to the database. The second kind of atomic computations are the computations used to *implement* the operations on the higher-level objects. They make the operations on the higher-level objects appear instantaneous and simplify the programming of the first kind of atomic computations. They are probably short. In System R, they last for the duration of an index operation. To distinguish the two kinds of atomic computations, we call the first kind *globally atomic computations* and the second *locally atomic computations* because we expect in most applications the second kind will execute within a single site. Figure 1-1 describes this paradigm of implementing globally atomic computations with locally atomic computations.

The locally atomic computations are atomic in the sense that they make operations

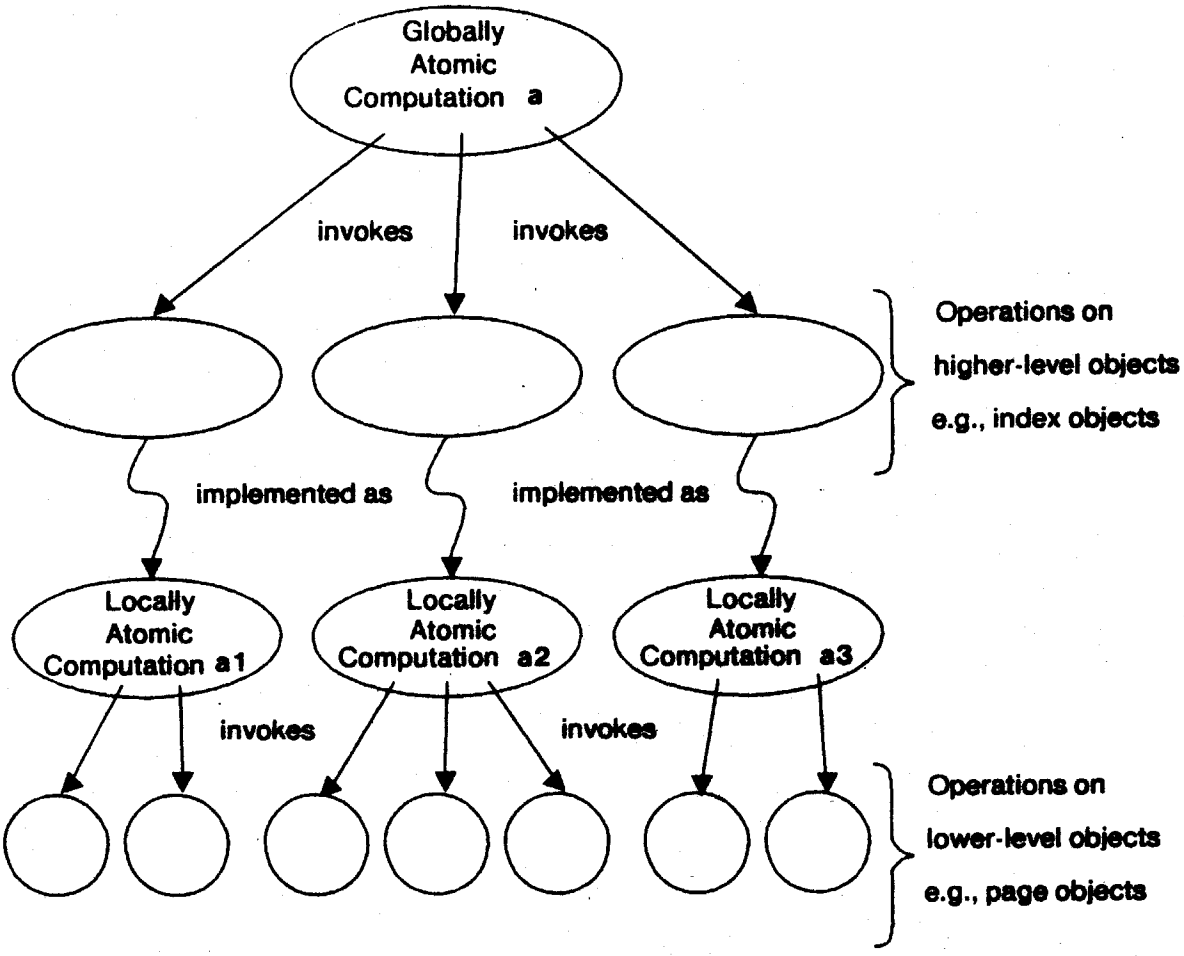

Figure 1-1:A Globally Atomic Computation Implemented with Locally Atomic Computations

on a higher-level object appear to be instantaneous to one another. On the other hand, they are *not* globally atomic in the sense that after one of these locally atomic computations (e.g., a1 in figure 1-1) is completed, its effects can be observed by other locally atomic computations even though the globally atomic computation that invokes it (e.g., a in figure 1-1) is not yet committed. For example, by releasing page locks at the end of an index operation o, changes made by o on the page objects can be observed by other index operations even when the globally atomic computation that invoked o is not yet committed.

Using the calendar example, each *mark* operation in a long globally atomic *set_up_meeting* computation can be executed as a short locally atomic computation. Obviously, we need the equivalent of the logical locks in System R to make sure that the collection of short locally atomic computations would appear to be a long globally atomic computation. For example, a *read_calendar* operation must be prevented from observing the effects of a *mark* operation *if* the result returned by *read_calendar* is supposed to be accurate. This is because the *set_up_meeting* computation that invoked the *mark* operations may be aborted later. The subject of logical locking will be discussed in the next section.

By implementing operations on a higher-level object with locally atomic computations, the programs that invoke these operations can treat them as instantaneous regardless of the complexity of their implementations. The complexity of synchronization and recovery is reduced by dividing them into two levels. For example, synchronization is divided between the logical locks and the page locks in System R. In our calendar example, a *read_calendar* operation would never observe the state of a calendar with partially executed *mark* operations. We call this idea of implementing long globally atomic computations with short locally atomic computations *level atomicity*. A similar idea has been presented by Beeri in [5] and Moss et al. in [42] although their work is not motivated by long atomic computations. The difference between our work and theirs lies in the different approaches used to implement logical locking.

1.3.2.2 Conflict Model

In this section we briefly describe our solutions to the following two questions:

1. How can the logical locking in System R be extended to different kinds of abstract objects?

2. How can logical locking be extended to cover "logical timestamping?"

To answer these questions, we will generalize from the concurrency control algorithms synchronizing objects with only read/write operations. Examining the

timestamp and locking algorithms, we can identify three common components of these algorithms:

1. Determining how computations are serialized. It is determined by the order in which computations commit in a locking algorithm, and by the timestamp order in a timestamp algorithm.

2. Determining when a "conflict" arises. For example, in a locking algorithm, a conflict arises for a read operation when it tries to acquire a read lock and there is another incomplete computation holding a write lock. In a timestamp algorithm, a conflict arises for a write operation when there are previously executed read operations invoked by other computations with larger timestamps.

3. Determining the action to take when a conflict arises. In a locking algorithm, operations are delayed. In a timestamp algorithm, operations are either restarted or delayed.

Programming the logical locking needed for any abstract object can follow the pattern above. First, determining how computations are serialized can be achieved with the following:

1. a concurrency control algorithm similar to the locking and timestamp algorithms,
2. a programming interface from which an object implementation can determine the serialization order of the computations that had invoked operations on the object.

Second, when conflicts are created is application-dependent and depends on the functionality of an object. For example, whether a *read_calendar* operation creates a conflict depends on its functionality and, if it is required to return an accurate view of the calendar, whether there are incomplete *set_up_meeting* computations that may be serialized before it. In addition to capturing the serialization order, the programming interface that we described above should also capture the history of previously invoked operations and the status (e.g., incomplete, committed) of the computations that invoked them. In the next section we will describe such a programming interface. It allows an object implementation to express the conditions under which a conflict arises.

These conditions are expressed in such a way that they are insensitive to whether a locking or timestamp algorithm, or some other concurrency control algorithm, is used to determine the serialization order. For example, the condition under which a conflict arises for a write operation on a read/write object can be expressed as follows: previously executed read operations invoked by other computations *may be* serialized after this computation.

With a locking algorithm, this condition translates into the following condition: read locks are held by other computations. With a timestamp algorithm, the equivalent condition is: previously executed read operations invoked by other computations have larger timestamps. Similarly, the condition under which a conflict arises for a read operation is that there are previously executed write operations invoked by other computations that are not committed or aborted and *may be* serialized before this computation. Notice that we have hidden underneath these conditions the choice of how to determine the serialization order.

We will describe a process in which these conflict conditions can be systematically derived from the functionality of an abstract object. The conflict conditions provide an approximation of the level of concurrency that can be achieved with a certain functionality.

Finally, the action that needs to be taken when a conflict arises depends on how the serialization order is determined. For example, some algorithms require that an operation be delayed whereas other algorithms require the computation that creates a conflict to be restarted. Similar to the conflict conditions, these actions can be expressed without exposing the underlying concurrency control algorithm.

1.3.2.3 Programming Interface

To implement the conflict model that we have described above, we provide a programming interface that is characterized by the use of *history objects*. A history object captures the history of operations that had been executed in an abstract

object. Queries can be directed to the history object to determine whether a conflict condition is met. The interface of the history objects will make the underlying concurrency control algorithm transparent to the application programmers.

When a conflict arises, some of the actions that can be taken are delaying or restarting a computation that is involved in the conflict. Again, these actions can be made transparent to the programmer and expressed in the programming interface as a generic `resolve conflict` statement.

We will also discuss how recovery can be performed in our programming interface. For example, if the execution of an operation changes only the state of a history object, aborting a computation can be achieved by simply undoing the changes in the history object. This is a simple action and can be automated easily.

1.3.2.4 Concurrency Control Algorithms

Although we have provided a programming interface so that the programmer is unaware of the underlying control concurrency algorithm, the system implementation has to make a choice among the available options. The system implementation should also provide the necessary translation from the programming interface to the option chosen.

We have argued that in some applications the concurrency problem can only be solved by changing the functionality of the application. It remains to be seen whether the choice of the concurrency control algorithm affects the concurrency of a system with long atomic computations significantly. We will argue that in some cases it does make a difference. We will present some novel algorithms that minimize the likelihood that costly conflicts will arise. For example, the cost of restarting a short computation is much smaller than restarting a long computation. Consequently, an algorithm that makes restarting long computations less likely provides a higher level of concurrency.

1.3.3 Resilience Problem and Its Solutions

To increase the resilience of long computations, we propose a *checkpoint* mechanism and the use of *relay message servers*. Each checkpoint specifies some intermediate state of a computation; the state specified by the last checkpoint will be restored after a transient failure and the computation will be restarted from that checkpoint. In addition to limiting the effect of site crashes, checkpoints can also serve as fire walls to limit the rollback due to deadlocks. Relay message servers provide buffering and reliability when the network partitions frequently. Some other systems [10, 19] also use reliable communication primitives to simplify the implementation of distributed atomic computations. The relay message service in this thesis is easier to implement because it does not provide any guarantees on the order that messages are delivered.

1.4 Roadmap

Chapter 2 describes our model of system hardware and assumptions. In particular, we do not assume a reliable communication network in which messages are not lost and are delivered in a bounded time. We believe that implementing such a network is prohibitively expensive and any upper bounds on delivery times would be so large as to be useless. The hardware model is followed by a model of computation. Chapter 2 concludes with a more careful definition of atomicity.

Chapter 3 describes our conflict model and how functionality can be traded off for concurrency. Chapter 4 describes our programming paradigms and presents examples of application programs. Chapter 5 compares concurrency control algorithms and argues that some algorithms would have better performance with certain types of applications. We will also present two novel algorithms: a *hierarchical* algorithm and a *time-range* algorithm. These algorithms minimize the occurrences of costly conflicts under certain conditions. Chapter 6 shows that atomicity is as powerful as some other correctness definitions [50, 38] in which atomicity is abandoned and replaced with explicit descriptions of how computations

can interleave. In Chapter 7 we turn our attention to the resilience problem of long computations. We will describe a checkpoint mechanism and the use of relay message servers to buffer messages. Chapter 8 is the conclusion.

1.5 Related Work

In this section, we compare our work with related work on concurrency control and resilient computing. In our comparison of concurrency control, we focus on other systems that use application semantics to improve concurrency. Much work has been done in this area. Many proposals [e.g., 23, 24, 25, 5, 8] do not consider recovery issues and will not be covered in this section. Comparison with related work can also be found in the rest of this thesis as we describe more details of our proposal.

1.5.1 Predicate Locks

Eswaran et al. [14] describe the use of *predicate locks* for a relational database management system. An operation must acquire a predicate lock before it can proceed. Two predicate locks conflict if a tuple in a relation satisfies both predicates. Other than assuming a locking algorithm, the predicate locks differ from our conflict conditions in that the unit of concurrency is limited to a tuple. For example, using predicate locks does not solve the concurrency problem of our calendar application if each slot is implemented as a tuple. There is also no obvious way in which a slot can be broken into smaller units to increase concurrency.

1.5.2 Schwarz's Thesis

Schwarz [50] defines correctness as the acyclicity of computations with respect to a set of dependency relations. A dependency between two computations is formed if they each execute an operation at the same object. Correctness requires that the dependency graph be acyclic. The dependency relations are parameterized by the type of the operations invoked and the value of the arguments. Dependency

relations are "insignificant" and ignored in the dependency graph if the two operations involved in the dependency commute. Serializability is viewed as a special case in a range of possible correctness definitions with only the insignificant dependency relations ignored. Less restrictive correctness definitions can be obtained by leaving out "significant" dependency relations in the dependency graph.

The limitation of this approach is that the commutativity of two operations depends on many factors usually. It depends not only on the types of the operations and their arguments, but also on the history of operations invoked previously and the results returned by operations. For example, whether an operation to withdraw money from a bank account commutes with a previous withdraw operation depends on the balance of the account and the responses to these withdraw operations (*insufficient_funds* or *okay*). Whether an operation can proceed cannot in general be determined by *pairwise* dependencies with previously invoked operations. In other words, the limitation of Schwarz's approach is due to a *static* specification of the set of dependency relations included in the dependency graph.

1.5.3 Allchin's Thesis

Allchin [2] describes several different mechanisms to synchronize concurrent computations. One of them uses locks with user-defined lock modes. This approach is similar to Schwarz's and suffers from the same limitations. Allchin also suggests the use of a history mechanism similar to ours but tailored for a locking algorithm. Recovery is supported with *recoverable objects* that return to their initial values when a computation is aborted. The state of an implementation has to be carefully encoded with recoverable objects. In general, the changes made to a recoverable object by two computations will be lost if the computation that made the first changes is aborted. The recovery paradigms discussed in this thesis are different in that an application can invoke application-dependent recovery code explicitly. Two different recovery paradigms will be discussed in this thesis. One of them allows application-dependent code to be executed to perform state changes when a computation

commits. The other allows application-dependent code to be executed when a computation aborts.

1.5.4 Weihl's Thesis

Our atomicity definition follows the work of Weihl [55]. Weihl describes two types of objects called *atomic* and *mutex objects*. Mutex objects are in general locked for the duration of an operation whereas atomic objects are locked until the end of a computation. Two approaches, implicit and explicit, are suggested for synchronization and recovery.

In the implicit approach, synchronization is achieved by testing whether an atomic object was accessed by a still incomplete computation. Presumably the programmer can set up enough atomic objects to encode the history information needed for synchronization. For recovery, the programmer sets up the atomic objects so that when a computation aborts, its effects are nullified by the atomic objects reverting to their previous states. The effects of concurrent computations should not be undone in the process. In the explicit approach, objects are associated with undo records or intentions lists constructed explicitly by the programmer. The undo records or intentions lists can be examined to determine whether an operation can proceed. When a computation commits or aborts, the undo records or intentions lists are used to determine the state changes that need to be made.

In the implicit approach, it is unclear how other types of concurrency control algorithms can be employed because the lock testing of atomic objects exposes the underlying algorithm. Although the explicit approach does not exclude using other concurrency control algorithms, it does not provide an interface that makes the concurrency control algorithm transparent.

1.5.5 Garcia-Molina's Semantic Consistency

Garcia-Molina [38] describes a system in which computations are divided into steps and counter-steps. The counter-steps undo the previous steps if the computation is aborted. Two steps can proceed concurrently if they are "compatible" according to the *compatibility sets* of the computations to which they belong. A compatibility set is determined by the type of a computation and consists of sets of other types of computations that can interleave with this type. The limitation of the compatibility sets is similar to that of the dependency relations in Schwarz's thesis [50]. Since the compatibility sets are defined statically, there are a large number of applications in which two computations are defined to be incompatible because they are incompatible for a small class of situations. It is also unclear how an application programmer can describe the behavior of an implementation in a high-level abstract specification. The compatibility sets and counter-steps are rather implementation-oriented descriptions of the behavior.

1.5.6 Montgomery's Thesis

Montgomery [39] describes the use of *polyvalues* to represent the values of data objects accessed by incomplete computations. Each polyvalue represents the possible values that the object may take on depending on the outcomes of the concurrent computations. It deals with the problem of failure atomicity but not serializability, because two computations can access two objects in different orders and both commit.

1.5.7 Gifford's Persistent Actions

Gifford and Donahue [15] describe executing a computation as a *persistent action*. A persistent action consists of atomic actions and other persistent actions. Atomic actions in [15] can be equated with the atomic computations in this thesis. The results returned by the component actions of a persistent action are logged in stable memory. When a persistent action is interrupted by a site crash, it is restarted from the beginning. When it invokes a component action that had its result logged, the

result can be reused instead of calling the component action again. Any non-idempotent operations, such as reading the time-of-the-day clock, have to be cast as component actions. The component actions of two persistent actions can interleave arbitrarily.

In the system described in [15], it is unclear how abstract specifications of the behavior of persistent actions can be provided. Another difference between our work and theirs is our emphasis on how application-dependent synchronization and recovery can be programmed.

Our approach to resilience is also different. Instead of requiring the operations executed in a persistent action to be either idempotent or cast as a component action, the operations executed by the atomic computations in this thesis can be non-deterministic. A careful structuring of idempotent actions is not necessary. Checkpoints are specified explicitly. Stable memory access is necessary only at checkpoints instead of whenever a component action returns.

1.5.8 Sha's Thesis

Sha [51] describes a system in which data objects are partitioned into *atomic data sets*. Consistency constraints in the system cannot span atomic data sets. A computation is called a *compound transaction*, which is subdivided into consistency-preserving *elementary transactions*. The elementary transactions are further subdivided into *atomic commit segments*, each of which accesses a different atomic data set. When an atomic commit segment is finished, locks acquired to assure serializability are released, but write locks are retained to guarantee failure atomicity. When an elementary transaction is finished, the write locks are released and recovery is achieved through compensating transactions.

The atomic data sets provide a relatively coarse-grained concurrency control. Two data objects have to belong to the same atomic data set as long as there is at least one consistency constraint relating them. Furthermore, Sha's approach does not

take into consideration the semantics of the consistency constraint itself. Weakening a constraint does not increase the concurrency of a system unless the data objects can be divided into smaller atomic data sets as a result.

To increase the resilience of a compound transaction, Sha suggests storing the values of local variables in stable memory at the end of each atomic commit segment. Our approach is different in that a computation can save a portion of its local state selectively. Also, we describe how a computation can save its state when part of the state may be accessed by other computations concurrently.

1.5.9 Miscellaneous

Other researchers [41, 53] have suggested the use of checkpoints to increase the resilience of a computation. Our work is similar to theirs but is motivated by computations that experience long communication delays. As a result, we emphasize how a caller of a remote program can checkpoint in response to, or anticipation of, long communication delays. To avoid restarting the remote program that is expected to return after a long delay, the calling program should probably checkpoint at the remote call. Mechanisms are also provided to allow the calling program and other ancestor programs to checkpoint if an unexpected delay arises. It seems that in [41, 53] a computation checkpoints the entire state accessible to it, whereas we expect programmers to specify explicitly a portion of the computation state to be preserved.

Another approach to improving resilience is by replicating processors and memory, such as in Tandem and Auragen [3, 13]. These systems consist of a collection of *logical processes*. Each logical process is implemented by two physical processes, one primary and one secondary, on two processors. In the Auragen system, the messages received by a logical process are automatically checkpointed by the system in the memory of a secondary processor. The secondary processor can take over by re-processing the messages to bring its memory up-to-date. Any non-deterministic processing, such as reading the time-of-the-day clock, has to be cast as

another logical process, communicating with this process through messages. The application is not aware of the checkpointing except for management duties, such as choosing the processors for the process pair. In the Tandem system, any state change in the primary processor is checkpointed on the secondary processor. Our checkpoint mechanism is more economical because it assumes only the availability of some permanent memory. It is not always possible to have an available secondary processor to process the checkpoint messages. A site may be disconnected from the rest of the system and the cost of a secondary processor may be too high for some applications.

Replication also provides a limited solution to the concurrency problem. By replicating objects [16, 20], computations can access nearby replicas and long communication delays can be avoided. Unfortunately, replication has its drawbacks. First, it is expensive. When objects are replicated, constraints are imposed on accesses of the objects to ensure consistency. For example, if an object can be read with any one of the replicas, all replicas have to be written when the object is updated. There is also the cost of extra storage. Second, replication does not eliminate all long computations. In the read-one-write-all rule described above, read accesses can be serviced readily as long as there is a replica nearby. The availability of write accesses is decreased, however. The length of a computation that perform updates is actually increased by replication.

Another limited solution to the concurrency and resilience problems is to abort and retry a computation when it cannot be completed quickly. This is unacceptable as a general solution for the following reasons:

- Previous work is wasted.

- If the system does not retry the computation automatically, the user has to retry manually.

- The computation is likely to take longer to complete than if it were allowed to suspend and wait for communication problems to disappear. In fact, when the computation involves many sites and the network

partitions frequently or extensively, the computation is unlikely to be completed without encountering significant communication delay.

- The deferral of the entire computation due to the unavailability of several sites may be unacceptable. For instance, it is undesirable to abort a computation that sets up a meeting among many personal calendars because a few of them are unavailable. Also, the likelihood of setting up the meeting successfully decreases with the passage of time. The proposed meeting, though it may be tentative, is prevented from appearing in the available calendars. Abandoning the unavailable participants and declaring the computation completed is also not the most appropriate behavior.

Chapter Two

System Model

In this chapter we give an account of a system model to prepare for discussion in later chapters. We start in section 2.1 by describing the hardware abstractions on which the distributed systems considered in this thesis are based. In section 2.2, we present a higher level view of these systems and describe how activities inside them can be modelled. Then, in section 2.3, we give a definition for atomicity based on the model.

2.1 Physical Environment and Assumptions

In this dissertation, a distributed system is viewed as a collection of machines connected by a communication network. We call the machines *sites*; they can be any type of machines ranging from portable computers to mainframes or large multiprocessor machines. Sites can be added to or removed from the system dynamically. A site can send *messages* through the network to communicate with other sites. Messages may be lost, duplicated, delayed for an arbitrary period of time, or arrive out of order, but garbled messages will be discarded. In particular, messages can be delayed for an arbitrary period of time because the communicating sites are partitioned. We assume, however, that partitioned sites will be able to communicate eventually. We will not attempt to handle Byzantine failures: the sites in the system are assumed to be cooperative, and redundant bits can be added to packets in the network to keep the probability of undetected garbled messages arbitrarily low.

Each site possesses both volatile and stable memory[3]. A site also possesses one or

[3]This is not strictly necessary. Sites without stable memory can employ remote *stable storage servers*.

more fail-stop processors: a processor may crash at any moment, but when it crashes, it immediately stops all processing before sending any erroneous messages or corrupting its site's stable memory. The implementation of fail-stop processors from unreliable hardware is beyond the scope of this thesis. See [49] for a discussion of the subject. We assume that all crashed sites will recover eventually. When a site recovers, it loses the content of its volatile memory but preserves that of its stable memory.

When a site sends a message to another site, it may expect a response. If none arrives after a long time, it may be because:

- the original message is lost or still on its way, or
- the response message is lost or still on its way, or
- the two sites are partitioned, or
- the responding site is crashed, or
- the responding site is not ready to send the response.

We do not assume that the sender can differentiate among all these cases.

2.2 Model of Computation

At a higher level than the hardware abstractions described above, a system can be viewed as a collection of *objects*. For example, there may be objects controlling access to personal calendars, and objects acting as printer spoolers. An object may reside at one site or may be distributed among many sites. Each object supplies several *operation types*; for example, a personal calendar object can support a *mark* operation and a *delete* operation. *Arguments* can be passed when an operation is invoked. *Results* can be returned with an operation. For instance, a time duration and a purpose can be passed to *mark* as arguments. *Mark* can return either *okay* or *slot_filled*.

Computations are the units of work in a system. Inside a computation, operations on different objects can be invoked. A computation can span multiple sites. Computations are atomic and serve as units for synchronization and recovery.

Atomicity, defined more carefully in section 2.3.3, guarantees that the system behaves as if the computations were executed serially and each computation were executed either in entirety or not at all.

To provide a finer-grained unit in synchronization and recovery, a computation is decomposed into a nested tree of *actions* [40, 48, 34]. Actions are divided into *top-level* actions and *sub-actions*. A computation is associated with a single top-level action. The boundaries of a computation coincide with that of its top-level action. A top-level action can create sub-actions and sub-actions can in turn create their own sub-actions. Operations are *executed* within an action; they must start and finish within the same action. A parent action can create several sub-actions in parallel, but the sub-actions will appear to have executed serially within the parent action. A parent action can also *abort* a sub-action without abandoning the work performed in the rest of itself. An aborted action should appear never to have been executed.

Frequently, a computation creates a sub-action to execute an operation so that the effects of that operation can be undone by aborting the sub-action. However, an action should be distinguished from an operation because the former, like a computation, is merely a mechanism to define a unit of synchronization and recovery. It is not associated with any object.

Aborts of an action may be caused by hardware failures such as site crashes or communication failures. For example, the creator of an action can decide to abort the action if the latter is executed on a remote site and, due to communication failures, the creator cannot determine whether the action has terminated. Aborts can also be initiated by an application program in the absence of hardware failures. For example, an action that executes a *mark* operation in a *set_up_meeting* computation can be aborted if too few participants can attend. Depending on the concurrency control algorithm used in a system, an action can also be aborted because of deadlocks. When an action is aborted, all its sub-actions are aborted. A computation is *aborted* when a top-level action is aborted. In general, we will use the

same terminology to refer to an action and the operations that are executed within it: we say an operation is *aborted* when the action in which it is executed is aborted.

A computation, its nested actions (excluding those aborted), and the operations executed within these actions are *committed* when the top-level action terminates successfully. Committed computations, actions, or operations can not be aborted. A computation, action, or operation is *finalized* when it is committed or aborted. Otherwise it is *tentative*. The *outcome* of a computation, action, or operation is determined when the it is finalized. A nested action is still considered tentative during the time that it has terminated and the top-level action is still incomplete. See figure 2-1 for the possible states that a computation, action, or operation can go through.

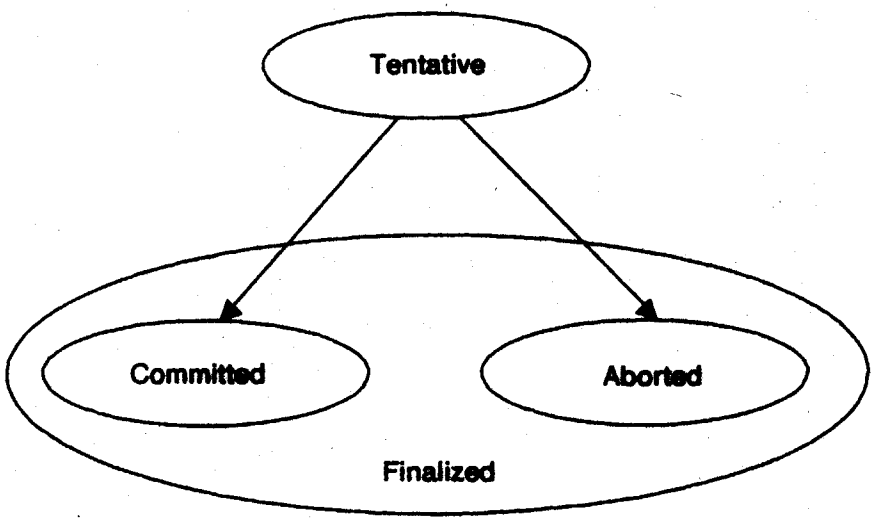

Figure 2-1:States of a Computation/Action/Operation

2.3 Atomicity

In this section we will give a more careful definition of the behavior of a system in which computations are atomic. Our goal is to define atomicity without constraining the system implementations unnecessarily. The definition will be stated only in terms of the *observable behavior* of a distributed system. More importantly, the observable behavior of a system will be cast in terms of the behavior of abstract objects with abstract operations instead of the behavior of objects with read/write operations. Using abstract objects in our definition allows atomicity to depend on the functionality of these abstract objects. Our definition is similar to that in [55] except that ours covers nested actions.

We will describe our atomicity definition in three steps. First, we will describe an *event model*, which models the externally visible activities that happen at the interface of an abstract object with *events*. The activities in a distributed system are modelled with a sequence of events, which we call a *history*. The events in a history can be generated by different computations. Since the model does not include the details of how an object manipulates its internal state, the implementation of the object is not constrained to a particular type of implementation.

Second, we will describe how applications can define their functionality by specifying *serial specifications* for the objects in a system. These serial specifications are similar to the specifications that are usually used to define the semantics of abstract data types [32]. They specify a set of states that an object can be in, and a set of operations that may cause a state transition. Pre-conditions on the state can be attached to the operations.

Third, since a computation can be modelled as a sequence of events, we will define the behavior of a system which executes computations atomically as a set of *atomic histories*. Informally, a history is atomic if it is "equivalent" to an acceptable "serial history." The set of acceptable serial histories is defined collectively by the serial specifications.

Section 2.3.1 describes the event model. Section 2.3.2 illustrates how a serial specification can be expressed conveniently with a *state machine*. The state machines help us capture the semantics of the example applications in later discussions more succinctly. As introducing a formal specification language is beyond the scope of this thesis, we will use informal notations to represent the state machines. Section 2.3.3 defines atomic histories with the event model and the serial specifications.

2.3.1 Event Model

In our event model, an event occurs when an operation is invoked or returned, or when an object is informed of the outcome of an action in which an operation of that object is executed.[4] Each event identifies the object and action that are involved with unique *object identifiers* and *action identifiers*. In this thesis, action identifiers are of the form **a.b...m.n** where **a.b...m** is the identifier of the parent action of **a.b...m.n**. There are four types of events in the model:

invoke events: <operation_type_name(arguments), Object_ID, Action_ID>
The named operation type is invoked at Object_ID. Action_ID is the unique identifier of the action in which the operation is executed.

return events: <result_type_name(results), Object_ID, Action_ID>
Object_ID returns the result of an operation invoked previously.

commit events: <commit, Object_ID, Action_ID>
Object_ID is informed that the action identified by Action_ID is committed.

abort events: <abort, Object_ID, Action_ID>
Object_ID is informed that the action identified by Action_ID is aborted.

To simplify our notation, we assume that an action can only invoke an operation after the result to a previous operation is returned. Parallelism within an action can be

---

[4] We will ignore I/O operations in our model although they are externally visible.

achieved with parallel sub-actions. The invoke and return events of an action can be paired in the obvious way.

To illustrate the event model, suppose **r1** and **r2** are personal calendar objects, each providing a *mark* operation to reserve a slot in the calendar. Further suppose an implementation of *set_up_meeting* that creates sub-actions to execute the individual *mark* operations in the participating personal calendar objects. The following sequence of events may be observed when a user tries to set up a meeting between **r1** and **r2** in a top-level action **a**.

<mark(time, description_of_meeting), r1, a.b>
<okay, r1, a.b>
<mark(time, description_of_meeting), r2, a.c>
<okay, r2, a.c>
<commit, r1, a.b>
<commit, r2, a.c>

or the following may happen, where **d** is another action:

<mark(time, description_of_meeting), r1, a.b>
<okay, r1, a.b>
<mark(time, some_other_business), r2, d>
<okay, r2, d>
<commit, r2, d>
<mark(time, description_of_meeting), r2, a.c>
<slot_filled, r2, a.c>
<abort, r1, a.b>[5]

Obviously, not every sequence of events is "well-formed." For example, a sequence of events should not have a commit event and an abort event for the same action. We will leave a more formal definition of well-formed sequences until Chapter 6 where we construct proofs using the event model. Meanwhile, we assume all the event sequences are well-formed in the sense that they represent some "reasonable" behavior of an implementation and call them *histories*.

---

[5] We have left the outcome of **a.c** unspecified in this example. However, it makes little difference at **r2**.

2.3.2 State Machines

The serial specification of an abstract object can be defined with a *state machine*. Intuitively, a state machine defines the abstract states that the object "passes through" as individual events are "processed." This section describes how a state machine is specified and gives an example.

A state machine for an object r_i has four components: S_i, I_i, T_i, and N_i. S_i is the set of possible states of the state machine. I_i is the initial state. T_i is the set of *transitions*; it corresponds to the set of possible invoke and return event pairs, since not only the invoke event, but also the result that has been returned, determine how the state is to be changed. N_i is a partial function which determines how and under what conditions the state machine would change its state. It takes two inputs: a "before" state and a transition, and returns an "after" state.

N_i can be extended in the following way to accept a *sequence* of transitions as its second input:

$$N_i: S_i \times T_i^* \rightarrow_p S_i$$

such that $N_i(s, \langle\rangle) = s$,

$$N_i(s, t_{seq} \| t) = N_i(N_i(s, t_{seq}), t), \text{ if } N_i(s, t_{seq}) \neq \perp$$

$$\perp, \text{ otherwise}$$

where $\langle\rangle$ **is the empty sequence,** $s \in S_i$, $t \in T_i$, $t_{seq} \in T_i^*$

The partiality of N_i can be used to exclude undesirable transition sequences from the object. In other words, a serial specification can be viewed as defining a set of *acceptable* transition sequences.

Suppose r_i is an object representing a set of integers. It supports three operations: *insert*, *delete*, and *member*. Each operation takes an integer as an argument. *Insert* adds the integer to the set and returns *okay*. *Delete* deletes the integer from the set if the integer is in the set and returns *okay* in any case. *Member* returns a boolean depending on whether the integer is an element of the set. The serial specification of

this set object is defined in figure 2-2. Abbreviations of the form **op_arg_result** will be used for the transition $\langle op(arg), r_i, a\rangle\langle result, r_i, a\rangle$.

S_i: sets of integers
I_i: $\emptyset$
T_i: insert_x_okay = $\langle insert(x), r_i, a\rangle\langle okay, r_i, a\rangle$
delete_x_okay = $\langle delete(x), r_i, a\rangle\langle okay, r_i, a\rangle$
member_x_b = $\langle member(x), r_i, a\rangle\langle b, r_i, a\rangle$
where x is an integer, b is a boolean

$N_i(s, insert\_x\_okay) = s \cup \{x\}$
$N_i(s, delete\_x\_okay) = s - \{x\}$
$N_i(s, member\_x\_b) = s$ if ($x \in s$ and b = true) or ($x \notin s$ and b = false)

Figure 2-2:A State Machine for a Set

In figure 2-2, the object starts with an empty set as its initial state. Three kinds of transitions are possible. Each kind of transitions changes the state in the obvious way. Notice that N_i is defined only under the condition **($x \in s$ and b = true) or ($x \notin s$ and b = false)** for the state **s** and the transition **member_x_b**. For example, a sequence of transitions in which an **insert_x_okay** transition is followed immediately by a **member_x_false** transition would be undefined with respect to N_i and hence unacceptable.

We have introduced the terms "event" and "transition" in this section. Each of them denotes something similar to an operation. The execution of an operation can be viewed as the generation of an invoke event and a return event, or as the generation of a transition. Since different results can be returned by an operation, different transitions may be generated by the execution of an operation. For example, the *member(x)* operation generates either a **member_x_true** or a **member_x_false** transition.

2.3.3 Atomic Histories

In this section we will combine the event model and serial specifications to define a set of *atomic histories*. First, we will define what a *serial history* is. Second, we will describe how a set of *acceptable* serial histories can be defined using the serial specifications. Finally, we will define when a history is equivalent to a serial history. An atomic history is a history that is equivalent to an acceptable serial history. Again, we will rely on informal descriptions and leave a more formal notation until Chapter 6.

A serial history is a history in which events from different actions are not interleaved, an invoke event is always paired with a return event, and only invoke and return events exist. The events in a serial history are ordered by a *linearization*, which can be defined as a total ordering between every pair of sibling actions [34]. As a special case, the top-level actions can be considered as sibling actions. An action **b** is *subsequent* to **a** according to a linearization **L** if either **b** or one of **b**'s ancestors is after **a** or one of **a**'s ancestors in **L**. An action **a** is *prior* to another action **b** if and only if **b** is subsequent to **a**.[6]

Ideally, this prior/subsequent relationship should be extended to the operations executed in two actions in the obvious way. However, because more than one operation may be invoked at the same object by the same action or by actions that bear an ancestral-descendant relationship, the following more complicated definition is needed. An *operation* **a** is *prior* to another *operation* **b** at the same object according to a serial history **sh** if:

---

[6]We assume that there are linguistic mechanisms for the application programmer to express the desired linearization constraints among sibling actions. For example, if **b** is created after **a** by the same parent action, then naturally **b** should be subsequent to **a**. In the rest of this thesis, we only consider linearizations that conform to these constraints. Occasionally, an action will create parallel sub-actions and the order among them is left unspecified by the application. Any total ordering will be acceptable in those cases.

We do not provide any facility for the users to constrain the order among the top-level actions except a guarantee of *external consistency*. If a linearization is externally consistent, a computation **a** is ordered after another computation **b** if **a** is begun after **b** is *completed* and the completion of **b** is communicated to the human user of **a** either externally (outside the system) or internally (through messages sent and received by the sites in the system).

1. the action in which **a** is executed is prior to that of **b** according to the linearization of **sh**, or

2. **a** and **b** are executed in the same action and **a** is executed before **b** in **sh**, or

3. the actions that **a** and **b** are executed in bear an ancestral-descendant relationship and **a** is executed before **b** in **sh**.

An operation **a** is *subsequent* to another **b** if and only if **b** is prior to **a**. This definition is well-formed because we assume that an action can execute only one operation at a time and a parent action cannot invoke any operation while a child action is not terminated.

We define a serial history **sh** to be *acceptable* if, by partitioning **sh** according to the object that an event is associated with, each of the sub-histories is an acceptable transition sequence according to the serial specification of the object associated with that sub-history.

Finally, a history **h** is *equivalent* to a serial history **sh** if **h** is identical to **sh** after all but the committed invoke and return events are removed from **h** and the events left behind are rearranged according to the linearization of **sh**. A history is atomic if it is equivalent to an acceptable serial history. A system is correct if it generates only atomic histories. The linearization of **sh** is called a *serialization order*. By excluding all but the committed events from a history **h**, we formalize the requirement on failure atomicity. By requiring **h** to be equivalent to a serial history in which events are not interleaved, we formalize the requirement on serializability.

Notice that our definition is different from some other atomicity definitions [46, 1]. In these definitions, an atomic history is defined as equivalent to a serial history if the two histories both cause the objects in the histories to reach the same states. Our definition requires that an atomic history has the same *external behavior* as a serial history. Our requirement is sufficient as a user cannot determine the state of an object except through observing its visible behavior. For example, a bank customer

does not care about the internal state of a bank account object as long as he can withdraw what is in his account and the balance on a monthly report is not less than expected. Our definition also has the advantage that we do not have to define the states that the objects will be in after executing a possibly non-serial history.

The major advantage of our atomicity definition, however, lies in its ability to incorporate serial specifications of abstract objects. If serial specifications are relaxed to enlarge the set of acceptable serial histories, the set of atomic histories is also enlarged and the system becomes more concurrent, provided an implementation can utilize the relaxed semantics. Thus concurrency is increased without sacrificing the simplicity offered by atomicity.

Chapter Three

Using Application Semantics

In this chapter we describe the increase of concurrency that can be achieved through the use of application semantics in an implementation. To avoid being encumbered by excessive implementation details, we ignore how the implementation is actually programmed in this chapter. Instead, we assume an idealized implementation that would illustrate how concurrency can be improved when compared to an implementation that, say, uses read/write locks and 2-phase locking. We will describe how the idealized implementation can be approximated by a practical implementation in Chapter 4. The concurrency level afforded by the idealized implementation is only an approximation of the actual concurrency level of a practical implementation. We will argue why it is a useful approximation later in the chapter.

Our idealized implementation consists of multiple program modules, each implementing an abstract object. We assume that a program module has encoded a history of previously invoked operations and that the history information can be retrieved. Each of the objects[7] has an associated queue of requests to invoke operations at that object. These requests are issued by computations running in the system. An object executes by taking a request from its queue, examining the request and the history of previous operations, and determining whether a result can be returned for the requested operation. A result can be returned when an object can guarantee that only atomic histories are generated.

If a result can be returned, the request and the result will be added to the object's

[7] We will use the word "object" to refer to the program module implementing the object.

history. Otherwise, a *conflict* is created and we assume that some action will be taken against the request or the computation that issues the request. We leave these actions unspecified for the moment, since our purpose is to evaluate the concurrency of the implementation, which can be measured by how often a result can be returned to a request. In an actual implementation, the operation may be delayed or the computation that invokes the operation may be restarted when a conflict occurs. Thus, how often a conflict arises is a realistic measure of concurrency. We assume that an object can process a request instantaneously. Details such as how the internal state of an object is encoded and how recovery is performed will be left unspecified. However, we do assume that an object will learn of the outcomes of computations eventually.

In order to illustrate how application semantics improves the concurrency of the idealized implementation, we will describe a *conflict model*, which is one of the contributions of this thesis. The conflict model allows a programmer to determine the condition under which a conflict is created based on the serial specification of the object. We call this condition a *conflict condition*. The model is useful in that it abstracts away the details of the concurrency control algorithm underneath. A conflict condition will remain the same regardless of whether the abstract objects in a system use timestamps assigned at the beginning of execution, or the order in which computations commit, to determine a serialization order. Conflict conditions can serve as a guide when serial specifications are designed, so that concurrency can be traded off against functionality.

In section 3.1 we describe our conflict model. In section 3.2 we use a bank account object to illustrate how conflict conditions can be derived and how concurrency is improved when compared to an implementation that uses, say, read/write locks and 2-phase locking. A bank account example is used in this chapter to facilitate comparison with other work. In section 3.3 we discuss how conflict conditions can be derived for any abstract object. Because the practical implementations that will be described in Chapter 4 approximate the idealized implementation closely, the

process of deriving conflict conditions is also helpful to a programmer writing the practical implementations. In section 3.4 we describe how concurrency can be increased by relaxing the serial specification of an object. Relaxing the serial specification of an object makes conflicts less likely to arise. Using several examples, we will illustrate that there are interesting classes of applications in which the trade-off between concurrency and functionality can be usefully employed. In Chapter 6 we will show that this approach of increasing concurrency is as powerful as other correctness definitions that abandon atomicity [50, 38].

3.1 Conflict Model

This section describes our conflict model and defines conflicts more carefully. We show how the requirement of generating only atomic histories can be translated into a requirement of detecting conflicts.

3.1.1 Generating Atomic Histories

To ensure that only atomic histories are generated by our idealized implementation, the objects in the implementation must guarantee that any history generated will be equivalent to some acceptable serial history. To provide this guarantee, the objects must agree on a particular serialization order, which, in an actual implementation, may be determined by the timestamps that are assigned at the beginning of execution, or by the order in which computations commit. How this serialization order is arrived at in an actual implementation depends on the concurrency control algorithm and is the subject of Chapter 5. We refer to this serialization order determined by the concurrency control algorithm as *the* serialization order of the system. We assume that this is what is referred to when we speak about the serialization order among operations.

3.1.2 Guaranteeing Equivalence to Serial Histories

To ensure that the history generated by the implementation is equivalent to an acceptable serial history defined by the serialization order, each object must ensure that the committed events involving itself, after being rearranged according to the serialization order, will be an acceptable transition sequence according to the object's serial specification. More informally, each object must make sure that the transitions that it generates are *part* of an acceptable serial history defined by the serialization order. We say that an object exhibits *atomic behavior* when this is satisfied.

For example, consider a bank account object r_i with a serial specification described by the state machine in figure 3-1. To simplify our example, we assume the state of the bank account contains only its balance, which can be represented with a real number. The account object has three types of operations: *deposit*, *withdraw*, and *read_balance*. The first two take a real number as an argument. *Deposit* increments the balance by the amount indicated in the argument and returns *okay*. *Withdraw* decrements the balance by the amount indicated in the argument and returns *okay* if the balance is large enough to cover the withdrawal. Otherwise it returns *insufficient_funds*. *Read_balance* returns the balance.

S_i: real numbers
I_i: 0
T_i: <deposit(x), r_i, a><okay, r_i, a> = deposit_x_okay
<withdraw(x), r_i, a><okay, r_i, a> = withdraw_x_okay
<withdraw(x), r_i, a><insufficient_funds, r_i, a> = withdraw_x_insuf
<read_balance(), r_i, a><x, r_i, a> = read_x
where a is an action, x is a positive real number.

N_i(s, deposit_x_okay) = s + x
N_i(s, withdraw_x_okay) = s - x if s $\geq$ x
N_i(s, withdraw_x_insuf) = s if s < x
N_i(s, read_x) = s if s = x

Figure 3-1:A State Machine for a Bank Account Object

Suppose the history depicted in figure 3-2(a) has action **a** serialized before action **b**. Because the transition sequence **deposit_40_okay || read_balance_60** depicted in 3-2(b) is not a member of the set of acceptable transition sequences defined by the state machine in 3-1, the history in figure 3-2(a) is not atomic, and hence the bank account object that generates the history in figure 3-2(a) does not exhibit atomic behavior.

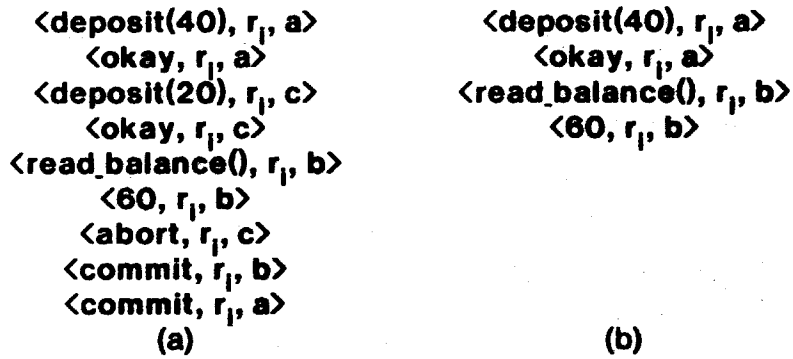

Figure 3-2: A History and a Transition Sequence

3.1.3 Generating Atomic Behavior

To ensure atomic behavior, each of the results returned by an object must be *valid*. A result is valid if the corresponding transition[8] causes a defined state change in the state machine representing the serial specification of the object, given that the state machine starts in a state defined by executing all the committed transitions serialized before this transition. For example, in the previous bank account example, the result 60 is invalid because the state machine has a state of 40 after executing the committed **deposit_40_okay** transition, and the state machine requires a **read_balance_x** transition to have its result x equal to the current state. Notice that when an object generates a result to an operation, it must ensure that not only the result is valid, but that all other results returned to previously invoked operations should remain valid.

---

[8] Recall that a transition corresponds to a pair of invoke and return events.

3.1.4 Generating Valid Results

Obviously, in many cases we need some knowledge of the serialization order to generate valid results. For example, to return a valid result to a *read_balance* operation invoked on a bank account object, we need to determine how the *read_balance* operation is serialized with respect to previously invoked *deposit* and *withdraw* operations.

In addition to knowing the serialization order, we also need some knowledge of the outcomes of the operations that have been invoked. For example, knowing the serialization order between a *read_balance* operation and a *deposit* operation is not enough to determine a valid result for *read_balance*; we also need to know the outcome of the *deposit* operation if the *read_balance* operation is serialized after the *deposit* operation. How the knowledge of a computation outcome is disseminated to the objects that the computation had accessed is determined by a *commit protocol*. We will discuss commit protocols in Chapter 5.

In our conflict model, each object is viewed as possessing some knowledge of the serialization order and the outcomes of the operations that have been invoked. An object may not possess *complete* knowledge because some operations are still tentative; they may be either aborted or committed later. In fact, a computation can be finalized already but the objects that it has accessed will not have the knowledge of its outcome until the outcome is propagated to these objects. In Chapter 5, we will discuss how the serialization order is determined. In some algorithms, it is pre-determined and an object always has complete knowledge of the serialization order among the operations that have been invoked. In some algorithms the order is determined dynamically.

When determining whether a valid result can be returned while preserving the validity of all previous results, an object must be prepared for all the possible combinations of serialization orders and outcomes of the tentative operations that are consistent with the local knowledge. Informally, a conflict is created when no result can be

returned such that it and all previously returned results will be valid under all circumstances consistent with the local knowledge of the serialization order and operation outcomes. For example, a *read_balance* operation invoked at a bank account object may create a conflict because the object lacks the knowledge of the serialization order between the *read_balance* operation and a previously invoked *deposit* operation. The serialization order determines the valid balance to return and there is not a result that will be valid under all circumstances.

3.1.5 Conflicts

A conflict may be created even when an object possesses complete knowledge of the serialization order and operation outcomes. For example, a *deposit* operation can create a conflict because the local knowledge dictates that the *deposit* operation is serialized before a previously invoked *read_balance* operation. Unless the *deposit* operation is refused, the result returned to the *read_balance* operation may be invalidated when the *deposit* operation is committed.

On the other hand, suppose we have a bank account object with an initial balance of $100 and the following history of events:

<withdraw(40), r, a>
<okay, r, a>
<commit, r, a>
<withdraw(30), r, b>
<okay, r, b>

No conflicts would be generated if a *withdraw(20)* operation were invoked on the account object, since an *okay* response to the *withdraw* operation is valid, and the *okay* responses to the previous *withdraw* operations are not invalidated, regardless of the serialization order and outcomes of the operations.

Notice that whether conflicts are created depends not just on operations that are tentative or for which the serialization order with respect to the incoming operation is unknown, but actually on the entire history of events. In the previous example, conflicts would be created if action a had withdrawn more than $50, since whether

the incoming withdrawal can succeed would depend on the outcome of **b**.

Conflicts can also disappear with the execution of new actions not already in the history. Suppose action **a** in the example above had withdrawn more than $50 and a conflict is created when an action **c** invokes *withdraw(20)* at the account object. The conflict will disappear if another action **d** executes a *deposit* operation, commits, is serialized before **c**, and the amount deposited by **d** is large enough to cover the withdrawal by **c**.

When a conflict is created, it can be resolved in several ways:

- delay the operation generating the conflict, e.g., 2-phase locking [17];
- restart the computation generating the conflict, e.g., timestamp algorithm [48];
- make an assumption about the serialization order or operation outcomes and verify the assumption later, e.g., optimistic algorithms [26].

In this chapter, we will not elaborate on how conflicts are resolved. The appropriate way to resolve a conflict is related to how the serialization order is determined. We will discuss the subject in Chapter 5 when we discuss concurrency control algorithms. Suffice it to say that resolving a conflict represents a potentially high cost.

3.1.6 Conclusion

In this section we have described how the requirement of generating only atomic histories can be translated into the requirement of detecting conflicts. The conflict conditions that can be derived from serial specifications are a useful indication of the level of concurrency of our idealized implementation because they abstract away the details of the concurrency control algorithm underneath. The conflict conditions are a good approximation of an actual implementation's concurrency if the actual implementation approximates closely the assumptions of our idealized implementation. For example, for a long computation whose length is attributed to communication delays, regarding the execution of an operation in the computation as instantaneous is a close approximation to the actual execution. Our model of the

structure of the idealized implementation is also sufficiently general so that for any implementation that conforms to this structure, the conflict conditions can be regarded as an indication of the upper bound on an implementation's concurrency level. Executing an operation non-instantaneously would only decrease concurrency.

3.2 An Example

In this section we will use the bank account object defined in figure 3-1 to show the following:

1. How conflict conditions can be derived from a serial specification.
2. How the semantics of an application can be used to increase concurrency over an implementation that uses, say, read/write locks and 2-phase locking.

3.2.1 Read_Balance Operations

Consider when the operation *read_balance* is invoked on the bank account object r_i defined in figure 3-1. Since the **read_balance_x** transition does not mutate the state of r_i, the results returned to the previously invoked operations will remain valid regardless of the outcome and the serialization order of *read_balance*. However, *read_balance* itself returns a result whose validity depends on the serialization order and outcomes of other operations.

Among the set of transitions, only **deposit_x_okay** and **withdraw_x_okay** change the balance. Hence, a conflict is created if the following condition is met:

1. there are *deposit* or successful *withdraw* operations (ones that had returned *okay*) that are tentative and may be serialized before the *read_balance* operation, or

2. there are committed *deposit* or successful *withdraw* operations that may be serialized either before or after the *read_balance* operation,

In other words, the account object can not return any number to the *read_balance* operation that is guaranteed to be valid under all possible situations. Notice that we

have used the terms "may be serialized before/after" and "tentative" in the conflict condition above. It reflects the view in our conflict model that an object possess *some* knowledge of the serialization order and operation outcomes. In the following discussions, we will use the terms "potentially prior" and "potentially subsequent as abbreviations for "may be serialized before" and "may be serialized after" respectively. The terms "definitely prior" and "definitely subsequent" are abbreviations for "definitely serialized before" and "definitely serialized after" respectively.

There is a remote possibility that some tentative *deposit* and *withdraw* operations may cancel one another's effects, and because they are executed by the same action or by sibling actions in the same computation, they are constrained to commit or abort together. In those cases, no conflicts are created although there are tentative *deposit* and *withdraw* operations. We will ignore such possibilities because it is rather unlikely for a computation to deposit as well as withdraw from the same account.

Suppose we have an implementation that uses a read/write lock on the balance such that both *deposit* and *withdraw* would first acquire a read lock and then a write lock, and *read_balance* would acquire a read lock only. For the *read_balance* operation, there is no increase in concurrency with the use of the semantics of the account object. The situations under which conflicts are created for this operation are exactly the same in our idealized implementation and the implementation that uses a read/write lock.

3.2.2 Withdraw Operations

The *withdraw* operations can illustrate how concurrency is increased with the use of application semantics. Consider when the operation *withdraw(x)* is invoked at r_i. The result of the operation is either *okay* or *insufficient_funds*, depending on whether *x* is less than the balance. Since an *insufficient_funds* reply does not imply a change to the abstract state, no previous results returned will be invalidated. However,

because an *insufficient_funds* reply implies that the balance is less than x, the reply can be returned only when the highest possible balance under the possible combinations of serialization orders and outcomes of the operations that may be serialized before the *withdraw* operation is less than x. This highest possible balance can be calculated by adding all the unaborted and potentially prior deposits to the initial balance and subtracting all the committed and definitely prior withdrawals.

Briefly, as long as the balance is so low that there would not be sufficient funds under any circumstances, *Insufficient_funds* can be returned, even if there may be tentative update operations or update operations that may be serialized either before or after the *withdraw* operation. Consequently, some conflicts that would be created had a read/write semantics been imposed are avoided. Although this is not the most significant improvement in concurrency over an implementation using read/write locks, it does illustrate the use of the history of previous invocations, the current operation's argument values and results, and the types of operations in determining whether conflicts are created. This is in contrast to some other approaches that rely only on the operation type and argument values to determine whether conflicts are created [50].

A more significant improvement in concurrency happens when there is a large balance. Again consider the *withdraw* operation but this time consider an *okay* reply. Since an *okay* reply implies a decrement of the balance, the commitment of this operation may invalidate the results of the following kinds of operations:

1. a potentially subsequent *read_balance* operation, or
2. a potentially subsequent and successful *withdraw* operation[9].

To avoid creating any conflicts, there must be no operations of either kind if an *okay* reply is to be returned. The number of conflicts can be further reduced if we recognize that potentially subsequent *withdraw(x')* operations are permissible as long as there is enough money to cover all the withdrawals. Or, more algorithmically,

[9]The result of an unsuccessful *withdraw* operation will not be invalidated because the newly arrived *withdraw* operation will never increase the balance.

when the lowest balance under the possible combinations of serialization orders and outcomes of the operations potentially prior to the *withdraw(x')* operation (with this operation included) is at least *x'*.

Again, in addition to preserving the validity of the previous results, we must also make sure that the *okay* reply is valid before returning it to the *withdraw* operation. Because an *okay* reply implies that the balance is at least *x*, it can be returned only when the lowest balance under the possible combinations of serialization orders and outcomes of the operations potentially prior to the *withdraw* operation is at least *x*.

The discussion above shows that a *withdraw* operation will not create any conflicts as long as the balance is either large enough to accept the withdrawal or small enough to refuse the withdrawal, despite any uncertainty created by concurrent updates. When compared to an implementation that uses read/write locks, it represents a significant improvement on concurrency.

The *withdraw* operation is representative of a large class of operations that can avoid the creation of conflicts most, but not all, of the time. Whether a conflict is actually created depends on the state of the object. The state of the object includes not only what other concurrent operations are being executed, but also all previous committed operations.

We will not discuss the conflicts that will be generated by a *deposit* operation, except to note that because there is only one possible result (*okay*), which is defined for all input states, this result is always valid. However, *deposit* may still create conflicts because it mutates the state of the account and so it May affect the validity of other results. In Chapter 4 we will discuss how this bank account object may be implemented practically. Two different implementations are shown in figures 4-4 and 4-5.

3.3 Deriving Conflict Conditions

In the previous section we illustrated, with the bank account object example, how conflict conditions can be derived. In this section we will generalize from the bank account example, and describe the process by which conflict conditions can be derived from the serial specification of any abstract object. As will be seen in Chapter 4, deriving these conflict conditions is an essential component of an actual implementation.

In general, a conflict condition depends on the type of a transition. For example, different conditions are required for a *withdraw* operation to reply with an *okay* or *insufficient_funds* response. A conflict is created for an operation if every possible transition of that operation creates a conflict. For each transition, the process of deriving the conflict condition can be expressed conceptually as follows:

1. Based on how the abstract state is mutated by the transition, determine the set of potentially subsequent operations in the history of the object whose results may be invalidated. For a transition that only observes the abstract state, such as a **withdraw_x_insuf** transition, the set is empty. For a **withdraw_x_okay** transition, the set includes any potentially subsequent *read_balance* operations and other successful *withdraw* operations.

2. Derive the condition **c1** under which the results of the set of operations discussed in item 1, if the set is not empty, will remain valid with every possible combination of serialization order and outcomes of their potentially prior operations. For example, in order to return *okay* to a *withdraw(x)* operation, there must not be any potentially subsequent *read_balance* operations, and, if there are any potentially subsequent successful *withdraw(x')* operations, the lowest balance under the possible combinations of serialization order and outcomes of the operations potentially prior to the *withdraw(x')* operation (with this withdrawal included) should be at least *x'*.

3. Based on how the abstract state is mutated by other operations, determine the set of potentially prior operations whose outcomes or serialization order may affect the result to this transition. For example, the set is empty for a **deposit_x_okay** transition because the *deposit* operation can only return *okay*. For a **withdraw_x_okay** transition, the set includes all *deposit* and successful *withdraw* operations that are

tentative and potentially prior to this transition, or that can be either prior or subsequent to this transition.

4. Derive the condition **c2** under which the result of this transition will remain valid with every possible combination of serialization order and outcomes of the set of operations discussed in item 3 (if the set is not empty). For example, in order for an *okay* reply of a *withdraw_x_okay* transition to be valid, the lowest balance under the possible combinations of serialization order and outcomes of the operations potentially prior to this transition should be at least *x*. This lowest possible balance can be calculated by assuming that all the potentially prior tentative *deposit* operations are either aborted or serialized after this transition, and all the potentially prior and tentative successful *withdraw* operations are committed and serialized before this transition.

5. The result of this transition can be generated without creating any conflicts if the condition **(c1 and c2)** is satisfied.

The result of following the process above is a conflict condition, **~(c1 and c2)**. The conflict condition can be used as an indication of the concurrency that can be achieved with the particular functionality assumed in the process.

The process described above can be simplified considerably when the concurrency control algorithm is specified. For example, with a timestamp algorithm, there is only one possible serialization order. It is not possible for an incoming operation to be both potentially prior and subsequent to another operation.

3.4 Increasing Concurrency

In the last two sections we described how conflict conditions can be derived based on the serial specification of an object and how the semantics of an application can be used to increase the concurrency of a system. By relaxing a serial specification, or more precisely, by increasing the set of acceptable transition sequences, conflicts become less likely to arise and concurrency is increased. The same idea has been suggested by Liskov and Weihl in [33]. This section uses several examples to illustrate this trade-off between functionality and concurrency. We hope to convince

the reader of the usefulness of the trade-off. In Chapter 6 we will take a more formal approach to show the power of our atomicity definition. We will show that our atomicity definition is at least as powerful as other correctness definitions [50, 38] that had abandoned atomicity. The same gain in concurrency through the use of these correctness definitions can be achieved through trading off functionality in our atomicity definition.

There are several interesting classes of situations in which the semantics of an application can be changed to increase concurrency while the new semantics remains useful. The following list is not intended to be exhaustive, but rather serves to illustrate *some* interesting ways in which semantics can be changed.

3.4.1 Reducing Precision of Numerical Results

In one class, the precision of a numerical result is reduced to allow for more concurrency. For example, a bank account object can provide an operation *lower_bound_balance* that does not take any argument and returns a value that is a lower bound for the balance. The following can be added to the state machine in figure 3-1 on page 54.

T_i: <lower_bound_balance(), r_i, a><x, r_i, a> = lbalance_x
N_i(s, lbalance_x) = s if $s \geq x$

By returning the lowest balance under all possible combinations of serialization orders and operation outcomes, the result is valid yet never create any conflicts. Note that the increase in concurrency is "two-way." Not only does *lower_bound_balance* never create a conflict, but a *deposit* operation invoked afterwards will also avoid creating any conflict due to the possibility that it may be serialized before the *lower_bound_balance* operation.[10] Although the result to *lower_bound_balance* is not exact, it may be useful when the caller is using it as an estimate.

---

[10] However, it is possible for a *withdraw* operation invoked afterwards to create a conflict due to the *lower_bound_balance* operation.

Similar operations that increase the concurrency of the account object are *upper_bound_balance*, *balance_range* (which returns the upper and lower bounds), and *approximate_balance*, which takes a fraction as an argument and returns a value guaranteed to be within a range of the balance determined by the fraction.

T_i: <approximate_balance(p), r_i, a><x, r_i, a> = abalance_p_x
N_i(s, abalance_p_x) = s if $s*(1-p) \leq x \leq s*(1+p)$

Consider another example in which an application is implementing a distributed ticketing agent. A fixed number of tickets is divided among several sites for availability reasons. Each site can sell tickets from its allotment. Occasionally, a computation may be started by one of the sites to record the number of tickets left in other sites and re-distribute the tickets. Suppose we regard each site as a ticket account, supplying operations identical to those of the bank account defined in figure 3-1. The "balance" of the account represents the numbers of tickets unsold in the allotment in this site. Re-distributing the tickets would involve two phases: in the first phase, *read_balance* operations are invoked at each of the sites; in the second phase, based on the values returned by the *read_balance* operations, *deposit* and *withdraw* operations will be invoked at the appropriate sites. The entire computation can be aborted if one or more of the *withdraw* operations returns *insufficient_funds* (more accurately, *insufficient_tickets*).

One of the problems of this implementation is that the semantics of the *read_balance* operation may prove to be too restrictive. Tickets are prevented from being sold while the re-distribution is proceeding because selling a ticket involves invoking *withdraw(1)*, which may create a conflict with a potentially subsequent *read_balance* operation. Concurrency can be improved if the value returned by *read_balance* is treated as a hint. Although the *withdraw* operations in a re-distribution computation may find the actual number of tickets available for re-distribution is not the same as that claimed in the hint, correctness is not compromised. A re-distribution computation can always be aborted. In fact, the two phases of the re-distribution computation can be separated into two computations. However, it may become

counterproductive if the hint loses too much of its accuracy. A more appropriate strategy is to keep the two phases in the same computation but use the *approximate_balance* operation in the first phase to record the tickets left in each site. *Approximate_balance* allows other update operations to proceed concurrently. On the other hand, it sets a limit on the imprecision of the result returned so that in most cases tickets are re-distributed "reasonably."

3.4.2 Conditional Operations

Another interesting class of situations in which the semantics of an application can be relaxed to increase concurrency involves "conditional" operations. Consider a *change_meeting_place* operation for the personal calendar object described in section 1.2.1. The *change_meeting_place* operation takes two arguments, a unique identifier of a meeting and a new place for the meeting. If it finds the meeting in the calendar, it changes the place of the meeting and returns *okay*. Otherwise, it returns *no_such_meeting*. A portion of an informal definition of the state machine defining the serial specification for the calendar object is as follows:

T_i: <change_meeting_place(m, p), r_i, a><okay, r_i, a> = change_place_m_p_okay
<change_meeting_place(m, p), r_i, a><no_such_meeting, r_i, a>
= change_place_m_p_none

N_i(s, change_place_m_p_okay) = s' if s contains the meeting m and s' = s except that the place of m is changed to p

N_i(s, change_place_m_p_none) = s if s does not contain the meeting m

A *global_change_meeting_place* computation invokes a *change_meeting_place* operation at each of the participants of a meeting. The problem with the semantics of *change_meeting_place* is that if a *global_change_meeting_place* computation is started before the corresponding *set_up_meeting* computation is committed, their operations may arrive at different calendars in different orders and conflicts may be created.[11]

---

[11] The motivation for executing the *set_up_meeting* and *global_change_meeting_place* computations concurrently is that at least those reachable participants can be informed of the place change as early as possible. We assume that a participant can observe a tentative *set_up_meeting* computation using the non-deterministic *read_calendar* operations described in Chapter 1.

The conflicts are created because the result to a *change_meeting_place* operation depends on whether the meeting exists in the calendar. Restarts may be needed to resolve these conflicts. The problem can be avoided with the semantics of *change_meeting_place* modified to the following:

T_i: <change_meeting_place(m, p), r_i, a><okay, r_i, a> = change_place_m_p_okay
N_i(s, change_place_m_p_okay) = s' if s contains the meeting m and s = s'
except the place of m is changed to p
s if s does not contain the meeting m

The new semantics implies that *change_meeting_place* will change the meeting place to the new place if the meeting is in the calendar and return *okay*. Otherwise, no changes are made but *okay* is still returned.

With the new semantics, the *mark* and *change_meeting_place* operations from two computations can be executed in different orders in different calendars. No conflicts will be created. The only problem left is to make sure that *set_up_meeting* is serialized before *global_change_meeting_place*. It can be accomplished with, for example, the assignment of appropriate timestamps in a timestamp algorithm.

In this example, the *change_meeting_place* operation becomes "conditional" because whether it makes any changes to the state depends on whether the meeting exists. The reply *okay* does not indicate one way or another. A similar semantics can be used for a *cancel_meeting* operation to reduce conflicts.

A similar but slightly different class of situations can be illustrated by the *withdraw* operation in a bank account object. Originally, we have:

$$N_i(s, \text{withdraw\_x\_insuf}) = s \text{ if } s < x$$

However, by changing the specification to:

$$N_i(s, \text{withdraw\_x\_insuf}) = s$$

a *withdraw* operation can return *insufficient_funds* whenever there is a possible combination of serialization order and operation outcomes that would lead to insufficient funds for the *withdraw* operation. Conflicts due to other *withdraw* or *deposit* operations are minimized.

One can argue that a semantics similar to the more relaxed **withdraw_x_insuf** transition above is necessary for a *make_reservation* operation in an airline reservation object. The semantics is acceptable because most computations that invoke *make_reservation* operations would probably not expect a reply of *insufficient_tickets* to indicate that there are "absolutely" no tickets left. An airline reservation object cannot afford to be blocked for other reservation operations because a computation that had made a reservation is tentative. A computation may last an arbitrarily long period of time, especially when some objects in the system are unreliable. The application would rather turn away customers when it is not absolutely sure that there is a ticket to be sold.[12]

3.4.3 Discussion

A trade-off between precision and concurrency exists in all these examples. Normally, if there are no communication problems and all computations are short, it is probably not worthwhile to sacrifice the precision of the result in exchange for the concurrency. However, concurrency becomes a much more serious concern in a system with long atomic computations. The examples illustrate that there are many interesting situations in which an application would be willing to exchange the precision for the extra concurrency.

Our approach of relaxing the semantics of the application is not without problems. For instance, an implementation of *lower_bound_balance* that always returns zero is a correct implementation as zero is always a valid result. However, it is not very useful. To eliminate this type of behavior, we need to impose additional constraints on the implementation. In this particular example, we need to assert, in addition to the requirements in the serial specification, that there must be a serial history sh consistent with the local knowledge of the account object, such that the result returned by *lower_bound_balance* is not smaller than the balance generated by

---

[12] The fact that airlines overbook their flights does not change our arguments since there is a limit on how much overbooking is allowed.

executing operations in the order of **sh**. In other words, an implementation should only return **x** when **x** is a "possible" balance.

Similarly, to eliminate uninteresting implementations that return *insufficient_funds* to a *withdraw(x)* operation unnecessarily, we assert that there must be a serial history **sh** consistent with the local knowledge of the account object, such that **x** is larger than the balance generated by executing operations in the order of **sh**.

3.5 Summary

In this chapter we described a conflict model, which allows conflict conditions to be derived from the serial specification of an object. We argued that the conflict conditions are useful indications of the concurrency level of an implementation of that object due to the masking of the underlying concurrency control algorithms. Based on the conflict conditions, a programmer can determine the appropriate trade-off between the functionality and concurrency of an application.

Chapter Four

Implementing Atomic Objects

In the last chapter we focused on the functionality of abstract objects. We described how the semantics expressed in the serial specifications of abstract objects can be used to increase concurrency over an implementation that uses read/write locks and 2-phase locking. We discussed how functionality can be traded off for concurrency. In this chapter we will describe how abstract objects can be implemented with a concurrency level approximating that of the idealized implementation in the last chapter.

Like the idealized implementation described in the last chapter, the implementations described in this chapter are *object-oriented*. To guarantee that computations execute atomically, we ensure that each of the abstract objects accessed by a computation behaves atomically[13]. We call an object that behaves atomically an *atomic object*. The advantage of an object-oriented implementation is its modularity. When changes are made in the implementation of an atomic object, other program modules are not affected as long as the serial specification of the object remains unchanged.

A simple way to implement atomic objects is to build them from smaller atomic objects. For example, Argus [31] supports atomic records and atomic arrays. These objects are equipped with read/write locks and follow a 2-phase locking protocol. Their recoverability is implemented using some logging or shadow mechanisms. Because these "system-level" atomic objects provide the necessary synchronization

---

[13]Recall that an object that behaves atomically guarantees that the committed events involving itself, after being rearranged according to the serialization order, will be an acceptable transition sequence according to the object's serial specification.

and recovery, the implementation of abstract atomic objects on top of them can ignore any concurrency or failures in the system. Unfortunately, as we have illustrated in previous chapters, using these system-level atomic objects fails to take advantage of the semantics of an application. The resulting concurrency level is too low for a system with long computations. So in this chapter we will explore how abstract atomic objects can be implemented from objects that do not mask the underlying concurrency and failures.

There are three goals in this chapter. First, we will introduce programming paradigms that allow abstract atomic objects to be constructed easily. These paradigms should not only simplify application programming, but also help the programmer to convince himself of the correctness of the implementations. The simplicity of an implementation is an important consideration because subtle programming errors can be introduced easily, especially when the complexity of an implementation increases with the desire to increase concurrency.

Second, our implementations should maximize concurrency while maintaining reasonable performance in terms of the computing needed to execute an operation. The performance requirements of our implementations are not as stringent as in some real-time applications. Comparing long computations and short computations, the former are not as sensitive to increases in execution time as the latter.

Third, the programming interface and programming paradigms used in this chapter should make the underlying concurrency control algorithm transparent. Either a timestamp algorithm or a locking algorithm, or maybe some other algorithms, could be used to determine the serialization order and the actions to take when conflicts arise. The motivation for this transparency is that a programmer can implement atomic objects without having to learn different concurrency control algorithms. Another motivation is that the programs written are portable when the underlying concurrency control algorithm changes. Implicit in this goal is the belief that different systems may use different concurrency control algorithms. We will justify

this belief in Chapter 5.

This chapter is structured in the following way. First, we present an overview of our programming paradigms in section 4.1. For the next few sections (4.2 to 4.5) we discuss individual aspects of our paradigms in more detail and provide motivation for them. Section 4.6 presents some program examples illustrating our paradigms. To illustrate that there is enough flexibility in our paradigms to optimize an implementation, we discuss some of the trade-offs of different implementation techniques in section 4.7.

4.1 Overview of Implementation Paradigms

When the underlying concurrency and failures are not masked, two issues have to be addressed: synchronization and recovery. The implementations described in this chapter follow the structure of the idealized implementation in the last chapter closely. To simplify synchronization and recovery, we divide them into two levels. At the lower level, concurrent activities at an atomic object are executed such that they appear to be instantaneous. Candidates for such activities are the processing of an invocation request, or the processing of a message that conveys the outcome of a computation. At the higher level, the execution of an atomic computation is viewed as the execution of a collection of these instantaneous activities. Since the collection of instantaneous activities of two atomic computations can interleave with each other arbitrarily, synchronization is needed before processing a new invocation request. An operation can only proceed when no conflicts are created. Recovery is implemented by compensating activities when an object is informed of the abort of a computation.

4.1.1 Lower-Level Synchronization and Recovery

In section 4.2, we will discuss the lower-level synchronization and recovery: how to make the concurrent activities at an atomic object appear to be instantaneous. An obvious solution is to apply the concept of atomicity again. Two kinds of atomic

computations can be used in an implementation. The first kind of atomic computations are the one that we have been discussing in previous chapters. They invoke operations on atomic objects and can last a long time. The second kind of atomic computations are used to make the concurrent activities at an atomic object appear to be instantaneous to one another. They are usually much shorter because these activities are usually small portions of an atomic computation of the first kind.

To distinguish the two kinds of atomicity, we call them *global atomicity* for the long atomic computations of the first kind, and *local atomicity* for the short atomic computations of the second kind. The serialization order that the locally atomic computations appear to be executing in bears no relationship to that of the globally atomic computations. A locally atomic computation can also be committed before the long globally atomic computation that it is executed in is committed. Globally atomic computations appear to execute in a *global serialization order* and locally atomic computations in a *local serialization order*.

Since our model of a computation is a sequence of operation invocations at various objects, we are essentially implementing a long globally atomic computation with a collection of short locally atomic computations. In Chapters 2 and 3 we described how computations can be made atomic by accessing only atomic objects. Corresponding to the two kinds of atomic computations are two kinds of atomic objects: *globally atomic objects* and *locally atomic objects*. A different way to understand our implementations is that we are implementing the globally atomic objects with locally atomic ones.

An analogy can be drawn with the two level of objects in System R [14]. In System R, a page object is locally atomic in the sense that the page locks and recovery mechanisms make the RSS actions (e.g., an operation on an index object, which is a globally atomic object) appear to be atomic to one another. However, since page locks are released at the end of an RSS action, a page object is not globally atomic and a higher level of synchronization and recovery is needed on top of the lower level

of synchronization and recovery provided by the page locks and recovery mechanisms.

4.1.2 Higher-Level Synchronization

In section 4.3 we discuss how the higher-level synchronization is implemented. Given that each operation on a globally atomic object is executed as a locally atomic computation, there is still the task of determining whether a conflict is created with each new operation invocation. In order to determine when conflicts are created, each globally atomic object encodes a history of the operations invoked and the results returned at that object. When a new invocation request arrives, the locally atomic history object is examined to determine whether a conflict is created. If no conflict is created, a result is returned and the transition[14] corresponding to the operation and its result is added to the history object. Otherwise, a conflict is created and must be resolved.

A history object captures the transitions that have been executed at a globally atomic object. The important operations of the history objects are operations to insert a transition, delete a transition, enumerate transitions, and update the status of a transition, which indicates whether the globally atomic computation invoking that transition is committed or tentative. Each operation invoked on a globally atomic object will insert a transition into the history object associated with the globally atomic object. To prevent a history object from growing indefinitely, committed transitions are deleted periodically and "merged" into a more compact representation. When transitions are enumerated from a history object, they can be filtered by their status or the type of operation and results. The caller of the enumerate operation can also supply another transition t and a condition c (e.g., "potentially subsequent according to the global serialization order") such that only transitions that satisfy c with respect to t will be returned. The words "potentially"

[14] Recall that a transition is a pair of invoke and return events.

and "definitely" capture the local knowledge on the global serialization order. With the use of the history objects and other locally atomic objects, an operation can determine what other operations have been executed at the globally atomic object and the possible combinations of serialization orders and operation outcomes. We will describe the implementation of these history objects in more detail in section 4.3 and Chapter 5.

4.1.3 Higher-Level Recovery

In section 4.4 we discuss how the higher-level recovery is implemented. When locally atomic objects are used to implement globally atomic objects, locally atomic computations are committed before the corresponding globally atomic computation is completed. The effects of the operations invoked on the locally atomic objects have to be explicitly undone when the globally atomic computation is aborted. The combined effects of the original operations and the compensating operations should make the globally atomic objects appear to be failure atomic.

We introduce two recovery paradigms in section 4.4. These paradigms are stylized approaches to performing recovery for globally atomic objects implemented with locally atomic objects. Their goal is to simplify the writing of application-dependent recovery code. Simpler code makes it easier to convince ourselves that an implementation is correct.

In the first paradigm, only one mutator operation is performed on locally atomic objects during an operation on a globally atomic object: inserting a transition into the locally atomic history object. When the globally atomic computation containing the operation is committed, the transition can be used to determine other mutator operations to be performed on other locally atomic objects in the representation of the globally atomic object. This type of processing after the globally atomic computation is committed is called *commit processing*. In that sense, the history object serves as an *intentions list*. When the globally atomic computation is aborted, the only compensating activity needed is to delete the transition from the history

object.

In the second paradigm, arbitrary operations can be invoked on the locally atomic objects. Here, the goal is to minimize the work that needs to be done when the globally atomic computation is committed. An *undo operation* is associated with each of the tentative operations on a globally atomic object. The undo operation is invoked when the tentative operation is aborted. The undo operation invokes compensating operations on the underlying locally atomic objects. The history object is a natural place to store names of the undo operations and their arguments. In this case, the history object serves as an *undo log*.

4.2 Global Atomicity and Local Atomicity

The separation of synchronization and recovery into two levels allows division of labor and greatly simplifies the task of programming application-dependent synchronization and recovery. By limiting the higher-level synchronization to happen at operation boundaries, each globally atomic computation will observe only a limited set of well-defined intermediate states of another computation. Similarly, higher-level recovery is simplified because the compensating activities, which can be executed as locally atomic computations, start with a limited set of well-defined intermediate states.

This section describes the idea of having two kinds of atomic objects in more detail. We will describe how locally atomic objects can be implemented and compare our paradigm of implementing globally atomic objects using locally atomic objects with related work.

4.2.1 Definitions of Global Atomicity and Local Atomicity

With the introduction of the distinction between global atomicity and local atomicity, we have separated the objects in a system into globally atomic objects and locally atomic objects. Recall that in Chapter 2 we have defined a history to be atomic if it is

equivalent to an acceptable serial history. We have defined a serial history **sh** to be *acceptable* if, by partitioning **sh** according to the object with which an event is associated, each of the sub-histories is an acceptable transition sequence according to the serial specification of the object associated with that sub-history. The same definition can be used to define local atomicity if we limit our attention to locally atomic objects.

A history is globally atomic if it is equivalent to an acceptable *globally serial history*. A globally serial history is a history in which the events are rearranged according to a linearization of the globally atomic computations. A globally serial history **sh** is *acceptable* if, by partitioning **sh** according to the *globally* atomic object with which an event is associated, each of the sub-histories is an acceptable transition sequence according to the serial specification of the globally atomic object associated with that sub-history. Local atomicity can be defined analogously using the concept of a *locally serial history* in which events are rearranged according to a linearization of the *locally* atomic computations. Notice that there is a *local* serialization order and a *global* serialization order. The behavior of the locally atomic objects is not necessarily valid according to the global serialization order.

4.2.2 Implementing Locally Atomic Computations

We assume that a programmer can declare the boundaries of locally and globally atomic computations. An access to a globally (locally) atomic object should always be enclosed in a globally (locally) atomic computation. Typically, a locally atomic computation is a small portion of a globally atomic computation and should last only a short time (e.g. executing an operation on a globally atomic object). A globally atomic computation can contain several locally atomic computations. A locally atomic computation is committed when it terminates successfully. The locally atomic computation remains committed even though the globally atomic computation that contains it may be aborted later. Notice that given the same serial specification, the concurrency of a locally atomic object is potentially much higher than a globally

atomic object because of the shorter locally atomic computations. If a locking algorithm are used to implement an atomic object, a shorter computation allows locks to be released sooner.

Locally atomic objects can be implemented using a traditional concurrency control algorithm and recovery mechanisms based on read/write semantics [17]. With such an implementation, it is in general inappropriate to access a locally atomic object in a long locally atomic computation. The concurrency of the implementations described in this chapter depends on the use of short locally atomic computations. Alternatively, the same implementation paradigm described in this chapter can be used to implement the locally atomic objects as well as the globally atomic objects. A multiple-level atomicity model can be extended easily from the current dichotomy of global atomicity and local atomicity. In section 4.7.2 we will explore the situations in which the generality of multiple-level atomicity is needed.

In order for the effects of an atomic computation to remain permanent, the updates made by the computation have to be stored into stable memory when the computation commits. Afterwards, the updates will survive site crashes. If accessing stable memory is expensive, the cost of implementing each operation to a globally atomic object with a locally atomic computation may become prohibitive.

To avoid the cost of accessing stable memory every time a locally atomic computation is committed, we can make use of the fact that locally atomic computations are not invoked directly by human users. Consequently, the changes made by a locally atomic computation a do not have to be stored in stable memory until the globally atomic computation that contains a commits, or until other locally atomic computations store their changes in stable memory. The latter condition is needed because other locally atomic computations may have observed the effects of a. Since these other locally atomic computations can also delay their access to the stable memory, all the accesses due to the commitments of locally atomic computations can be piggybacked on a single access when some globally atomic

computation commits. The details of how a distributed computation coordinates its accesses to stable memory in different sites will be discussed in section 5.4.

4.2.3 Related Work

The same idea of having multiple levels of atomicity has been suggested by Beeri et al. in [5] and Moss et al. in [42]. The difference between our work and theirs lies in how synchronization and recovery is performed. To implement serializability, Moss proposes a conflict-based locking mechanism: locks to the level I-1 objects are released when the level I operation that accessed them is finished. However, a lock at level I is retained so that conflicting level I operations are delayed. In [42] the conditions under which "simple aborts" exists are also derived: recovery of a level I object can be achieved by simply omitting the effects of the operations on the level I-1 objects. The conditions require that no conflicting level I-1 operations have been executed by other level I operations.

Weihl [55] describes how atomic objects can be built with other smaller atomic objects and *mutex* objects. Mutex objects behave like monitors [21]. Programs can acquire and release mutex objects to achieve mutual exclusion. The activities performed while the mutex lock is held are serialized as a result. The mutex objects can be viewed as a simple way to implement local atomicity.

4.3 Synchronization

Since locally atomic computations might not be serialized in the global serialization order, a higher level of synchronization is needed so that the behavior of a globally atomic object appears to be globally atomic. Our approach to the higher-level synchronization is to capture sufficient information of the history of events generated at a globally atomic object using history objects, so that it can be used to determine whether conflicts are created. Since all the relevant local information in our atomicity definition is being captured by these history objects, our approach is "complete" in the sense that unnecessary conflicts need not be created except when there is a lack

of global or future knowledge.

In [55] the state of an atomic object is encoded with smaller atomic objects and monitor-like mutex objects. By encoding enough information in these objects, an incoming operation can determine whether conflicts are created and delaying is necessary. It is up to each application to determine how the state is to be encoded. To simplify programming, we have provided a more stylized approach of using history objects for the same purpose.

In general we can expect our programs that handle an invocation request to follow the following pattern:

```
if condition1 then ...; insert t1 into history; return result1 end
if condition2 then ...; insert t2 into history; return result2 end
...
if conditionN then ...; insert tN into history; return resultN end
resolve conflict
```

In the expressions **condition1** the program determines whether certain transitions can be generated without creating any conflicts. If an operation can proceed immediately, a valid result is determined from the history object and other objects. A transition for this operation can be inserted into the history object to be examined by later operations before returning the result. If none of the **condition1**'s are satisfied, a conflict is created and must be resolved. In the rest of this section we will first discuss the operations provided by a history object that plays an important role in the programming of the **condition1** expressions. Then we will discuss the **resolve conflict** statement.

4.3.1 History Objects

Figure 4-1 describes an informal specification of the interface of a history object. To avoid a lengthy digression describing all the operations supported by a history object, figure 4-1 is only a partial list and presents only the operations relevant to our approach of higher-level synchronization. We will continue our description of history operations in section 4.5.1.

A history object can be pictured as a tree of *transition objects*. These transition objects correspond to the different types of transitions in a serial specification. The order of the transitions in the tree is determined by their global serialization order. A tree instead of a linear list is used because a history object may not have complete knowledge of the serialization order. We will not discuss the transition objects in this section. A informal specification of their interface will be presented in section 4.5.2.

```
p_sub = procedure(h: history, t: transition) returns(history)
% returns the largest sub-history of h in which all the transitions
% are potentially subsequent to t.

p_prior = procedure(h: history, t: transition) returns(history)
% returns the largest sub-history of h in which all the transitions
% are potentially prior to t.

d_sub = procedure(h: history, t: transition) returns(history)
% returns a sub-history of h in which all the transitions are
% definitely subsequent to t.

d_prior = procedure(h: history, t: transition) returns(history)
% returns a sub-history of h in which all the transitions are
% definitely prior to t.

p_between = procedure(h: history, t1, t2: transition) returns(history)
% returns a sub-history of h in which all the transitions are
% potentially subsequent to t1 and potentially prior to t2.

d_between = procedure(h: history, t1, t2: transition) returns(history)
% returns a sub-history of h in which all the transitions are
% definitely subsequent to t1 and definitely prior to t2.
```

Figure 4-1:Interface of a History Object

4.3.1.1 Masking Concurrency Control Algorithms

In order to mask the concurrency control algorithm used underneath the programming interface, we follow the conflict model described in section 3.1. We assume that each history object has some knowledge of operation outcomes and the global serialization order determined by the concurrency control algorithm underneath. The operations *p_sub*, *d_sub*, *p_prior*, *d_prior*, *p_between*, *d_between* supported by a history object reflect the view. For example, *p_sub* takes a history

object as its first argument and a transition object as its second argument, and returns the sub-history in which the transitions are serialized *potentially* before the argument transition. How the transitions in the sub-history are ordered is again determined by the global serialization order. The uncertainty about operation outcomes can be reflected with an attribute on the transition objects, which are either committed or tentative. Aborted transitions will be deleted from a history object. We will describe the use of this attribute in sections 4.5.2 and 4.5.3.

These $p\_$ and $d\_$ operations can be implemented and optimized rather straightforwardly given the underlying concurrency control algorithm. For example, if the global serialization order is determined by timestamps assigned at the beginning of a globally atomic computation, the tree in which the transition objects are arranged degenerates to an ordered list, since the global serialization order is known when an operation on an globally atomic object is invoked. There is also no difference between the $d\_$ and $p\_$ operations. A different implementation is required for a concurrency control algorithm in which the global serialization order is determined in a way similar to 2-phase locking. We will defer our discussion of concurrency control algorithms and how these history operations can be implemented until Chapter 5. Note that an implementation for these operations does not necessarily have to copy the history object. A lazy evaluation scheme can be used to enumerate the transitions in the returned history object without changing the semantics of the operations.

With the $p\_$ and $d\_$ operations to capture the global serialization order relationship among the transitions in a history object, the concurrency control algorithm used by the system becomes *transparent* to the application programmers. Although application-dependent synchronization is needed in an implementation, the programmer does not have to be aware of the choice of concurrency control algorithm made by the system. This transparency is the primary characteristic that distinguishes our proposal from all previous ones that involve application-dependent synchronization.

4.3.1.2 Advantages and Disadvantages of Transparency

There are both advantages and disadvantages of providing this transparency. There are two advantages. First, programmers do not have to understand the details of different concurrency control algorithms. The same conflict model can be used during programming. Second, the application programs remain correct even when the underlying concurrency control algorithm is changed. No program modification is needed. One may question how often a concurrency control algorithm would change underneath the application programs. A situation in which this may happen is when application programs are ported, especially for "common" abstract objects such as a FIFO-queue, a set, or some kind of table. Another possibility is for a system to change its concurrency control algorithm in order to combine with another system, so that computations that span both systems can be executed.

One of the disadvantages of the transparency is its over-generality. Application programs become more difficult to write than necessary. For example, given a timestamp concurrency control algorithm, the serialization order is always known. The difference between $p\_$ and $d\_$ disappears. Furthermore, the programmer does not need to consider cases where a transition is both potentially subsequent and potentially prior to another transition.

Another possible disadvantage of the transparency is decreased efficiency. An application program may require several passes over a general history object to determine whether a conflict is created. On the other hand, because of the simpler structure of a history object when the concurrency control algorithm is known, one-pass versions can be constructed more easily than when the concurrency control algorithm is transparent.

Whether these disadvantages outweigh the advantages cannot be evaluated without more experience implementing abstract atomic objects. On the other hand, it seems that without actual experience of the performance of different concurrency control algorithms, a safer investment would be to emphasize portability.

4.3.2 Resolving Conflicts

When an object decides that a conflict has been created, it must resolve the conflict. Depending on the concurrency control algorithm, and why the conflict arises, the range of actions that may be taken include delaying the current operation, restarting the current computation, or making an assumption of the serialization order or some other transition's outcome.

Ideally, a programming interface can provide a generic `resolve conflict` statement which maintains the transparency of the concurrency control algorithm underneath. An intelligent compiler or run-time system can generate code to determine the actions to take, such as when to reschedule a request if delay is needed, or whether to restart a computation or delay a request, or what assumptions to make. However, supporting such a generic statement efficiently is difficult as conflict conditions can be arbitrary expressions.

Depending on the concurrency control algorithm, simple-minded solutions can be devised. For example, in some algorithms a request would be able to proceed given that sufficient time has passed. In those algorithms, a simple solution is to reschedule an invocation request periodically. For some other algorithms, in which restarts and delays are the only two possible alternatives to resolve a conflict, the more pessimistic restart can be chosen whenever delays do not guarantee eventual progress. For algorithms that makes assumptions on operation outcomes and the serialization order, different assumptions can be tried to determine whether they can maintain a valid behavior given that those assumptions are correct.

The drawback of these simple-minded solutions is the loss of concurrency in the form of unnecessary delays, spurious reschedules, unnecessary restarts, or unnecessary assumptions. To provide a compromise between this loss of concurrency and a complicated programming interface, we replace the `resolve conflict` statement with a `retry` statement and require the programmers to specify a *proceed condition* with a `retry` statement. The purpose of the proceed conditions

is to provide a hint to the language system as to when conflicts would not be created. The structure of the proceed conditions is required to obey a *well-formedness* requirement described below so that a proceed condition can be analyzed by the language system. Based on the proceed conditions, the language system can determine whether a delay would lead to eventual progress, when to reschedule, or what assumptions to make.

A proceed condition is taken as a hint to the condition under which an invocation request would be able to proceed. However, in order to guarantee that a request is not delayed indefinitely, a proceed condition should be *well-formed*. A well-formed proceed condition satisfies the following requirements:

1. The proceed condition should be satisfied if:
    a. new operations are not started, and
    b. all current operations in the system, except the one being considered, are finalized and the outcomes are known by all history objects, and
    c. the operation being considered is serialized after all existing transitions and the global serialization order among existing transitions are known.
2. It is not satisfied currently.
3. It is constructed with boolean operations and the operations provided by the history objects.

The first two requirements guarantee that by analyzing a proceed condition, a language implementation can discover the set of "events" that may cause the proceed condition to *become* satisfied. In some concurrency control algorithms, these events may correspond to the finalization of incomplete computations. In some algorithms, the events may involve a restart of the computation that invokes the current operation. The first requirement prevents situations in which the proceed condition is too restrictive. If a proceed condition is too restrictive, the current operation may never be rescheduled, or unnecessary restarts may be initiated. Application programmers should expect the language implementation to make better decisions in determining how to resolve a conflict if the proceed condition is a closer

approximation of the negation of the conflict condition. The second requirement prevents situations in which the proceed condition is already satisfied. If the proceed condition is already satisfied, the language implementation may not be able to determine the set of events that can cause the current operation to resolve the conflict. In that case, the only alternative is busy-waiting in the form of constantly rescheduling or constantly restarting. It is probably not the most desirable solution. The third requirement restricts the structure of a proceed condition so that it can be analyzed by the language implementation. In Chapter 5 we will describe how a language implementation can use the proceed conditions to determine the actions that need to be taken to resolve a conflict.

The `retry` statements are also paired with `begin entry` statements so that a program that uses the `retry` statement has the following form:

```
...
begin entry
        if condition1 ... end
        ...
        if conditionN ... end
        retry whenever c    % c is a proceed condition
```

The semantics of the `retry` statement is to abort any work performed in the last retry loop and retry from the matching `begin entry` statement. A retry might be attempted after a certain delay, a computation restart, or the making of some assumptions. The proceed condition c may or may not be satisfied when the loop is retried.

4.4 Recovery

When locally atomic objects are used to implement globally atomic objects, the effects of a committed locally atomic computation have to be compensated explicitly when the containing globally atomic computation aborts. This section describes how these compensating activities can be programmed. Similar ideas have been proposed in [55, 38, 1]. We will not present any comparison since the purpose of this section is merely to show that recovery paradigms compatible with the rest of our implementation paradigm can be designed.

We will describe two different recovery paradigms in this section. One of them uses the history objects as intentions lists and the other uses them as undo logs. The two paradigms described in this section are not mutually exclusive methods; rather, they represent two ends of a spectrum of possibilities. For example, an application may use one paradigm for certain operations and the other paradigm for the other operations. Depending on the type of an application, one paradigm may be more efficient and/or convenient than the other.

In addition to performing compensating activities, it is also necessary to condense the information contained in the history objects which would otherwise grow indefinitely. We can condense the information contained in the transition objects with a more compact representation after they are committed. How the compaction is performed is related to the recovery paradigm.

4.4.1 Intentions list Paradigm

In the intentions list paradigm, the state of a globally atomic object is represented by a collection of locally atomic objects (which will be called a *snapshot*) and a locally atomic history object. The history object records the transitions of the operations that have been invoked at the globally atomic object. For committed transitions, the application can specify a locally atomic computation which merges their effects into the snapshot and deletes them from the history object. Aborted transitions can be deleted without further action. This kind of commit processing can be viewed as taking the processing "off-line" after the serialization order and the outcomes of the transitions are known. To simplify the application, the committed transitions are merged in the global serialization order. In other words, a committed transitions can be merged only if there are no prior unmerged transitions in the history object.

When an operation is invoked on the globally atomic object, the snapshot and the history object are examined to determine whether a conflict is created. If the operation can proceed immediately, a valid result for the operation is also determined from the snapshot and the history object. Before returning, the transition for this

operation is inserted into the history object. The accesses to the snapshot and the history object are executed in a locally atomic computation.

The intentions list paradigm minimizes the work performed when an operation on the globally atomic object is aborted. If the operation is aborted before the locally atomic computation is committed, changes to all the locally atomic objects will be undone. If the operation is aborted afterwards, the only compensating activity needed is to delete the corresponding transition from the history object. The deletion can be executed as a short locally atomic computation.

Deleting transitions from the history object and merging them into the snapshot as soon as they are committed may create a problem. Occasionally, a committed transition may be needed in a history object to determine whether conflicts are created for an incoming operation that can be serialized before it. Depending on the concurrency control algorithm, committed transition may or may not be needed. In a 2-phase locking algorithm, a committed transition is never needed and a transition can be deleted when it is merged. In a timestamp algorithm, a transition is useful in determining whether conflicts are generated when operations with older timestamps are invoked. If committed transitions are deleted, incoming operations with older timestamps must be refused.

A solution to this problem is to keep a sequence of pairs of snapshots and history objects. Before deleting committed transitions from a history object and modifying the snapshot, a copy of the history object and the snapshot can be kept. For an incoming operation **o**, the appropriate pair of snapshot and history object that is the most updated and yet contains all the transitions that may be serialized after **o** in the history object can be chosen. A complication arises when inserting a transition. Since the transition has to be inserted into all the history object versions, those that have already deleted transitions prior to the transition being inserted have to be discarded.

Since storage is limited, some of the pairs are also discarded when it becomes more

and more unlikely to have incoming operations that need to access the transitions in those pairs. In a timestamp algorithm that assigns timestamps using a real-time clock, transitions invoked by computations that are started before the currently executing computations in a system can be discarded. Without global knowledge, a transition can be deleted if it is estimated to be older than the currently executing computations. If an older computation is still executing and access the history object later, it has to be restarted. Notice that a language implementation can make the maintenance of a sequence of pairs of snapshot and history object transparent to the programmer. It can also make the copying of history object and snapshot more efficient by, for example, keeping *one* history object and having each history object "version" be an index to this single history object.

4.4.2 Undo Log Paradigm

In the undo log paradigm, the state of a globally atomic object is represented by a collection of locally atomic objects (which will be called a *projection*) and a locally atomic history object. In this paradigm, instead of merging the committed transitions during commit processing, the projection is mutated before an operation on the globally atomic object returns. The transition for the operation is also inserted into the history object. The projection should represent the correct abstract state according to any global serialization order in which the transitions in the history object may be serialized, even though there may be many such orders. No extra work is needed if all the tentative operations eventually commit. The accesses to the projection and the history object are executed in a locally atomic computation.

If an operation on the globally atomic object is aborted before the locally atomic computation commits, changes made to the locally atomic objects will be undone. No extra work from the application is necessary. If the operation is aborted after the locally atomic computation is committed, the aborted transition will be deleted from the history object and it will be "unmerged" from the projection with an *undo operation*. The undo operation should compensate for the previous mutation of the

projection and preserve the failure atomicity of the globally atomic object. If two operations are aborted because one of their common ancestor actions is aborted, the undo operation of the operation serialized afterwards is invoked first. An undo operation, along with its arguments, is specified by each operation on the globally atomic object before the latter returns, and remembered in the transition in the history object. The undo operation and the deletion of the transition from the history object are executed in a locally atomic computation. The high-level synchronization performed by an implementation, by guaranteeing that only atomic histories are generated, ensures that this locally atomic computation does not encounter any permanent failures. For example, the undo operation of a *deposit* operation in an account object deducts the amount deposited from the projection. Since an implementation should be prepared for the possibility of the deposit being aborted, there is always enough funds in projection to cover the undo operation. Transient failures that interrupt the undo operation, such as site crashes, can be masked by retrying.

The projection and the history object will be used to determine a valid result that can be returned to an operation invoked at the globally atomic object, and to determine whether conflicts are created. As will be discussed in section 4.7.1, the undo log paradigm may be more efficient than the intentions list paradigm in some applications. The comparison of the two recovery paradigms will be delayed until we have presented some example programs.

Two problems with the undo log paradigm prevent its applicability to general applications. The first problem arises because the paradigm requires the projection to be maintained such that it represents the correct abstract state according to any global serialization order in which the transitions in the history object may be serialized, even though there may be many such orders. For some applications, this is not possible. For example, when the operations *insert(i)* and *delete(i)* are executed on a set object, the correct projection state depends on the serialization order of the two operations.

There are two possible solutions for this problem. One of them is to regard this situation as the creation of a conflict. This is not ideal, as concurrency is decreased unnecessarily. In the set example above, no conflicts are generated by the *insert(i)* and *delete(i)* operations, since the only valid reply for both operations is *okay*, which would remain valid regardless of the serialization order. Another possibility is to allow the projection to be modified when the operation commits. This is also undesirable because of the complexity introduced into the structure of the projection.

The second problem arises because occasionally the projection and a history suffix do not capture enough information on the entire history of previous invocations. For example, suppose the projection is a locally atomic array object representing the abstract state of a set object. Invoking an *insert(e)* operation causes **e** to be inserted into the array. On the other hand, after inserting **e** from the array, we have lost the information indicating whether **e** was in the array before the *insert(e)* operation was invoked, unless the history object contains the transition for the last *insert(e)* or *delete(e)* operation prior to this operation. This information is needed in case the operation is aborted, and also to determine the result of a *member(e)* operation serialized before the *insert(e)* operation. Selecting an undo operation based on the state of the projection when the *insert(e)* operation is executed is not an adequate solution either, since the state of the projection might be changed by other undo operations.

As a remedy, we can delay the deletion of transitions from the history object, or add a snapshot to the state. This raises the question of how transitions should be deleted from the history object. A possibility is to declare all committed transitions which are certain to be serialized before all other tentative transitions eligible for deletion. However, this does not solve the problem described above. A more complicated scheme in which the application makes the final decision over which transition is deleted can be devised. However, it seems complicated and may add a significant cost to accessing the history object.

The addition of a snapshot can be regarded as a combination of the two recovery paradigms. Snapshots can be maintained as described in the last section. They can also be derived more cheaply than described in the previous section by saving old projections. After all the mutator operations that have been merged into the projection are committed, the projection can be regarded as a snapshot.

A final possibility is to encode the necessary information in a more complicated way. In the set example, we can associate an item in the array with a linked list of boolean values. When a *delete* or *insert* operation is invoked/aborted, a boolean value can be inserted/removed from the list. Boolean values at the beginning of the list can be removed by a background process as long as there is a subsequent boolean value inserted by a committed operation.

Despite these limitations and complications, the undo paradigm is still useful in many applications which do not have "overwrite" operations. These overwrite operations, such as *insert(e)* in a set object, have the characteristic that they destroy some significant piece of information in the old state necessary for recovery. Without "overwrite" operations, an operation can determine all the necessary information from the projection and the history object.

4.5 Programming Interface

This section describes some more programming constructs in order to present the program examples in section 4.6. However, this is not meant to be a language proposal. There is a trade-off involved in introducing specialized constructs into a language. While the programs that motivate these constructs become more efficient and easier to write, the language also becomes more complicated and specialized. More detailed study is needed before deciding what linguistic constructs are desirable.

4.5.1 History Objects Continued

The following is a continuation of the description of the operations provided by a history object.

```
delete_first = procedure(h: history) returns(transition)
% returns the transition in h that is serialized before all other
% transitions and is committed.  The transition returned is deleted
% from h.

match = iterator(h: history, t: template) iterates(transition)
% iterates the transitions in h that matches t.

exists = procedure(h: history, t: template,
          p: proctype(transition) returns(bool)) returns(bool)
% returns true if there is a transition s in h such that s matches the
% template t and p(s) returns true.  Otherwise false is returned.
% p is an optional argument.  If p is omitted, only the template t is
% used to filter transitions in h.

% The following operations are internal and invoked only by the
% language system implementation that we will discuss.

insert = procedure(h: history, t: transition)
% inserts t into h.  This operation is invoked by the language
% implementation when an operation on a globally atomic object returns.

get_transitions = iterator(h: history, a: action_id)
                                  iterates(transitions)
% iterates the transitions that are executed in a.  This operation is
% used by the language implementation to search for transitions whose
% status should be updated when information about the outcome of an
% action is received.  We will not show the invocations of these
% operations in our programs in section 4.6.  The update of
% the status of a transition can be executed in a locally atomic
% computation.
```

In addition to the *sub*, *prior*, and *between* operations described in section 4.3.1, the history objects also support an *exists* operation and a *match* operation which allow for searching of particular transitions in a history object. The *exists* operation takes a history object, a *transition template*, and a procedure as arguments. Transition templates will be described in section 4.5.3. Both the transition template and the procedure argument are used to filter the transitions in the history object. The procedure in the procedure argument takes a transition as an argument and returns a boolean. The *exists* operation also returns a boolean as its result. It returns *true* if

there is a transition in the argument history object that matches the template and *true* is returned when the procedure argument is invoked with this transition. *False* is returned otherwise.

In the example programs that we will present in section 4.6, we will in fact need a *closure* rather than a procedure in most calls to *exists*. To allow closures to be passed, the programmer can specify a multiple-argument procedure p:
p = procedure(arg_1: T_1, arg_2: T_2, ..., arg_N: transition) returns(bool)
and the closure can be specified as p(a_1, a_2, ..., a_{N-1}) where a_1 is an object of type T_1.

4.5.2 Transition Objects

Figure 4-2 is an informal specification of the operations supported by a transition object. A transition object can be regarded as a type of record, with various attributes.

```
get_arg1 = procedure(t: transition) returns(type_of_arg1[15])
% return the first argument of the operation represented by t.

% Similarly for get_arg2, get_arg3, ..., get_result1, ...

match = procedure(t: transition, temp: template) returns(bool)
% returns true if t matches temp, otherwise false is returned.

set_status = procedure(t: transition)
% set the status of t to be committed.

set_undo = procedure(t: transition, undo: proctype)
% remembers undo as the undo operation of t.
% This is needed only when the undo recovery paradigm is used.
```

Figure 4-2:Interface of a Transition Object

We assume that the language system supports abbreviations of the form:

---

[15] To avoid cluttering our programs with excessive type information, we have abandoned strict typing here. However, this is not a serious problem as transition objects and history objects can be parameterized.

```
transition$get_arg1(t) abbreviated as t.arg1
transition$set_undo(t, s) abbreviated as t.undo := s
```

In our example programs, a procedure either returns normally or signals an exception. We use a special keyword **okay** to represent the result of a transition when no results are returned. The exception name is used as the result value when an exception is signalled.

We also assume that the language system supports a distinguished variable **this_transition**. This variable can be regarded as the current transition being executed. It can be implemented with a value of the current action identifier which allows comparison with other transitions to determine the relative global serialization order. For example: **history$p_sub(h, this_transition)** returns a history that only has transitions that are potentially subsequent to the caller. We assume that a program can execute

```
this_transition.undo := p
```

to indicate to the language system that the undo operation of the current transition is **p**.

4.5.3 Template Objects

Template objects can be used to match against transitions and filter out irrelevant transitions in a history object. When defining transition templates, programmers are interested only in the status of a transition, the types of the operation and result, the arguments, and the values of the results. We will ignore the action identifiers and object identifiers of the transitions in the templates. For example, for the set object defined in figure 2-2 on page 47, we assume that the language interpreter can parse transition templates of the form:

committed_member_x_true
committed transitions of the form **<member(x)><true>**.

tentative_insert_x_okay
tentative transition of the form **<insert(x)><okay>**.

Determining whether a transition is committed is slightly more complicated than one would expect. Normally, one would expect an implementation of a transition object to associate a status flag with a transition and determine the status accordingly. A complication arises when an action currently executing belongs to the same computation as some of the transitions in the history object. Since an operation may expect to see the effects of other transitions in the same computation that are definitely prior to and would not be aborted independently from itself, the set of "committed" transitions is defined to include these transitions also. Given that another transition t is prior to the current operation o and belongs to the same computation as o, and the names of the actions executing t and o are at and ao respectively, determining whether t would abort independently from o can follow the following algorithm:

1. If an ancestor of at (or at itself) and an ancestor of ao (or ao itself) are parallel sibling actions, then t can be aborted independently from o,
2. otherwise it is not possible.

The action identifiers of at and ao can be used to determine the family relationship.

To avoid long template names in our programs, we assume that abbreviations can be defined (e.g., `ins_x = insert_x_okay`). For similar reasons, we assume that templates can be constructed from other templates using boolean operations. For example, if a program defined `successful_update = withdraw_x_okay or deposit_x_okay`, then a transition matches `successful_update` if it matches either `withdraw_x_okay` or `deposit_x_okay`. We also assume that templates with fixed values in the transition arguments or results can be constructed. For example, if x is an variable defined in a program, then `insert_x_okay` defines a template that matches any transition of an *insert* operation invoked with the object denoted by x.

4.5.4 Resource Managers

The program examples in this chapter are structured in modules called *resource managers*, which are similar to the *guardians* in [30]. In fact, many of the linguistic constructs are copied from the *Argus* language described in [30].

At run time, an instance of a resource manager can be instantiated on a site. (In the rest of this chapter, the term "resource manager" will be used to refer to an *instance* of a resource manager and no distinction will be made.) Each resource manager can be regarded as a virtual site in the system, with a name known by other resource managers. We assume there are name servers [11, 45] which map resource manager names to network addresses. The indirection allows a resource manager to be moved to a different physical site easily. Multiple resource managers can be instantiated on a single physical site.

A resource manager has associated with it a collection of procedures. These procedures share some state, which only procedures from this resource manager can access. A subset of these procedures are exported and can be called by procedures outside the resource manager.

There are many possible ways in which top-level actions and sub-actions can be declared. Since choosing the best way for these declarations is not relevant to the ideas proposed in this thesis, we simply assume that all our example programs are executed in some globally atomic computation. To insulate the caller of a procedure from the site crashes at the resource manager invoked, we also assume that a sub-action surrounding the call would be created if the caller executes in a different resource manager. We assume that the locally atomic computation boundaries are defined by a **`begin entry ... retry whenever c`** statement or a **`begin local computation ... end local computation`** statement.

The caller and callee of a procedure, if they are in different resource managers, communicate using a remote procedure call (RPC) paradigm: the caller suspends until the remote procedure returns. To facilitate the implementation of atomic objects, we use a zero-or-once semantics: when the remote action returns, the action invoked was executed exactly once; otherwise it is executed at most once. We assume that the system will generate an exception to the application when a response has not been received for a remote call after a system-defined timeout. In

Chapter 7 we will describe the measures that the application can take to handle these communication failures. For the time being, we assume that the remote action will be aborted.

Resource managers can be used to implement atomic objects. An object may be implemented within a single resource manager, or with several resource managers. Depending on the overhead of using a resource manager, an application may decide, for example, to implement a single bank account with a resource manager, or many accounts with a resource manager. Procedures in a resource manager can be used to implement the operations on an object.

We assume that the objects used in our example programs are locally atomic unless otherwise specified. Two kinds of locally atomic objects are used: history objects and regular objects. Regular objects consist of the usual `array`, `record`, ... `int` types, which have the usual serial semantics expected for these types.

For the time being, we assume each resource manager has a distinguished history object called `history_suffix`. Several atomic objects implemented on the same resource manager will share the same history object. To shorten our programs, we also assume the distinguished history object is the first argument in a history operation if it is not supplied. Transitions are inserted into `history_suffix` automatically when an operation on a globally atomic object returns. When an action is committed, the status of the transitions in `history_suffix` is updated automatically. When an action is aborted, the aborted transitions in `history_suffix` are deleted automatically.

4.6 Program Examples

Figures 4-3 on page 100 and 4-4 on page 103 show two application programs. Figure 4-3 shows an implementation of the set object of section 2.3.2 with the intentions list paradigm. The implementation is parameterized by the type of the items in a set. Three operations, *insert*, *delete*, and *member*, are supported. The state of the set is

```
% This example uses the intentions list paradigm.

set[T] = resource_manager is insert, delete, member

% abbreviations for transition templates
no_x = member_x_false % <member(x)><false>
yes_x = member_x_true % <member(x)><true>
del_x = delete_x_okay % <delete(x)><okay>
ins_x = insert_x_okay % <insert(x)><okay>

permanent state is
        snapshot: array[T],
        history_suffix: history

while true do % background process
        begin local computation
                t: transition := history$delete_first()
                if transition$match(t, committed_del_x)
                        then ... % remove t.arg1 from snapshot
                   elseif transition$match(t, committed_ins_x)
                        then ... % insert t.arg1 into snapshot
                   end
                end local computation
        end

insert = procedure(x: T)
        begin entry % begin local computation
        if ~history$exists(history$p_sub(this_transition), no_x,
                not_changed(del_x))
           then % insert this transition into history_suffix
                return
           end
        % If no <member(x)><false> transitions can be potentially
        %       serialized after this transition, or if there are but
        %       the effect of this transition is overwritten by another
        %       committed <delete(x)><okay> transition, then this
        %       operation can proceed and return.

        retry whenever % end local computation
                ~history$exists(history$p_sub(this_transition), no_x,
                                not_changed(del_x))
        end insert

not_changed = procedure(op: template, t: transition) returns(bool)
        return(~history$exists(history$d_between(this_transition, t),
                               committed_op))
        end not_changed
```

Figure 4-3:An Implementation of a Set RM with the Intention List Paradigm

```
delete = procedure(x: T)
        begin entry
        if ~history$exists(history$p_sub(this_transition), yes_x,
                           not_changed(ins_x))
           then % insert this transition into history_suffix
                return
           end
        retry whenever ~history$exists(history$p_sub(this_transition),
                         yes_x, not_changed(ins_x))
        end delete

member = procedure(x: T) returns(bool)
        begin entry
        if history$exists(d_prior(this_transition), committed_del_x,
                          not_p_changed(ins_x))
           then % insert this transition into history_suffix
                return(false)
           end
        % If there is a committed <delete(x)><okay> transition
        %       serialized before this transition, and there are no
        %       intervening <insert(x)><okay> transitions, then false
        %       can be returned.
        % The following three clauses are similar.

        if history$exists(d_prior(this_transition), committed_ins_x,
                          not_p_changed(del_x))
           then % insert this transition into history_suffix
                return(true)
           end

        if ~history$exists(history$p_prior(this_transition), ins_x)
           and ~array[T]$member(snapshot, x)
           then % insert this transition into history_suffix
                return(false)
           end

        if ~history$exists(history$p_prior(this_transition), del_x)
           and array[T]$member(snapshot, x)
           then % insert this transition into history_suffix
                return(true)
           end

        retry whenever
        history$exists(d_prior(this_transition),
                committed_del_x, not_p_changed(ins_x)) or
        history$exists(d_prior(this_transition),
                committed_ins_x, not_p_changed(del_x)) or
        (~history$exists(history$p_prior(this_transition), ins_x) and
         ~history$exists(history$p_prior(this_transition), del_x))
        end member
```

Figure 4-3, continued

```
not_p_changed = procedure(op: template, t: transition) returns(bool)
        return(~history$exists(history$p_between(this_transition, t),
                                op))
        end not_p_changed

end set
```

Figure 4-3, continued

represented by a locally atomic array and a locally atomic history object. When *insert* and *delete* operations are committed, they are merged into the snapshot in an infinite loop. The operation `history$delete_first()` returns the committed transition in `history_suffix` that is serialized before all other transitions. Thus the committed operations are merged in the global serialization order.

In the implementation of the *insert* operation, a test is made to ensure that no conflict is created before returning *okay*. No conflict is created if there are not any potentially subsequent `member_x_false` (i.e., `no_x`) transitions. Notice that the `x` in the template refers to the `x` in the argument of the procedure. Furthermore, even if such a transition does exist, no conflict is created if the effects of the *insert* operation are overwritten by another committed `delete_x_okay` transition serialized between this *insert* operation and the *member* operation. This extra filtering is achieved with the closure `not_changed(del_x)`. If a conflict is created, the current local computation is aborted. The proceed condition specified in the `retry` statement is used as a hint to determine how the conflict can be resolved. The implementations of *delete* and *member* follow a similar pattern.

Figure 4-4 shows an implementation of the bank account object of section 3.2 with the undo log paradigm. Instead of merging the committed transitions in an infinite loop, the projection in the implementation is modified when the operation is executed. Each transition is paired with an undo operation. The undo operation is invoked when the transition is aborted. Changes to the projection are undone. The correct undo operation to invoke depends on the result of the original operation.

```
% This example uses the undo log paradigm
account = resource_manager is read_balance, deposit, withdraw

% transition template abbreviations
read = read_balance_x % <read_balance()><x>
dep = deposit_x_okay % <deposit(x)><okay>
withdr = withdraw_x_okay % <withdraw(x)><okay>
successful_update = dep or withdr
insuf_funds = withdraw_x_insufficient_funds
                        % <withdraw(x)><insufficient_funds>

permanent state is
        projection: real % the balance of the account
        history_suffix: history

while true do % background process
        begin local computation
                history$delete_first()
                end local computation
        end

deposit = procedure(x: real)
        begin entry
        if ~history$exists(history$p_sub(this_transition), read) and
           ~history$exists(history$p_sub(this_transition),
                           insuf_funds, high(x))
           then projection := projection + x
                % declare undo operation for deposit
                this_transition.undo := un_deposit(x)
                % insert this transition into history_suffix
                return
           end
        retry whenever
        ~history$exists(history$p_sub(this_transition), read) and
        ~history$exists(history$p_sub(this_transition), insuf_funds)
        end deposit

un_deposit = procedure(x: real)
        % this procedure, together with the update of the status of
        % the deposit transition, are executed as a local computation
        projection := projection - x
        end un_deposit

high = procedure(x: real, t: transition) returns(bool)
        return(highest_possible_balance_at(t) + x ≥ t.arg1)
        end high
```

Figure 4-4:An Implementation of a Bank Account Object with the Undo Log Paradigm

```
highest_possible_balance_at = procedure(t: transition) returns(real)
        return(projection - definite(dep, t) + possible(withdr, t))
        % Unmerge effects of deposits that are definitely subsequent
        % to t and withdrawals that are tentative or potentially
        % subsequent to t.
        end highest_possible_balance_at

low = procedure(x: real, t: transition) returns(bool)
        return(lowest_possible_balance_at(t) - x < t.arg1)
        end low

lowest_possible_balance_at = procedure(t: transition) returns(real)
        return(projection - possible(dep, t) + definite(withdr, t))
        end lowest_possible_balance_at

definite = procedure(opname: template, t: transition) returns(real)
        value: real := 0
        for each s: transition in history$match(history$d_sub(t),
                                                opname) do
                value := value + s.arg1
                end
        return(value)
        end definite

possible = procedure(opname: template, t: transition) returns(real)
        value: real := 0
        for each s: transition in history$match(history$p_sub(t),
                                                opname) do
                value := value + s.arg1
                end
        % Add in the values of potentially subsequent transitions.

        for each s: transition in history$match(d_prior(t),
                                                tentative_opname) do
                value := value + s.arg1
                end
        % Add in the values of tentative transitions, but avoid
        %       repeating those above.
        return(value)
        end possible
```

Figure 4-4, continued

```
read_balance = procedure() returns(real)
        begin entry
        if ~history$exists(history$p_prior(this_transition),
              tentative_successful_update) and
           ~history$exists(history$p_sub(history$p_prior(
                    this_transition), this_transition),
                    committed_successful_update)
           then this_transition.undo := null_procedure
                % no undo is necessary
                % insert this transition into history_suffix
                return(highest_possible_balance_at(this_transition))
           end
        retry whenever
           ~history$exists(history$p_prior(this_transition),
                tentative_successful_update) and
           ~history$exists(history$p_sub(history$p_prior(
                this_transition), this_transition),
                committed_successful_update)
        end read_balance

withdraw = procedure(x: real) signals(insufficient_funds)
        begin entry
        if highest_possible_balance_at(this_transition) < x
           then this_transition.undo := null_procedure
                % insert this transition into history_suffix
                signal insufficient_funds
           end

        if ~history$exists(history$p_sub(this_transition), read) and
           ~history$exists(history$p_sub(this_transition), withdr,
                           low(x)) and
           lowest_possible_balance_at(this_transition) ≥ x
           then projection := projection - x
                this_transition.undo := un_withdraw(x)
                % insert this transition into history_suffix
                return
           end

        retry whenever
           ~history$exists(history$p_sub(this_transition), read) and
           ~history$exists(history$p_sub(this_transition), withdr) and
           ~history$exists(history$p_prior(this_transition),
                tentative_successful_update) and
           ~history$exists(history$p_sub(history$p_prior(
                this_transition), this_transition),
                committed_successful_update)
        % retry when the balance can be determined with certainty.
        end withdraw
```

Figure 4-4, continued

```
un_withdraw = procedure(x: real)
        projection := projection + x
        end un_withdraw

end account
```

Figure 4-4, continued

4.7 Implementation Trade-Offs

In this section we discuss several trade-offs that an implementor of a globally atomic object may have to make. Section 4.7.1 compares the two recovery paradigms. Section 4.7.2 explores the possibility of implementing globally atomic objects with other globally atomic objects. We have not considered this possibility for concurrency reasons. However, such an implementation may have sufficient concurrency if the underlying globally atomic objects are highly concurrent. For example, in implementing a bank object which consists of many bank accounts, implementing the bank object with globally atomic bank account objects is a viable alternative to the paradigm that we have been describing in this chapter. Finally, section 4.7.3 discusses how history objects can be partitioned to reduce the cost of accessing them.

4.7.1 Comparison of Recovery Paradigms

Before comparing the two recovery paradigms, it should be emphasized that the recovery paradigm is a local choice. Each resource manager can be coded with a different recovery paradigm. In fact, the two paradigms can be combined. Both snapshots and projections can be maintained, and each operation can derive its result from the appropriate objects. The comparison below is based on efficiency and programmability. We have commented in section 4.4.2 that occasionally a simple projection and a history object cannot capture the entire state of an object. In those cases, either a more complicated projection is needed or an intentions list paradigm should be used.

Figure 4-5 shows an implementation of the bank account object using an intentions

list paradigm. Comparing with the implementation that uses the undo log paradigm in figure 4-4 on page 103, we see that the undo log paradigm is more efficient in observing a "recent" state of the object, in other words, when there are few prior operations whose effect needs to be "unmerged" from the projection. For example, in executing the procedure `highest_possible_balance_at(t)`, if there are few tentative `withdr` transitions in `history_suffix` and `t` is a recent transition, then only those few transitions potentially subsequent to `t` and the tentative `withdraw` transitions need to be "unmerged" from the projection. On the other hand, the effects of all the `withdr` transitions definitely prior to `t` and `deposit` transitions potentially prior to `t` have to be merged with the snapshot. Conversely, the intentions list paradigm is more efficient in observing an "old" state.

The efficiency of the paradigms also depends on the frequency of aborted operations. If an operation is aborted, the undo log paradigm has to undo the changes made to the projection, in addition to wasting the effort expended in changing the projection when the operation is invoked. With an intentions list paradigm, little work is needed. However, we anticipate aborted operations to be uncommon.

The intentions list paradigm is easier to program with because the programmer does not have to provide undo operations. With the undo log paradigm, undoing is needed not only during recovery, but also when the effect of operations has to be "unmerged" from the projection, such as when determining the result to a *read_balance* operation. The fact that the state has to be merged and unmerged may complicate programming.

4.7.2 Implementing Atomic Objects with Atomic Objects

In previous sections we have discussed how to implement globally atomic objects using locally atomic objects. The implementations are characterized by application-dependent synchronization and recovery because a locally atomic computation is committed before the globally atomic computation in which it executes is committed.

```
% This example uses the intentions list paradigm

account = resource_manager is read_balance, deposit, withdraw

% transition abbreviations
read = read_balance_x % <read_balance()><x>
dep = deposit_x_okay % <deposit(x)><okay>
withdr = withdraw_x_okay % <withdraw(x)><okay>
successful_update = dep or withdr
insuf_funds = withdraw_x_insufficient_funds
                % <withdraw(x)><insufficient_funds>

permanent state is
        snapshot: real
        history_suffix: history

% background process
while true do
        begin local computation
                t: transition := history$delete_first(history_suffix)
                if transition$match(t, committed_dep)
                        then snapshot := snapshot + t.arg1
                   elseif transition$match(t, committed_withdr)
                        then snapshot := snapshot - t.arg1
                   end
                end local computation
        end

deposit = procedure(x: real)
        begin entry
        if ~history$exists(history$p_sub(this_transition), read) and
           ~history$exists(history$p_sub(this_transition), insuf_funds,
                           high(x))
           then % insert this transition into history_suffix
                return
           end
        retry whenever
        ~history$exists(history$p_sub(this_transition), read) and
        ~history$exists(history$p_sub(this_transition), insuf_funds)
        end deposit

high = procedure(x: real, t: transition) returns(bool)
        return(highest_possible_balance_at(t) + x ≥ t.arg1)
        end high

highest_possible_balance_at = procedure(t: transition) returns(real)
        return(snapshot - definite(withdr, t) + possible(dep, t))
        end highest_possible_balance_at
```

Figure 4-5:An Implementation of a Bank Account Object with the Intention List Paradigm

```
lowest_possible_balance_at = procedure(t: transition) returns(real)
        return(snapshot - possible(withdr, t) + definite(dep, t))
        end lowest_possible_balance_at

low = procedure(x: real, t: transition) returns(bool)
        return(lowest_possible_balance_at(t) - x < t.arg1)
        end low

definite = procedure(opname: template, t: transition) returns(real)
        value: real := 0
        for each s: transition in history$match(history$d_prior(t),
                                                committed_opname) do
                value := value + s.arg1
                end
        return(value)
        end definite

possible = procedure(opname: template, t: transition) returns(real)
        value: real := 0
        for each s: transition in history$match(history$p_prior(t),
                                                opname) do
                value := value + s.arg1
                end
        return(value)
        end possible

read_balance = procedure() returns(real)
        begin entry
        if ~history$exists(history$p_prior(this_transition),
                tentative_successful_update) and
           ~history$exists(history$p_sub(history$p_prior(
                this_transition), this_transition),
                committed_successful_update)
           then % insert this transition into history_suffix
                return(highest_possible_balance_at(this_transition))
           end
        retry whenever
           ~history$exists(history$p_prior(this_transition),
                tentative_successful_update) and
           ~history$exists(history$p_sub(history$p_prior(
                this_transition), this_transition),
                committed_successful_update)
        end read_balance
```

Figure 4-5, continued

```
withdraw = procedure(x: real) signals(insufficient_funds)
        begin entry
        if highest_possible_balance_at(this_transition) < x
           then % insert this transition into history_suffix
                signal insufficient_funds
           end

        if ~history$exists(history$p_sub(this_transition), read) and
           ~history$exists(history$p_sub(this_transition), withdr,
                           low(x)) and
           lowest_possible_balance_at(this_transition) ≥ x
           then % insert this transition into history_suffix
                return
           end

        retry whenever
           ~history$exists(history$p_sub(this_transition), read) and
           ~history$exists(history$p_sub(this_transition), withdr) and
           ~history$exists(history$p_prior(this_transition),
                tentative_successful_update) and
           ~history$exists(history$p_sub(history$p_prior(
                this_transition), this_transition),
                committed_successful_update)
        end withdraw

end account
```

Figure 4-5, continued

The locally atomic computations are also serialized in a different order than the globally atomic computations. An alternative is to construct globally atomic objects with globally atomic objects. For example, instead of using locally atomic record objects, a bank account can be constructed with globally atomic record objects. No application-dependent synchronization or recovery is needed. Application programs can be written as if there were no concurrency or failures.

We argued that using globally atomic record objects to construct globally atomic account objects is not concurrent enough when a globally atomic computation can last a long time. The semantics of the record objects does not allow sufficient concurrency. However, this approach of implementing a globally atomic object with smaller globally atomic objects may be viable if the underlying globally atomic objects are abstract objects and their semantics can be used to increase concurrency.

In this section we will illustrate two different approaches of implementing a bank object. A bank object consists of many bank accounts. The semantics of a bank object is described in figure 4-6. Notice the difference between a bank object and a bank *account* object. At first glance, a bank object may look similar to a bank account object because they both support *withdraw*, *deposit*, and *read_balance* operations. However, the bank object is in fact capturing the state of a *collection* of bank accounts; hence it also supports a *transfer* operation that transfers funds between two accounts and an *audit_sum* operation that returns a sum of the balances in all the accounts.

S_i: a mapping s from account numbers to real numbers
I_i: undefined for any account number yet
T_i: $\langle$deposit(an, x), r_i, a$\rangle\langle$okay, r_i, a$\rangle$ = deposit_an_x_okay
$\langle$withdraw(an, x), r_i, a$\rangle\langle$okay, r_i, a$\rangle$ = withdraw_an_x_okay
$\langle$withdraw(an, x), r_i, a$\rangle\langle$insufficient_funds, r_i, a$\rangle$ = withdraw_an_x_insuf
$\langle$read_balance(an), r_i, a$\rangle\langle$x, r_i, a$\rangle$ = read_an_x
$\langle$transfer(an1, an2, x), r_i, a$\rangle\langle$okay, r_i, a$\rangle$ = transfer_an1_an2_x_okay
$\langle$transfer(an1, an2, x), r_i, a$\rangle\langle$insufficient_funds, r_i, a$\rangle$ =
transfer_an1_an2_x_insuf
$\langle$audit_sum(), r_i, a$\rangle\langle$x, r_i, a$\rangle$ = audit_sum_x
where a is an action, ani's are account numbers,
x is a positive real number.

N_i(s, deposit_an_x_okay) = s' where s' = s except s'(an) = s(an) + x
N_i(s, withdraw_an_x_okay) = s' if s(an) $\geq$ x, where s' = s except s'(an) = s(an)-x
N_i(s, withdraw_an_x_insuf) = s if s(an) < x
N_i(s, read_an_x) = s if s(an) = x
N_i(s, transfer_an1_an2_x_okay) = s' if s(an1) $\geq$ x,
where s' = s except s'(an1) = s-x, s'(an2) = s + x
N_i(s, transfer_an1_an2_x_insuf) = s if s(an1) < x
N_i(s, audit_sum_x) = s if Σ_is(an_i) = x

Figure 4-6:A State Machine for a Bank Object

We will assume that an operation on the bank object last for only a short period of time, even though the operation may involve more than one account. This is possible if, for example, the bank object is implemented on a single site. However, we assume that there are long computations in this application because some computations

might access multiple bank objects.

An obvious approach to implement the bank object is to implement it using locally atomic record/array/history objects and the paradigm described in this chapter. The semantics of the bank object is used to increase concurrency. A different approach is to implement the bank object out of the globally atomic bank account objects that we have described in this chapter. The implementation is simple because the account objects are globally atomic and hide the concurrency and failures in a system. The complexity is instead hidden in the implementation of the account objects. Notice that in this approach the semantics of the *account* objects is used to increase concurrency.

We will compare these two approaches of implementing a bank object. The difference lies in one approach using the semantics of a bank object to increase concurrency, while the other using the semantics of the account objects. We will argue that concurrency and complexity of the implementations can be comparable. However, there are several potentially significant differences also.

4.7.2.1 Two Approaches to Implement a Bank Object

In figure 4-7 we show a partial implementation of a bank object using the implementation paradigm described in this chapter and some *locally* atomic record, array and history objects. Each bank operation is executed as a local computation, in which locally atomic record, array, and history objects are accessed. We have not shown the locally atomic record and array objects in figure 4-7 because they are hidden in the implementation of the locally atomic `directory` object.

Figure 4-8 shows an implementation of a bank object that uses globally atomic account objects. Notice that because concurrency and failures are hidden by the implementation of the account objects, the implementation in figure 4-8 is relatively simple.

```
% This implementation uses an intentions list paradigm.

bank = resource_manager is read_balance, deposit, withdraw, transfer,
                           audit

% abbreviations for templates
% <read_balance(an)><x> or <audit_sum()><x>
read_an = read_balance_an_x or audit_sum_x

% <deposit(an, x)><okay> or <transfer(an', an, x)><okay>
deposit_an_x = deposit_an_x_okay or transfer_an'_an_x_okay

% <withdraw(an, x)><okay> or <transfer(an, an', x)><okay>
withdraw_an_x = withdraw_an_x_okay or transfer_an_an'_x_okay

% <deposit(an, x)><okay> or <withdraw(an, x)><okay>
successful_update = deposit_an_x_okay or withdraw_an_x_okay

% <withdraw(an, x)><insufficient_funds> or
%                   <transfer(an, an', x)><insufficient_funds>
insuf_funds = withdraw_an_x_insufficient_funds or
                   transfer_an_an'_x_insufficient_funds

permanent state is
        snapshot: directory[account_number, real]
        history_suffix: history

while true do % background process
        begin local computation
                t: transition := history$delete_first()
                if transition$match(t, committed_deposit_an_x)
                   then snapshot(t.arg1) := snapshot(t.arg1) + t.arg2
                elseif transition$match(t, committed_withdraw_an_x)
                   then snapshot(t.arg1) := snapshot(t.arg2) - t.arg2
                end
                end local computation
        end

deposit = procedure(an: account_number, x: real)
        begin entry
        if ~history$exists(history$p_sub(this_transition), read_an) and
           ~history$exists(history$p_sub(this_transition), insuf_funds,
                           high(an, x))
           then % insert this transition into history_suffix
                return
           end
```

Figure 4-7:An Implementation of a Bank Object with the Intention List Paradigm

```
        retry whenever
           ~history$exists(history$p_sub(this_transition), read_an) and
           ~history$exists(history$p_sub(this_transition), insuf_funds)
        end deposit

high = procedure(an: account_number, x: real, t: transition)
                                        returns(bool)
        return(highest_possible_balance_at(an, t) + x ≥ t.arg1)
        end high

highest_possible_balance_at = procedure(an: account_number,
                                        t: transition) returns(real)
        return(snapshot(an) -
              definite(withdraw, an, t) + possible(deposit, an, t))
        end highest_possible_balance_at

audit_sum = procedure() returns(real)
        begin entry
        if ~history$exists(history$p_prior(this_transition),
              tentative_successful_update) and
           ~history$exists(history$p_sub(history$p_prior(
              this_transition), this_transition),
              committed_successful_update)
           then r: real := 0
              for an: account_number in
                              directory$elements(snapshot) do
                    r := r + balance_at(an, this_transition)
                    end
              % insert this transition into history_suffix
              return(r)
           end
        retry whenever
        ~history$exists(history$p_prior(this_transition),
              tentative_successful_update) and
        ~history$exists(history$p_sub(history$p_prior(this_transition),
              this_transition), committed_successful_update)
        end audit_sum

definite = procedure(opname: template, an: account_number,
                     t: transition) returns(real)
        value: real := 0
        for each t: transition in history$match(history$d_prior(t),
              committed_opname_an_x) do
              value := value + x
              end
        return(value)
        end definite
```

Figure 4-7, continued

```
possible = procedure(opname: template, an: account_number,
                     t: transition) returns(real)
        value: real := 0
        for each t: transition in history$match(history$p_prior(t),
                                                opname_an_x) do
                value := value + x
                end
        return(value)
        end possible

balance_at = procedure(an: account_number, t: transition) returns(real)
        return(snapshot(an) -
               definite(withdraw, an, t) + definite(deposit, an, t))
        end balance_at

...

end bank
```

Figure 4-7, continued

Depending on how the globally atomic account objects are implemented, our bank application may or may not have enough concurrency. An application that uses a combination of the implementation in figure 4-8 with the implementation of globally atomic bank accounts in figure 4-9 is probably not concurrent enough in a system with long computations, since no semantics of the application has been utilized. On the other hand, if the application uses the globally atomic bank account implementations described in figures 4-4 and 4-5, which make use of the semantics of a bank account, the resulting application allows much more concurrency.

Notice that there is some similarity between figures 4-5 and 4-7. For example, the `deposit` operations in the figures are almost identical. However, part of this similarity is due to clever encoding of the transition templates. The `read_an` transition template in figure 4-7 stands for either a `read_balance_x` transition or an `audit_sum_x` transition that involves `an`, whereas the `read` transition template in figure 4-5 stands for a `read_balance_x` transition only.

Figure 4-10 depicts the two different approaches to implement a globally atomic bank object. Notice that both Approach 1 and Approach 2 use the implementation

```
bank = resource manager is deposit, withdraw, read_balance, audit,
                                transfer

permanent state is
        dir: directory[account_number, account_resource_manager]
        % this is a directory that maps account numbers to the account
        % resource manager that implements the account.  To simplify
        % our example, we assume all input account numbers are valid.

deposit = procedure(a: account_number, r: real)
        dir(a).deposit(r)
        % dir(a) looks up the resource manager corresponding to a.
        % The syntax "resource_manager_name.procedure_name(arguments)"
        % is used to call a procedure in another resource manager.
        end deposit

withdraw = procedure(a: account_number, r: real)
                        signals(insufficient_funds)
        dir(a).withdraw(r) resignal insufficient_funds
        % The resignal statement catches any insufficient_fund
        % signal from the withdraw procedure of the bank account object
        % and resignals it to the caller of this withdraw procedure.
        end withdraw

read_balance = procedure(a: account_number) returns(real)
        return(dir(a).read_balance())
        end read_balance

audit_sum = procedure() return (real)
        result: real := 0
        for an: account_number in directory$elements(dir) do
                result := result + read_balance(an)
                end
        return(result)
        end audit_sum

transfer = procedure(from, to: account_number, amount: real)
                        signals(insufficient_funds)
        withdraw(from, amount) resignal insufficient_funds
        deposit(to, amount)
        end transfer

end bank
```

Figure 4-8:A Simple Implementation of a Bank Object

```
account = resource manager is deposit, withdraw, read_balance
                              % Procedures exported

permanent state is
        state: globally_atomic_record[balance: real, ....]

deposit = procedure(r: real)
        state.balance := state.balance + r
        end deposit

withdraw = procedure(r: real) signals(insufficient_funds)
        if state.balance < r then signal insufficient_funds end
        state.balance := state.balance - r
        end withdraw

read_balance = procedure() returns(real)
        return(state.balance)
        end read_balance

end account
```

Figure 4-9:A Simple Implementation of a Bank Account Object

paradigm described in this chapter, though at different levels of abstraction.

4.7.2.2 Comparison of the Two Approaches

In this section we compare Approach 1 and Approach 2. The two approaches are comparable in complexity and concurrency. However, there are also some subtle differences. The complexity of Approach 1 is in the implementation of the globally atomic bank objects using locally atomic objects, whereas the complexity of Approach 2 is in the implementation of the globally atomic bank account objects. Building globally atomic bank objects from globally atomic account objects is a simple task, because the necessary synchronization and recovery have been implemented with the underlying globally atomic account objects.

Concurrency and Complexity

It is not obvious whether Approach 1 or Approach 2 is more desirable. In an implementation that follows Approach 1 (figure 4-7), the *transfer* and *audit_sum* operations can avoid creating any conflicts with each another as long as the other is

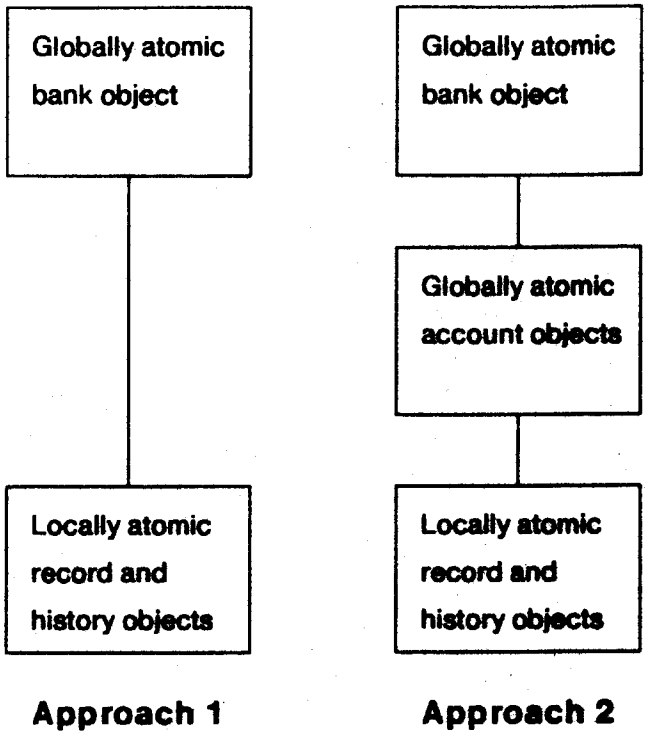

Figure 4-10: Two Different Approaches of Implementing a Globally Atomic Bank Object

finished but maybe tentative. Because a *transfer* or *audit_sum* operation can be part of a much longer computation, this period of being finished but tentative can be quite long. The concurrency is due to the semantics of *audit_sum*, which only requires the result returned to be a sum of the balances, and that of *transfer*, which keeps the total balance constant although it changes individual balances. As a result, when one of the operations is completed, the other operation can proceed even when the first operation is not committed.

If the bank object is implemented using globally atomic account objects, the *transfer* and *audit_sum* operations will be translated into *withdraw/deposit* and *read_balance* operations on the bank account objects. These operations interfere with one another and cause conflicts to be created even after the higher-level operations at

the bank object are already finished.

One may be tempted to implement the *transfer* and *audit_sum* operations with special versions of the lower-level operations. In fact, a possible implementation is to define the globally atomic bank accounts with the semantics in figure 4-11.

S_i: [s1, s2] where s1 and s2 are real numbers
I_i: [0, 0]
T_i: ⟨deposit(x), r_i, a⟩⟨okay, r_i, a⟩ = deposit_x_okay
⟨withdraw(x), r_i, a⟩⟨okay, r_i, a⟩ = withdraw_x_okay
⟨withdraw(x), r_i, a⟩⟨insufficient_funds, r_i, a⟩ = withdraw_x_insuf
⟨tdeposit(x), r_i, a⟩⟨okay, r_i, a⟩ = tdeposit_x_okay
⟨twithdraw(x), r_i, a⟩⟨okay, r_i, a⟩ = twithdraw_x_okay
⟨twithdraw(x), r_i, a⟩⟨insufficient_funds, r_i, a⟩ = twithdraw_x_insuf
⟨read_balance(), r_i, a⟩⟨x, r_i, a⟩ = read_x
⟨aread_balance(), r_i, a⟩⟨x, r_i, a⟩ = aread_x
where a is an action, x is a positive real number.

N_i([s1, s2], deposit_x_okay) = [s1 + x, s2 + x]
N_i([s1, s2], tdeposit_x_okay) = [s1 + x, s2]
N_i([s1, s2], withdraw_x_okay) = [s1-x, s2-x] if s1 $\geq$ x
N_i([s1, s2], withdraw_x_insuf) = [s1, s2] if s1 < x
N_i([s1, s2], twithdraw_x_okay) = [s1-x, s2] if s1 $\geq$ x
N_i([s1, s2], twithdraw_x_insuf) = [s1, s2] if s1 < x
N_i([s1, s2], read_x) = [s1, s2] if s1 = x
N_i([s1, s2], aread_x) = [s1, s2] if s2 = x

Figure 4-11:A Specialized State Machine for a Bank Account Object

Special operations *tdeposit* and *twithdraw* are provided for the implementation of *transfer*, and an *aread* operation is provided for *audit_sum*. In essence, each bank account keeps track of two "balances." The first balance is the normal one. The second "balance" is updated when the update is *not* invoked by a *transfer* operation. The second balance is read to calculate the sum of the balances. As a result, no conflicts are created between an *audit_sum* operation and a *transfer* operation.

This technique does not work in general situations because the cost of keeping extra state information can be prohibitive. For example, suppose a database of employee records is partitioned among several sites. The application provides operations to

transfer employee records from one partition to another, update information in the employee records, and to evaluate queries. The interference between the transfer and query operations poses a problem similar to the interference between *transfer* and *audit_sum* in the bank application. However, keeping an extra copy of an employee record at the old partition when it is transferred does not seem to be acceptable. Not only is extra storage required, updating the employee records becomes more costly also. A more appropriate solution in this example would be to allow the partitions to return a superset of the records in that partition. The coordinator of the query can ignore redundant records collected from the partitions. If a record is being transferred from one partition from another, both partitions can return the record before the transfer computation is finalized. When a record is deleted, both partitions must be informed.

Although the examples above do not show that concurrency is necessarily decreased when globally atomic objects are implemented with other globally atomic objects, they do illustrate that the semantics of the lower-level globally atomic objects have to be customized. The customization increases the complexity of implementing a globally atomic object.

Reliability and Efficiency

A possible disadvantage of implementing the bank object with locally atomic objects is the centralization of synchronization and recovery information. When compared to an implementation in which the history objects are distributed among many account objects, the history object used by a bank object contains more transitions and is more expensive to access. In addition, the reliability of the application can be reduced because its functioning depends on the availability of the centralized history object of the bank object. A possible solution to overcome these disadvantages is to partition or replicate the state (directory) of the bank object. We will describe how history objects can be partitioned and/or replicated in the next section.

Another possible problem of implementing globally atomic objects with locally atomic

objects is the limitation in the length of locally atomic computations. Since locally atomic objects are implemented with other locally atomic objects, such as locally atomic arrays or records, the lengths of the locally atomic computations have to be kept short to minimize the cost of conflicts created in accessing locally atomic objects. Keeping locally atomic computations short is not always possible, especially when a locally atomic object may be partitioned or replicated. To minimize the cost of these conflicts, we can have a multiple-layered model of atomicity, instead of the dichotomy of local atomicity and global atomicity. A layer l atomic object can be implemented with a layer l + 1 atomic object. The semantics of the objects in each layer can be utilized. For example, a layer l-1 atomic bank object can be implemented with a layer l atomic history object and a layer l atomic bank account object, which can in turn be implemented with a layer l + 1 atomic history object and a layer l + 1 atomic record object.

4.7.3 Partitioning and Replicating History Objects

When computations are long, their transitions may remain tentative and be kept in a history object for a long period of time. Performance can become a problem when there are too many transitions in a history object. An obvious solution to this problem is to partition history objects into smaller history objects.

In our previous program examples, we assume one history object is shared by all the atomic objects implemented in a resource manager. This is not necessary and can be changed by having multiple history objects declared in the resource manager, with history operations specifying the history object being operated on explicitly.

More complicated schemes of partitioning the history object are possible. For example, if an operation x is only interested in a subset of the different types of transitions, a sub-history can be created containing only those transitions. The cost of inserting a transition, which happens once, may become higher because the transition may have to be inserted into several sub-histories. However, the cost of x accessing a history object is lowered because there are probably fewer transitions in

the sub-history in which x is interested.

For example, the history of a set object can be partitioned according to properties of the items involved. For example, if a set object is a set of integers, the history object can be partitioned according to the range of values of the arguments. A more complicated example can be illustrated with the history object in the implementation of a globally atomic employee file object. The application may decide to partition the history object and the snapshot/projection objects according to the department that a transition is related to. For example, if a transition involves an employee in Department X, then only the partition of Department X needs to be accessed. When an employee is transferred from Department X to Department Y, a transition is inserted into each of the partitions of the two departments. If a query involves potentially every department, all the partitions need to be accessed.

In these examples, the locally atomic and logically centralized history object is implemented with locally atomic history partitions. The semantics of the partitions reduces the number of partitions that need to be accessed. If only a few partitions are accessed, the cost of accessing the entire history is reduced and the operation can proceed even when some partitions are not available.

In [20] a history object is partitioned *and* replicated for availability reasons. The history object is not partitioned according to properties of the transitions but rather the availability of the replicas (partitions). Each transition has an *initial quorum* and a *final quorum*. When the history object is read, an initial quorum of replicas is read to guarantee that every transition relevant to the current operation is contained in at least one of the replicas. When a transition is inserted into the history object, a final quorum of replicas is accessed. For example, in determining whether conflicts are created for an observer operation, other observer transitions are irrelevant. Consequently, the replicas read may not overlap with the replicas updated when previous observer transitions are inserted.

A simpler scheme of replicating the *entire* history object can be used to increase

availability, though not performance, over an un-replicated implementation. However, because a history object is usually both read and written, a read-one-write-all algorithm will not increase availability. A slightly more complicated read-write quorum scheme [16] is needed.

Another way of partitioning the history object can be illustrated by the example in the previous section. By implementing the bank object with globally atomic bank account objects, no history needs to be kept for the bank object; rather, the history information is partitioned among the account objects. This partitioning is simpler than those described above because no centralized image is necessary. Unfortunately, as the example has illustrated, this partitioning may cause a loss of concurrency.

Finally, there is a possibility of avoiding the cost of accessing the history object altogether in some applications. Consider the semi-queue object specified in figure 4-12.

S_i: sets of items (we assume items enqueued are unique)
I_i: $\emptyset$
T_i: <enqueue(x), r_i, a><okay> = enqueue_x_okay
<dequeue(), r_i, a><x, r_i, a> = dequeue_x
<dequeue(), r_i, a><empty, r_i, a> = dequeue_empty
where a is an action, x is an item.
N_i(s, enqueue_x_okay) = s $\cup$ {x}
N_i(s, dequeue_x) = s - x if x $\in$ s
N_i(s, dequeue_empty) = s if s = $\emptyset$

Figure 4-12:A State Machine for a Semi-Queue

An implementation using an intentions list recovery paradigm can be found in figure 4-13. In the implementation of the *dequeue* operation, we find that when there are items in the snapshot, the history object has to be accessed to make sure the items have not been dequeued by previous *dequeue* operations. To avoid this access, the snapshot object can be partitioned into two arrays, say **a1** and **a2**. The idea is to put all the items which are definitely not dequeued into **a1** and items which *may* have

```
% This example uses the intentions list paradigm.

semiq[item] = resource_manager is enqueue, dequeue

% transitions in history suffix
% dequeue_x = <dequeue()><x>
% dequeue_empty = <dequeue()><empty>
% enqueue_x_okay = <enqueue(x)><okay>

permanent state is
        snapshot: array[item],
        history_suffix: history

while true do
        begin local computation
                t: transition := history$delete_first()
                if transition$match(t, committed_dequeue_x)
                        then ... % remove x from snapshot
                   elseif transition$match(t, committed_enqueue_x_okay)
                        then ... % insert x into snapshot
                   end
                end local computation
        end

dequeue = procedure() returns(item) signals(empty)
        begin entry
        for x: item in array[item]$elements(snapshot) do
                if ~history$exists(dequeue_x) then return(x) end
                end

        if history$exists(history$d_prior(this_transition),
                committed_enqueue_x_okay, not_used)
           then % insert this transition into history_suffix
                return(x)
           end

        if ~history$exists(history$p_prior(this_transition),
                enqueue_x_okay, not_d_used) and empty_snapshot()
           then % insert this transition into history_suffix
                signal empty
           end

        retry whenever
        ~history$exists(history$p_prior(this_transition),
                        tentative_enqueue_x_okay)
        and ~history$exists(tentative_dequeue_x)

        end dequeue
```

Figure 4-13:An Implementation of a Semi-Queue Object

```
enqueue = procedure(x: item)
        begin entry
        if ~history$exists(history$p_sub(this_transition),
                           dequeue_empty)
           then % insert this transition into history_suffix
                return
                end
        retry whenever
        ~history$exists(history$p_sub(this_transition), dequeue_empty)
        end enqueue

not_used = procedure(t: transition)
        x: item := t.arg1
        return(~history$exists(history$d_sub(t), dequeue_x))
        end not_used

not_d_used = procedure(t: transition)
        x: item := t.arg1
        return(~history$exists(history$d_sub(t), committed_dequeue_x))
        end not_d_used

empty_snapshot = procedure() returns(bool)
        for x: item in array[item]$elements(snapshot) do
                if ~history$exists(committed_dequeue_x)
                   then return(false)
                   end
                end
        return(true)
        end empty_snapshot

end semiq
```

Figure 4-13, continued

been dequeued into **a2**. When a committed enqueue transition is merged, the item can be inserted into **a2** if there is a subsequent dequeue transition of that item, and into **a1** otherwise. When the *dequeue* operation is invoked, it can enumerate **a1** first. If there is an item in **a1**, it can be deleted from **a1**, inserted into **a2**, and returned to the caller, without ever accessing the history object. If no items are found in **a1**, **a2** can be searched. Occasionally, a *dequeue* operation may be aborted after the item has been moved into **a2**. The item may stay in **a2** without affecting the correctness of the implementation; a background process can move such items back to **a1**.

4.8 Conclusion

In this chapter we have described programming paradigms that an implementation of an atomic object can follow. These paradigms simplify the writing of application-dependent synchronization and recovery code. With simpler code, arguing the correctness of an implementation becomes easier. In particular, we introduce the notion of locally atomic objects and locally atomic computations. Synchronization and recovery are partitioned into those performed by the locally atomic objects and those performed by the implementation of the atomic object. This partitioning helps the programmer convince himself that the implementation is correct.

In this chapter, we have also introduced the use of history objects, which capture all the relevant local information needed by an object to determine whether conflicts are created. The interface provided by these history objects makes the underlying concurrency control algorithm transparent to the programmers. This transparency provided by the history objects, together with the transparency provided by the conflict model, allow the programmer to design the functionality and program the implementations of an application without having to understand the details of the underlying concurrency control algorithm.

For recovery, we have discussed an intentions list paradigm and an undo log paradigm. By imposing constraints on how an operation may mutate the locally atomic objects, the recovery activities become a more structured process.

We have presented several program examples and illustrated the use of the paradigms we introduced.

Finally, we have discussed several implementation strategies and their trade-offs. First, there is the local choice of the recovery paradigm. Second, globally atomic objects can be implemented using locally atomic objects or other globally atomic objects. Finally, the cost of accessing history objects can be minimized by various ways of partitioning them. These options provide opportunities to customize the

implementation to specific needs.

Chapter Five

Concurrency Control Algorithms

In our conflict model and programming interface, each atomic object is assumed to possess some knowledge of the serialization order and operation outcomes. Based on this knowledge, an object can express conflict conditions without knowing the details of how the serialization order and operation outcomes are arrived at. In this chapter we discuss how the objects arrive at a serialization order through a concurrency control algorithm. The protocol that different entities in a distributed system use to arrive at a consensus of the outcome of a computation is called a *commit protocol*. Many papers [17, 37, 52] have been written on the subject and we will discuss it only briefly at the end of this chapter.

This chapter seeks to fulfill two goals. First, we will show that the programming interface that we present in Chapter 4 can be implemented on top of a large class of concurrency control algorithms. In particular, we show how the history operations, such as *p_sub* and *d_prior*, can be implemented. We will also show how the `retry` statement can be implemented so that the appropriate actions are taken when conflicts are created.

Second, we will argue that in some situations the concurrency of a system can be significantly affected by how the serialization order is determined. In deriving conflict conditions, we find that whether a conflict arises depends on the functionality of the operations of an application and the local knowledge of the serialization order and operation outcomes. Previous chapters have focused on how the functionality of an operation determines the likelihood of conflicts. This chapter shows that there are special situations in which some concurrency control algorithms can reduce the likelihood of costly conflicts significantly when compared to other algorithms. For

example, suppose long computations are rare in a system and it is unlikely for two long computations to overlap their execution. Given these conditions, it may be possible to develop concurrency control algorithms that distinguish between long computations and short computations so that only short computations will be restarted or cause delays in other computations. Given that, the overall cost of conflicts in these algorithms can be much smaller than that incurred by existing algorithms. One of the contributions of this thesis is the design of two novel concurrency control algorithms that are adapted to systems with long atomic computations.

Section 5.1 briefly describes some of the existing concurrency control algorithms and compares the likelihood of costly conflicts in these algorithms. Section 5.2 describes two novel concurrency control algorithms and explains the situations in which these algorithms can reduce the overall cost of conflicts significantly. Section 5.3 describes the implementation of the programming interface in Chapter 4 given that different concurrency control algorithms can be used underneath. Section 5.4 discusses commit protocols briefly.

To separate our consideration of concurrency control algorithms and the functionality of an application, we will use the terms "observer" and "mutator" in this chapter to refer to two classes of operations. The functionality of the first class observes the abstract state of an object. The second class mutates the abstract state. For example, a *read_balance* operation is an observer, a *deposit* operation is a mutator, a successful *withdraw* operation is both because it observes that there are sufficient funds and mutates the abstract state. To simplify our discussion, we will assume that conflicts are created when:

1. an observer may be serialized after a tentative mutator, or

2. a mutator may be serialized before an observer previously invoked.

This is not true in all cases, such as when the observer is a *withdraw* operation and the mutator is a *deposit* operation. No conflicts would be created if there were

sufficient funds for the withdrawal regardless of the deposit.

Also, we exclude the possibility of parallel sub-actions in our description of concurrency control algorithms. A computation executes with only one locus of control and sub-actions within a computation are serialized by the order they execute. In most cases, it is straightforward to extend the algorithms to handle parallel sub-actions. We will give brief explanations of how an algorithm can be extended when the extension is not obvious.

5.1 Concurrency Control Algorithms

The goal of a concurrency control algorithm is to ensure that a serialization order among the committed computations exists. It also determines the actions that need to be taken when a conflict arises.

Many different concurrency control algorithms have been proposed. Some of them [48] use the order in which computations are started as a serialization order, some [17] use the order in which computations commit as a serialization order. The actions that are taken when conflicts arise depend very much on how a serialization order is arrived at. In sections 5.1.1 and 5.1.2 we enumerate some of the well-known concurrency control algorithms that have been proposed in the literature and discuss the likelihood of costly conflicts in these algorithms. Enumerating all the algorithms proposed in the literature would be impossible. However, the performance of the algorithms described in section 5.1.1 and 5.1.2 is representative of a large class of algorithms.

5.1.1 Static Concurrency Control Algorithms

In general, concurrency control algorithms can be classified according to the time that the serialization order is determined. In *static* algorithms, the serialization order is determined at the beginning of a computation. When a computation is started, a unique timestamp is associated with the computation, and the value of the timestamp

determines the serialization order[16]. In the rest of this section, we will use Reed's multi-version timestamp algorithm [48] as an example of static concurrency control algorithms. In his algorithm, computations with larger timestamps are serialized after computations with smaller timestamps.

Recall that conflicts are created under two types of situations:

1. when a mutator **m1** is invoked and it may be serialized before an observer **o1**, or

2. when an observer **o2** is invoked and it may be serialized after a tentative mutator **m2**.

In [48], the mutator **m1** is refused and the computation that invokes **m1** is restarted with a larger timestamp. Restarting a computation with a larger timestamp is the only way to change the serialization order. The observer **o2** is delayed until the tentative mutator **m2** is finalized.

An alternative to refusing **m1** is to abort some of the previously invoked operations, such as the observer **o1**. However, this is not always possible as those operations may have committed. Furthermore, a race condition may develop in deciding to commit or abort those operations. The sites making the decisions must be synchronized.

The concurrency problem created by the formation of conflicts can be evaluated with the *likelihood* of formation and *costs* of the conflicts. The likelihood and cost of a conflict can be classified according to the two types of situations in which it is created. Besides depending on the functionality of the operations of an application, the likelihood that the first type of conflicts are created in a static algorithm depends on whether operations are arriving at an object in the predetermined static order. The more operations arrive in that order, the less likely it is that the first type of conflicts are created. However, considering that the time between when the

---

[16]For parallel sub-actions, it suffices to extend the timestamps to non-overlapping time ranges, with sub-actions subdividing the parent's time range. For details see [48].

computation begins (the timestamp assigned) and when the object is accessed has a larger variance in our system than in systems with only short computations, we may have a significantly larger percentage of operations arriving in an order that differs from the static serialization order. In particular, an operation from a remote caller may find that many local computations with larger timestamps have been executed, and probably committed, during the time the call travelled from the caller to the callee site. Obviously, when a computation may remain tentative for a long period of time, the second type of conflicts is also more likely to arise in a system with long computations than in a system with only short computations.

In static algorithms, the cost of the first type of conflicts is a restart of the refused computation. This is potentially disastrous as the refused computation may have executed for a long period of time. In addition to lost work, restarts also cause delays. If the top-level action of the refused computation is executed at a remote site, the restart is likely to be expensive: it adds an extra round-trip delay at least. Note that when a conflict of the first type is created in a static algorithm, the operation that creates the conflict is likely to be invoked from a remote site. It is also possible that a restarted computation may encounter another conflict and have to be restarted again.

The cost of the second type of conflicts depends on how long the tentative operation **m2** remains tentative. An alternative to delaying the observer **o2** is to restart the computation that invokes **o2** with a smaller timestamp. It is not always the most appropriate action as the computation may encounter some other conflicts of the first type because of the smaller timestamp. However, concurrency may be increased if:

1. the computation does not invoke mutator operations and cannot create the first type of conflicts, and
2. the computation has only been started recently and restarting it has a small cost, and
3. the mutator **m2** is invoked by a long computation and may not be finalized until after a significant delay.

If the conditions described above can be evaluated at run time, the concurrency

control algorithm can minimize the cost of the conflict accordingly.

Although the likelihood of formation and costs of conflicts are generally higher in a system with long computations than in a system with only short computations, there are some situations under which we can expect the two kinds of systems to have a similar concurrency level. In a static algorithm, short computations are less likely to create the first type of conflicts than long computations. This is because they are less likely to encounter operations with larger timestamps already executed. The cost of restarting a short computation is also lower. Short computations may include single-site computations and computations that execute within a tightly-coupled group of sites that can communicate with short delay. Consequently, if all the mutator computations are short and only read-only computations are long, short computations can usually succeed without incurring costly conflicts. Moreover, because read-only computations are never restarted unless a restart is cheaper than a delay, the long read-only computations are only delayed by short mutator computations.

5.1.2 Dynamic Concurrency Control Algorithms

In *dynamic* algorithms, the serialization order is determined during the execution of the computations at the objects. Typically, the serialization order between two computations in a dynamic algorithm is determined by the order in which they finish accessing the last object. The moment immediately after the last object is accessed is called a computation's *locked point* [6], which, to simplify matters, can be equated with the moment at which the computation is finalized.

Dynamic concurrency control algorithms have the property that an operation is always serialized after all other finalized operations. Other tentative operations can be either prior or subsequent to this operation in the serialization order. Given these properties and that a conflict of the first type is created (i.e., a mutator m1 is invoked and it may be serialized before an observer o1), the observer o1 must be tentative.

Usually the mutator **m1** is delayed until the observer is finalized.[17] Delaying the mutator eliminates the possibility that it can be serialized before the observer. When a conflict of the second type is created (i.e., an observer **o2** is invoked and it may be serialized after a tentative mutator **m2**), the observer **o2** is delayed until the mutator **m2** is finalized.[18]

Occasionally, several computations may be deadlocked, each waiting for another to finalize. A deadlock detection algorithm [43] can be used to detect and break the deadlock by restarting some computation in the cycle. After the victim computation has been chosen, one of its actions that causes the delay of other actions can be aborted and its parent action can be notified. If the parent action has not proceeded beyond the end of the victim action (e.g., the victim action has not finished, or the parent action has created several parallel sub-actions and is waiting for all of them to finish), the parent action can abort the victim action and start a new instance of it. Otherwise, the parent action becomes a victim action also. The process is repeated until the top-level action is reached. The top-level action could not have been committed since it is deadlocked.

The likelihood of formation of both types of conflicts in a dynamic algorithm increases with the number of tentative operations at an object. Unfortunately, the likelihood will be higher in a system with long computations than in one with only short computations. This is because the time between when an object is accessed and when the computation is finalized is, in general, longer in a system with long computations. To make matters worse, the delay caused by a conflict adds to the length of a computation and make the expected number of tentative operations even

---

[17] An alternative is to delay the *commitment* of the mutator until the observer is finalized. In this alternative, the mutator operation can proceed but cannot commit until the observer is finalized.

[18] An alternative is to delay the *commitment* of the observer until the mutator is committed. The observer can proceed but may be aborted later if the mutator is aborted. Depending on the likelihood of a computation being aborted, this alternative may or may not improve concurrency. However, the improvement is not significant because the observer has to wait for the mutator to commit in any case.

larger.

In dynamic algorithms, the cost of a conflict is a possibly long delay. Moreover, when the probability of being delayed is high, there is a possibility of cascaded delaying: a tentative operation delaying other operations is in turn delayed by another tentative operation.

In addition to cascaded delaying, there is also the cost of deadlocks. There is some empirical evidence [18] that deadlocks are uncommon in systems with short computations. However, it is unclear whether this is still valid when computations are long. When a deadlock occurs, there is the cost of detection, which usually involves passing messages around [43], and the cost of restarting a victim action.

5.2 Improving Concurrency with Concurrency Control Algorithms

In this section we suggest some novel concurrency control algorithms. We will show that these algorithms can reduce the likelihood that costly conflicts will arise in a system with long atomic computations. In particular, we will describe a *hierarchical* conflict algorithm that preserves the advantages of a static algorithm over a dynamic algorithm (short computations are less likely than long computations to encounter conflicts and less expensive to restart, and observer operations create conflicts only when a restart is cheaper than a delay), and generates less conflicts for long computations.

We will also describe a *time-range* concurrency control algorithm in which each computation is associated with a time-range instead of a timestamp. The static and dynamic algorithms can be shown to be special cases of this algorithm. The time-range algorithm allows the user to choose a "privileged" class of computations that can be made to be serialized after all other computations except those also in the privileged class. The ability to do so reduces the possibility that a privileged mutator computation is restarted or delayed.

5.2.1 Hierarchical Concurrency Control Algorithm

Suppose each computation is given a *period identifier* and a *serialization identifier*. The serialization identifiers can be assigned with unique timestamps (from a real-time clock). The two identifiers are concatenated, with the period identifier more significant, and used to determine the serialization order of the computations.[19]

Period identifiers are *not* necessarily unique. Computations receive their period identifiers from *period counters*. We assume each site has its own period counter, which is updated with the current clock value when a *distributed* computation is started at this site, or when the period identifier of an incoming distributed computation is larger than the current period counter.[20] The period counter will lag behind the clock most of the time, assuming that most computations are local. Notice that, although the period identifiers are not unique and lag behind the real time clock, the same is not true for serialization identifiers. Local computations in this algorithm are similar to those in the static algorithm in that they are unlikely to be restarted in their short duration and can be restarted inexpensively.

Distributed computations perform better in this algorithm than in a static algorithm. Consider a distributed computation c started at clock time t; it will have a period identifier and a serialization identifier, both approximately t. Consider the period counters at the remote sites that c will visit. If they are also t at the time c is started, then this algorithm will have the same performance as the static algorithm because it is just as likely that conflicts will be created. If they are greater or smaller than t, then this algorithm will perform more poorly or better respectively. Given that a period counter at a site s is updated only when there are other distributed computations visiting or started at s, the period counters at the remote sites that c will visit are likely

---

[19]The serialization identifiers can be extended to non-overlapping time ranges to handle parallel sub-actions.

[20]We assume that whether a computation is local or distributed can be determined, for instance, from the syntax of the program. In any case, this information is only a hint and does not affect the correctness of the algorithm.

to be less than t at the time c is started. An exception is when the clocks at those remote sites are running ahead of the one used to generate t and other distributed computations have visited or been started at those remote sites recently. If we assume distributed computations to be rare or clocks to be closely synchronized, the exception is unlikely to happen.

Given that the period counters of the remote sites that c will visit are smaller than t, it will be less likely for c to be aborted due to an old timestamp when c finally arrives at a remote site s. This is because the local computations started at s before s's period counter exceeds the period identifier of c will be serialized before c, and not cause c to be restarted. This algorithm performs better when distributed computations are infrequent.

Note that incrementing the period counters is an optimization and does not affect the correctness of the algorithm. A period counter can be left unchanged when, say, a distributed computation that only involves nearby sites is started. To avoid these distributed computations being restarted, the period counters of the nearby sites can be synchronized frequently by bringing the smaller counters to the values of the larger counters.

The hierarchical algorithm can be useful in a system in which distributed computations and long computations are rare. For example, most of the computations in a calendar application will be local. Occasionally a distributed computation involving a meeting is started. Also, in many distributed databases, the majority of computations will be local if the data is partitioned according to locality of reference.

Consider the two kinds of conflicts that can arise in a system in which distributed computations and long computations are rare. First, a mutator m1 may be restarted if there is an observer o1 serialized potentially after it. However, with our assumption that distributed computations are rare, only short mutator computations are likely to be restarted and the cost of restarting a short mutator computation is small. Second,

an observer **o2** may be delayed if there is a tentative mutator **m2** serialized potentially before it. If **m2** is invoked by a short computation, the cost of waiting for **m2** to be finalized is small. If **m2** is invoked by a long computation, a possible solution is to restart the computation that invokes **o2** with a smaller timestamp. If long computations are rare, we may expect the execution of two long computations to seldom overlap with each other. Hence, given that **m2** belongs to a long computation, we may expect **o2** to be invoked by a short computation most of the time and the cost of restart of **o2** is small. However, restarting a computation with a smaller timestamp is not always possible as the computation may invoke mutator operations. Hence short computations that invoke both observer operations and mutator operations may have to incur a high cost in being delayed by a long mutator computation.[21]

In a system where objects support only read/write operations, it is unreasonable to expect that short computations would invoke either only observer operations or only mutator operations. In a system where objects support abstract operations, this expectation is more likely to be valid. If the system also has the characteristic that distributed computations and long computations are rare, the hierarchical algorithm can be used to minimize costly conflicts. The hierarchical algorithm is also preferable to the dynamic algorithm because an incomplete long computation, though infrequent, can cause many other subsequently started short computations to be delayed.

5.2.2 Time-Range Concurrency Control Algorithm

The time-range algorithm we are going to describe is similar to the dynamic timestamp allocation protocol described by Bayer in [4] but with several important differences. We will describe Bayer's algorithm first and then the differences.

---

[21] Long computations that invoke both observer operations and mutator operations are less likely to be delayed by other long computations because we expect an overlap of execution of two long computations to be rare.

Bayer's Algorithm

In Bayer's algorithm each computation is associated with a time range (t1, t2) such that if the upper time bound of a computation a is less than or equal to the lower time bound of another computation b, then a is serialized before b. These time ranges can be shrunk dynamically but not expanded. The range will be shrunk to a single unique value when the corresponding computation is finalized. The upper time bound can initially assume the value infinity while the lower bound can assume negative infinity. It should be noted that for external consistency reasons, a computation probably should not be started with a lower time bound much smaller than the current time.

The static and dynamic algorithms are obvious special cases of this algorithm. The static algorithm starts with a time range with a single value. The dynamic algorithm has each computation associated with a time range in which the lower time bound is the current time, and the upper time bound is infinity, since the locked point of the computation can happen any time between the current time and the indefinite future.

The utility of this algorithm lies in its ability to shrink the time ranges dynamically so that conflicts can be avoided. For example, if computation a has a time range of (t1, t2) and computation b has a range of (t3, t4), then a can be serialized after b by raising t1 or shrinking t4 until t1 is greater than or equal to t4. Obviously this is not possible when t2 is less than or equal to t3. In those cases shrinking is disallowed and a has to be restarted if it is trying to invoke a mutator operation and b has invoked an observer operation.[22]

Our Time-Range Algorithm

In our time-range algorithm, time ranges are extended to a more general form:

---

[22]Parallel sub-actions can be serialized by sub-dividing the time range of the parent action into non-overlapping time rangee.

(max(L1, L2, ..., Lm), min(U1, U2, ..., Un))

where Li and Ui can be either a constant real number or a computation identifier. In the algorithm that we have described above, there is no way to ensure that a computation a will be serialized before/after b by shrinking a's time range if b's lower/upper time bound is negative infinity/positive infinity and cannot be changed[23]. To overcome this limitation, we allow the computation identifier of b to appear in a's lower/upper time bounds, which implies that b must be serialized before/after a. Initially a's time range can start with (negative) infinity or a constant in its upper or lower bound. The time range can be shrunk and computation identifiers of other computations, such as b's, can be added to ensure particular serialization order relationships.

We assume that each computation is associated with a site, called its *coordinator*, that keeps track of the final timestamp value of that computation. When b is finalized and the time range of b is shrunk to a single constant value, the sites that keep copies of a's time range can request this value from b's coordinator and replace the computation identifier with the constant. We call this process the *binding* of the computation identifiers. We will discuss how binding information can be propagated later.

To make sure that the time range is not empty, i.e. the lower bound is smaller than the upper bound, a computation should not commit until all the computation identifiers in its time range are bound. Any computation with an empty time range is aborted. This rule guarantees that if a cycle of serialization orderings is formed with each computation in the cycle assumed to be serialized before the next computation, at least one of the computations in the cycle will be prevented from committing. This is because one of the computations in the cycle must have an empty time range.

When a computation is aborted, infinity can be assigned to its computation identifier

---

[23]Changing b's time bound may involve sending messages to other sites and require a long delay.

if it is used as an upper time bound, or negative infinity if it is used as a lower time bound. This rule implies that when a computation **a** is to be serialized after two other computations **b** and **c**, **a** must include both **b**'s and **c**'s identifiers in its lower time bounds, even when **b** is constrained to be serialized after **c**. If **a** includes only the computation identifier of **b** in its time range and **b** is aborted later, the serialization ordering between **a** and **c** is expressed in neither **a**'s nor **c**'s time range.

When a cycle is formed, two different scenarios may happen. In the first scenario, some of the computations in the cycle will commit and at least one of the other computations will discover that it has an empty time range. For example, if the time ranges of the computations **a**, **b**, and **c** are as follows:

a: (t1, t2)
b: (max(a, t3), min(c, t4))
c: (t5, min(a, t6))

assuming that **t1 < t2**, **t5 < t6**, and the system chooses a final timestamp value for **a** in **(t1, t2)** that is larger than **t5**, **a** and **c** will be committed eventually but **b** will be aborted because **c**'s final timestamp value is less than **a**'s.

Deadlock Resolution

In the second scenario, a deadlock will develop, such as when:

a: (b, t1)
b: (a, t2)

A deadlock detection algorithm can be used to abort one of the computations in the cycle. However, not all deadlocks represent a cycle in the serialization orderings. For example, we may have:

a: (b, t1)
b: (t2, a)

where **t1 > t2**. In this example, **a** is assumed to be serialized after **b** and **b** is assumed to be serialized before **a**. These assumptions are obviously compatible and a serialization order is not ruled out by them. However, a deadlock is developed because both **a** and **b** are waiting for the other to finalize.

To avoid aborting any computation when these deadlocks occur, we can rely on the deadlock detection algorithm to switch the "direction" of waiting. For example, if **a** appears in the upper time bound of **b**'s time range and hence **b** is waiting for **a** to finalize, **b**'s computation identifier can be added to **a**'s lower time bound and then **a**'s computation identifier can be removed from **b**'s time range and replaced with the upper time bound of **a**. In our previous example:

$$\mathbf{a: (b, t1) \rightarrow (max(b, b), t1) = (b, t1)}$$
$$\mathbf{b: (t2, a) \rightarrow (t2, t1)}$$

The switching preserves the correctness of our algorithm because at least one of **a** and **b** is waiting for the other to finalize at all times. After the switch, **a** is waiting for **b** instead. In our example, **b** can proceed with its commitment and **a** can be committed if **t2** is less than **t1**.

To avoid having switchings that nullify one another's effects and to ensure that the deadlock will be resolved eventually, the switching can be limited to one direction. For example, we can limit the algorithm to remove computation identifiers only from upper time bounds and insert then only into lower time bounds. To avoid *creating* deadlocks with the switching when there are not any, identifiers can only be removed from the upper time bounds if there are not any other computation identifiers in the lower time bounds. In other words, the removal should allow the computation to commit. In the previous example involving computations **a**, **b**, and **c**, we will never have:

$$\mathbf{a: (t1, t2) \rightarrow (max(c, t1), t2)}$$
$$\mathbf{b: (max(a, t3), min(c, t4)) \rightarrow (max(a, t3), t4)}$$
$$\mathbf{c: (t5, min(a, t6)) \rightarrow (max(b, t5), t6)}$$

Binding Computation Identifiers

To make sure that every computation identifier used by a time range will be bound eventually, we have to make sure that the final timestamp value of a committed computation **c** (we will discuss aborted computations later) will be sent to each site that it had visited. In addition, since other computations that had visited those sites

might have included c's computation identifier in their time ranges and caused it to appear in other sites, the final timestamp value of c has to be propagated to those other sites as well. Notice that because of the indirect propagation of c's identifier, the sites visited by c may not overlap with the set of sites visited by another computation d that has c in its time bounds.

In order to make sure the computation identifiers can be bound eventually, we assume the coordinator of each computation c remembers the following in stable memory when c commits:

1. c's final timestamp value,
2. a list of all the sites that c had visited,
3. the computation identifiers that c had used in its time bounds.

After commitment, the coordinator will send c's timestamp value to all the sites that c had visited, which can be piggybacked on the messages that the coordinator uses to convey the outcome of c (see section 5.4). The coordinator will also try to find out the final timestamp values of the computation identifiers that c had used and send those values to the sites that c had visited. This is necessary as other computations may have learned those computation identifiers from c. Only then all these messages are acknowledged can the coordinator discard the information that it had stored during commitment. At each site being visited by c, each copy of the computation identifiers, if there is more than one, will be replaced with the final timestamp value before acknowledgment.

To see that every computation a_n that has the computation identifier of a computation a_1 in its time range will eventually learn of the final timestamp value of a_1, consider the path of computations a_1, a_2, ..., a_n along which a_n learns about the computation identifier of a_1. (The computation a_2 accesses an object accessed by a_1 and includes a_1's identifier in its time range. Then a_3 accesses an object accessed by a_2 and includes a_1's identifier in its time range. Eventually a_n accesses an object accessed by a_{n-1} and includes a_1's identifier in its time range.) Note that each pair of adjacent computations on this path visited some site in common. Since

the coordinator of a_1 makes sure that each site that it visited learns about its final timestamp value, the coordinator of a_2 can find out a_1's final timestamp value from the shared site, where a_2 first learned about a_1's computation identifier. Similarly, after a_2 sends that value to every site that it had visited, a_3 can learn about the value from the shared site between a_2 and a_3. The process is repeated until a_n learns about a_1's final timestamp value.

A complication arises when some of the a_i's are aborted. In the algorithm that we described above, a_n will be waiting indefinitely for the final timestamp value of a_1. A solution is for a_{i+1} to remember a list of all the sites and the name of the actions (e.g., the name of a_i) from which it has learned about a_1 along with the name of a_1 in stable memory when it commits. Instead of waiting for a_i to propagate the final timestamp value of a_1, a_{i+1} can send queries to each of the sites in the list. If records about those actions from which a_{i+1} learns about a_1 cannot be found in any of the sites in the list and none of those sites is in the process of sending out a_1's final timestamp value, a_{i+1} can propagate the value of positive infinity to a_{i+2} if a_1's identifier is used as an upper time bound by a_{i+1}, or negative infinity if it is used as a lower time bound. This is because the serialization constraint is established between a_i and a_{i+1}, instead of between a_1 and a_{i+1}. This solution is correct only if a_{i+1} limits the propagation of the infinity value to a_{i+2} and not to any other computation that happens to use a_1's identifier. So when a site receives an infinity value from a_{i+1}, it should bind an a_1 identifier in its memory only when the identifier has been learned from a_{i+1}.

Privileged Computations

In the rest of this section we will describe an optimization that will allow the computations in the system to have different priorities. In particular, we can use the optimization to make mutator computations less likely to be restarted. Suppose there is a class of computations with the following form of time ranges:

$$(\max(L1, L2, \ldots, Lm), \infty)$$

and the property that the identifiers of these computations are not allowed to be used in the lower time bounds of any computation. Consequently, the final timestamp values of these computations are never required to be smaller than any other value, and since they have no upper bound, we can always find real constants that exceed their lower time bounds. In other words, these computations can commit even when there are unbound computation identifiers in their lower time bounds. Choosing final timestamp values for these computations has to be delayed until the unbound computation identifiers are bound, however.

These computations are "privileged" because they can always avoid being restarted by including the upper time bounds or the identifiers of other computations (except those in the privileged class) in their lower time bounds. It should be noted, however, that a privileged computation may still be delayed due to tentative mutators that are serialized potentially before itself.

Because privileged computations can commit without binding their time ranges, a deadlock involving committed computations can be developed. Because of the restriction that identifiers of privileged computations cannot be used in the lower time bounds of other computations, a deadlock must involve non-privileged computations, which must be uncommitted and can be chosen as victims to be aborted.

The time-range algorithm is useful in a system where the only long computations are mutator computations. By assigning the long mutator computations as privileged computations, the mutator computations can avoid being restarted by other observer computations. Mutator computations are also not delayed by tentative computations because they do not observe any state. Short observer computations in the system can avoid being delayed by tentative long mutator computations by restarting with smaller timestamps. The cost of restart is low. However, this may not be possible if a short computation invokes both observer and mutator computations. Compared to other multi-version algorithms [9, 48], our algorithm has the advantage that the long mutator computations are never restarted by the concurrency control algorithm. The

cost of our time-range algorithm lies in the complexity of manipulating time ranges and sending messages around to bind the computation identifiers.

Examples of applications that have only long mutator computations and short observer computations are databases that are replicated on many sites for availability and efficiency of observer operations. Mutating the state of one of these databases is a long computation because of the large number of replicas. Frequently, a mutator computation also does not observe the state of the database, such as when old data values are overwritten with new data values. On the other hand, usually only short queries are directed at database because most data is available from the local site.

5.3 Making Concurrency Control Algorithms Transparent

In the previous two sections we have described various concurrency control algorithms. We have shown that under special situations concurrency control algorithms can be adapted to minimize costly conflicts. For example, in the case of the hierarchical algorithm, long computations would not suffer from repeated restarts when they are rare.

Given that different concurrency control algorithms might be appropriate in different applications, we have designed a programming interface which hides the concurrency control algorithm used underneath. The history operations, such as *p_sub* or *d_prior*, make the algorithm in which the serialization order is determined transparent. The `retry` statement also makes the actions that need to be taken when a conflict arise transparent.

This section describes how to implement such a programming interface given a particular concurrency control algorithm. In section 5.3.1 we will describe the implementation of the history operations that capture the serialization order. In section 5.3.2 we will describe the implementation of the `retry` statement.

5.3.1 Implementation of History Operations

In this section we will describe how the history operations *p_sub*, *p_prior*, *d_sub*, and *d_prior* can be implemented given that a static or a dynamic concurrency control algorithm is used. Our goal is to show that these operations can be implemented and our descriptions will not focus on efficiency. The operation *p_between* can be implemented by filtering a history object with *p_prior* and *p_sub*. *D_between* can be implemented with *d_prior* and *d_sub* similarly.

Figure 5-1 defines the subset of transitions that should be returned by the `sub` and `prior` operations for a static and a dynamic concurrency control algorithm. In the dynamic algorithm, we assume that each transition is labelled with two timestamps from a Lamport clock [27]: an *operation timestamp* and a *commit timestamp*. The operation timestamp is read immediately before the corresponding operation returns, and the commit timestamp is read when the computation commits. For the operation being executed currently, the current clock value can be used as its operation timestamp. We assume that these timestamps are remembered in a history object. A commit timestamp can be piggybacked on a message that informs a site of a computation's outcome and recorded in a history object when the computation's status in the history object is updated.

Implementations for other concurrency control algorithms are similar to those in figure 5-1. For example, the implementations for the hierarchical algorithm and the static algorithm are the same except that the two timestamps for a computation are concatenated for comparison in the former and a single timestamp is used in the latter. In an implementation for the time-range algorithm, a transition can be serialized potentially before or after another transition if their time ranges can possibly overlap. Otherwise, one of them is serialized definitely before the other.

Static serialization concurrency control algorithm:
(Implementations for the d_ *counterparts are identical.)*

```
p_sub = procedure(h: history, t: transition) returns(history)
        return transitions in h with larger timestamps than t
        end p_sub

p_prior = procedure(h: history, t: transition) returns(history)
        return transitions in h with smaller timestamps than t
        end p_prior
```

Dynamic serialization concurrency control algorithm:

```
p_sub = procedure(h: history, t: transition) returns(history)
        if t has a commit timestamp c
           then return all finalized transitions that have larger
                commit timestamps than c and all tentative transitions
                in h
           else return all tentative transitions in h and all finalized
                transitions that have larger commit timestamps than
                the operation timestamp of t
           end
        end p_sub

d_sub = procedure(h: history, t: transition) returns(history)
        if t has a commit timestamp c
           then return all finalized transitions that have larger
                commit timestamps than c and all tentative transitions
                that have larger operation timestamps than c in h
           else return an empty set
           end
        end d_sub

p_prior = procedure(h: history, t: transition) returns(history)
        return (all transitions in h - d_sub(h, t))
        end p_prior

d_prior = procedure(h: history, t: transition) returns(history)
        return (all transitions in h - p_sub(h, t))
        end d_prior
```

Figure 5-1:Implementations for Sub and Prior

5.3.2 Implementation of Retry Statement

This section describes how the `retry` statement can be implemented given a concurrency control algorithm. In particular, we will use a static algorithm as an example. We will also describe implementations for other concurrency control algorithms, although more briefly. Our description of implementations will focus on their feasibility, but brief references to efficiency will be made occasionally. We will first present an example of the kind of decision making that is involved in the execution of a `retry` statement. Then we will describe an implementation.

When a `retry` statement is executed in a system with a static algorithm, the language system should decide whether the computation executing the statement should be delayed or restarted, and if it is delayed, when it should be rescheduled. With a dynamic algorithm, the only possibility is to delay a computation. The only decision is when to reschedule a computation. With a time-range algorithm, the decisions are more complicated. A computation can be delayed, restarted, or have its time range shrunk in different ways.

When the system is faced with these decisions, there are no optimal decisions without knowledge of the future. Heuristics are needed to determine the relative likelihood of correctness and cost of each of the choices. For example, it is reasonable to expect that a tentative transition is more likely to commit than to abort and make decisions accordingly. We will also assume that it is unlikely to have an operation invoked in the future serialized before some existing operations.

An Example

Consider the proceed condition

```
~history$exists(history$p_sub(this_transition), no_x,
                not_changed(del_x))
```

in the `insert` procedure in figure 4-3 where

```
not_changed = procedure(op: template, t: transition) returns(bool)
        return(history$exists(history$d_between(this_transition, e)),
                committed_del_x))
        end not_changed
```

Suppose the proceed condition evaluates to **false** and a transition **t** is the only **no_x** transition that causes the condition to be false. The proceed condition will be satisfied when either:

1. **t** is aborted, or
2. **t** is serialized definitely before the invoked operation, or
3. a **committed_del_x** transition is serialized definitely between the invoked operation and **t**

Item 1 is unlikely to happen, regardless of the concurrency control algorithm used.

Suppose a static concurrency control algorithm is used. Item 2 will only happen with a restart because the predetermined serialization order does not change. Item 3 is only likely to happen if there is already an tentative **del_x** transition serialized between the invoked operation and **t**. In those cases, the invoked operation can be delayed until the **del_x** transition is finalized. In other cases, the invocation request should be refused and the computation that invokes it restarted. Although it is possible that the restart is unnecessary after all, it is the most appropriate choice under our assumptions.

If a dynamic concurrency control algorithm is used, delaying the current operation cannot cause item 3 to become true. In fact, it would achieve the opposite effect. Also, a **del_x** transition may not exist after all. Item 2 can be fulfilled by delaying the current operation until **t** is finalized, the most appropriate step to take in this case.

If a time-range concurrency control algorithm is used, the system may have several choices. The time range of the current computation may be shrunk, if necessary and possible, in such a way that item 2 is satisfied. If a **committed_del_x** transition exists and it is definitely serialized before **t**, the time range of this action may be shrunk such that item 3 is satisfied. If a **tentative_del_x** transition exists and it is serialized potentially before **t**, the current operation can be delayed until the serialization order

is known and the transition committed.

An Implementation for a Static Algorithm

Since we have limited our proceed conditions to be constructed with boolean operations and history operations, program analysis can be used to decide the action to take when a `retry` statement is executed. The goal of the program analysis is to determine whether a proceed condition is *likely* to be satisfied eventually without a restart. Only when should an operation be delayed. We assume that when a system is confronted with a choice of delaying or refusing an operation, delaying is preferred. A more sophisticated decision can be based on the expected costs of the delay and the restart.

Choosing Between Delay and Restart

To shorten our presentation, assume that a boolean operation can be either `and`, `or`, or `negate`. However, we would eliminate all the `negate` operations that are not immediately applied to the result of an `exists` operation by making suitable program transformations. For example, if a proceed condition is of the form:

```
~(exists(h, t, p) and exists(h', t', p'))
```

we change it to:

```
~exists(h, t, p) or ~exists(h', t', p')
```

If a proceed condition `c` is of the form:

1. `c1 and c2`: then `c` is likely only if both `c1` and `c2` are likely.

2. `c1 or c2`: then `c` is likely only if at least one of `c1` and `c2` is likely.

Other than the two forms above, `c` can also be of the form `exists(h, t, p)` or `~exists(h, t, p)` where `h` is a history object, `t` is a transition template, and `p` is a procedure. In order to allow the language system to determine the likelihood of an `exists` or `~exists` expression, we limit `p` to be of the form:

```
p = procedure(arg: transition) returns(bool)
        return(e)
        end p
```

where `e` is subjected to the same restrictions as proceed conditions. If `c` is of the form:

1. `exists(h, t, p)`: then `c` is likely only if `t` is of the form `committed_op_...`, and there is a transition `tr` in `h` that matches `op_...` and `p(tr)` is likely.

2. `~exists(h, t, p)`: then `c` is likely only if for all the transitions `tr` in `h`, either a committed version of `tr` does not match `t` or `~p(tr)` is likely.

Determining whether an `exists` expression is likely involves searching the history object `h` at run time. The same process can be used to determine whether `p(tr)` or `~p(tr)` is likely after replacing references to `arg` in `e` with `tr` at run time.

Determining Reschedules

Given that a proceed condition is likely to be satisfied without a restart, the language system should determine when the current invoke request should be rescheduled. In other words, the language system should determine when the proceed condition *becomes likely*.

There are many options for determining what kinds of events and processing are allowed to trigger the rescheduling of a suspended operation. For example, a simple scheme is to allow only the finalization of a fixed set of transitions determined at the execution of the `retry` statement to trigger rescheduling. A more complicated alternative is to also allow the finalization of subsequently invoked transitions and evaluation of arbitrary expressions to determine when rescheduling is appropriate. Since the goal of this section is to show the feasibility of an implementation that can resolve a conflict in a reasonable, but not necessarily optimal, fashion, we will use the simpler scheme. Another reason to use the simpler scheme is to minimize the cost of scheduling. One of the necessary consequences of using the simpler scheme is that we cannot guarantee that a proceed condition will be met when an operation is rescheduled, as some other operations may have executed between the suspension and the rescheduling. However, this is considered acceptable by our programming interface.

Suppose s is a set of transitions such that the finalization of a transition in s should trigger the rescheduling of an operation with a proceed condition c. A program analysis similar to the one above can be used to determine a non-empty s. If c is of the form:

1. `c1 and c2`: then s is the union of the sets that trigger c1 and c2.

2. `c1 or c2`: same as above.

3. `exists(h, t, p)`: if tr is a tentative transition in h such that if it is committed, it would match t and `p(tr)` would return `true`, then tr is in s.[24]

4. `~exists(h, t, p)`: if tr is a tentative transition in h that matches t and `p(tr)` returns `true`, then tr is in s.

All Delayed Operations are Rescheduled Eventually

To show that this implementation is correct, in the sense that if an operation is delayed, it will be rescheduled eventually, we need to show that s is not empty. We will now describe an informal argument showing that it is indeed the case.

Recall that a well-formed proceed condition satisfies the following requirements:

1. The proceed condition should be satisfied if:
   a. new operations are not started, and
   b. all current operations in the system, except the one being considered, are finalized and the outcomes are known by all history objects, and
   c. the operation being considered is serialized after all existing transitions and the serialization order among existing transitions are known.
2. It is not satisfied currently.
3. It is constructed with boolean operations and the operations provided by the history objects.

Given that a proceed condition c is not satisfied currently, there must be some

[24] More accurately, the *commitment* of tr is in s, since aborting tr would not make c become likely.

`exists(h, t, p)` or `~exists(h, t, p)` expressions not satisfied currently. Given that `c` is not restarted and hence likely to become satisfied eventually, at least one of these expressions is likely to become satisfied eventually. Suppose an `exists(h, t, p)` expression is likely to become satisfied eventually. Following our definition of when `exists(h, t, p)` is likely, we know that there is a transition `tr` in `h` such that either:

1. `tr` does not match `t` because `tr` is tentative, or
2. `p(tr)` is not satisfied currently but is likely to be satisfied eventually.

Given our rules for adding transitions to the triggering set, `tr` will be in the set if the first case is true. If the second case is true, induction can be used to argue that the program analysis of `p(tr)` will lead to the addition of some transitions in the triggering set. A similar argument can be used when an `~exists(h, t, p)` expression is not satisfied currently but likely to become satisfied eventually.

Discussion

In addition to guaranteeing that an operation will be eventually rescheduled if it is delayed, there is also a performance issue that unnecessary restarts should be avoided. This is achieved with the first requirement for the well-formedness of proceed conditions. By requiring a proceed condition to be satisfiable given the conditions 1.a, 1.b, and 1.c, we prevent an application from specifying a proceed condition which is unlikely to become satisfied when in fact an operation is likely to be able to proceed eventually.

For systems that use a dynamic or time-range concurrency control algorithm, rules similar to those above can be used to determine whether to restart or delay an operation, and, if the operation is delayed, when it is rescheduled. Correctness in the sense that a delayed operation is eventually rescheduled is not difficult to achieve as long as every delayed operation is rescheduled occasionally. The complexity of an implementation is in determining which set of events should trigger rescheduling and whether restart, delay, or some particular way of shrinking a time range should be employed. It is debatable whether a programmer or a language system

implementation can make better decisions. For example, we have discussed that the relative merits of restarts and delays depend on their expected costs. Having a language implementation calculate these costs avoids cluttering a program with optimizations. However, one may argue that a programmer has a better knowledge of these costs.

5.4 Commit Protocols

When a distributed computation commits or aborts, the sites that participated in the computation have to agree on its outcome. At any time during the process of reaching an agreement, site crashes or communication failures can occur. Once a computation is committed, each site should make sure that the computation would appear to have executed despite site crashes and communication failures. The sites that participated in an action should also be informed of the action's outcome as soon as possible, so that other actions will not be delayed. The protocol followed by the sites to reach agreement is called a *commit protocol*.

Section 5.4.1 reviews the *two-phase commit protocol* [17]. Section 5.4.2 describes an alternative, the *one-phase commit protocol*, and compares the two. We argue that the one-phase commit protocol is more suitable in our environment. In the description of these protocols, we assume that *call* and *return* messages are used to invoke processing on remote sites and to return results of those efforts.

5.4.1 Two-Phase Commit Protocol

The most common commit protocols used by distributed systems are two-phase commit protocols. In a two-phase commit protocol, one of the sites plays the role of a *coordinator* and the other sites become *subordinates*. We assume that the site that initiates the top-level action plays the coordinator role, and the other sites that have participated in the computation are subordinates.

At the end of the computation, if commitment is desired, the coordinator will send

prepare messages to the subordinates and wait for their replies. In a system with nested actions, only the subordinates with non-aborted sub-actions need to receive these *prepare* messages. At the subordinates, a *yes* vote is returned if commitment is desired. A *no* vote is returned otherwise. Before a *yes* vote is returned, the subordinates can decide to abort the computation unilaterally. In those cases, a *no* vote can be returned when *prepare* messages arrive.

At the coordinator, if all the votes are *yes* votes, the computation can be committed by writing the decision to stable memory atomically. Afterwards, *commit_computation* messages will be sent to the subordinates. If any of the votes returned is a *no* vote, or the coordinator has given up waiting for all the votes to return, *abort_computation* messages can be sent to the subordinates that had sent *yes* votes. *Abort_computation* messages can also be sent anytime during the execution of the computation. A parent action can also send *abort_action* messages to abort sub-actions before the end of a computation.

Commit_computation messages and *abort_computation* messages are mutually exclusive. A computation should never send both types of messages. Through the *commit_computation* and *abort_computation* messages, the subordinates will learn that the computation is finalized. The sending of *prepare* and vote messages is the first phase, and the sending of *commit/abort_computation* messages the second.

When sending messages to the subordinates, either the coordinator can send to each subordinate directly, or the messages can be relayed by other subordinates. A convenient strategy is to have the site of a parent action relay the messages to its sub-actions [37]. The first messages are sent by the site that executes the top-level action, the coordinator of the computation. The strategy is convenient because each parent action knows the names of its sub-actions, where they are executed, and whether they should be aborted or committed. However, having the coordinator send the messages directly avoids any delay in relaying. To do so, each action should include the names of its sub-actions, where they are executed, and whether

they should be aborted or committed when it returns to its parent. In this way, the top-level action will collect all the necessary information to send the messages directly.

5.4.2 One-Phase Commit Protocol

An alternative to the 2-phase commit protocol is a 1-phase commit protocol. In the 1-phase commit protocol, no *prepare* or vote messages are sent. A site is prepared to commit when it sends a return message. It stays prepared until notified by the coordinator to commit or abort. The 1-phase commit protocol takes one less round-trip delay to finish. In a system with long communication delays, this is an important savings. In a simple 2-site distributed computation using a 2-phase commit protocol, the coordinator and the subordinate are informed of the outcome of the computation after 2 and 2.5 round-trip delays respectively. With a 1-phase commit protocol, the delays are reduced to 1 and 1.5 round-trips respectively.

One of the advantages of the 2-phase commit protocol over the 1-phase commit protocol is that a subordinate retains the privilege to abort a computation unilaterally until it has responded *yes* to a *prepare* message. Presumably, by aborting an tentative computation, a site can recover the resources held by that computation.

It is not clear whether this window of vulnerability, during which a subordinate has to wait for a decision from its coordinator, is in fact shorter in a 2-phase commit protocol than in a 1-phase protocol. In a 2-phase commit protocol, the length of the window is at least the time required for a vote to travel to the coordinator and the decision to come back to the participant. In addition, assuming that most computations commit, the coordinator has to wait for all the votes before sending out the decision in most cases. In a 1-phase commit protocol, the length of the window is determined by the time required to execute the rest of the computation after a subordinate has returned plus the time needed for the coordinator to send a decision. If a site is accessed near the end of a computation and sending messages to sites accessed in the beginning of the computation from the coordinator leads to

long delays, then the site accessed near the end of the computation has a shorter window with a 1-phase commit protocol. On the other hand, the window is probably longer for sites accessed in the beginning of a computation if the computation accesses more than two sites serially. In a simple 2-site distributed computation the window of vulnerability is approximately a round-trip delay in length for both protocols if we ignore the time the coordinator uses to compute after it has received a reply from the subordinate. This period of computation should be negligible compared to the round-trip delay. The same argument can be applied to an n-site distributed computation in which the coordinator invokes the n-1 participants in parallel.

In a 2-phase commit protocol, by delaying the preparation of an action until the coordinator is ready to commit, there is a possibility that several actions' preparations can be piggybacked in a single write to stable memory. In a 1-phase protocol, a sub-action that executes in the same site as some of its ancestors can delay its preparation until the oldest ancestor returns because a site crash before its preparation would also abort the ancestor. Otherwise, it has to be prepared before it returns. The 2-phase commit protocol is more efficient if accessing stable memory is an expensive operation.

A compromise between the 2-phase and 1-phase commit protocols is to leave a choice in the protocol. When a subordinate returns, it can set a flag in the return message to indicate whether it has prepared. If it has not, the coordinator has to send a *prepare* message and wait for a *yes* vote from that subordinate before the coordinator can commit. Meanwhile, the subordinate can piggyback its preparation with a later stable memory access; afterwards, as an optimization, it can send a *yes* vote to "catch up" with its return. In other words, the preparation can become an asynchronous process as long as it is performed before the computation is committed. In Chapter 7 we will discuss the use of checkpoints, of which an early preparation is a special case, to increase the resilience of a computation.

There are other commit protocols proposed in the literature. Skeen proposed a 3-phase non-blocking commit protocol in [52]. In addition to the extra delay, the assumptions about the communication network in his protocol are incompatible with our model. We believe that the 1-phase commit protocol is more appropriate in a system with long computations because of the reduced delay with one less phase of messages.

5.5 Summary

This chapter discussed how the programming interface described in Chapter 4 can be supported. In particular, we showed that it is possible to mask the concurrency control algorithm used in a system. We have described how history operations, such as *p_sub* or *d_prior*, and the `retry` statement can be implemented in different concurrency control algorithms. We have also proposed two novel concurrency control algorithms which minimize the likelihood of costly conflicts given that special conditions are met. We have described commit protocols briefly and described a 1-phase protocol which has a shorter delay between an action returning and its being finalized. A compromise between a 1-phase protocol and a 2-phase protocol using an asynchronous preparation allows the cost of accessing stable storage to be reduced.

Chapter Six

Power of Atomicity

In this chapter we compare our atomicity definition with other correctness definitions in which atomicity is abandoned. Atomicity is used in this thesis to model computations because it is easy to understand and reason about. We have also shown that the concurrency of a system can be increased by using semantics in an implementation. In particular, by incorporating the functionality of an application into the atomicity definition, our approach allows a trade-off between functionality and concurrency. However, if there were other correctness definitions which permitted more concurrency, the importance of concurrency might outweigh the simplicity of atomicity, especially in a system with long computations. In this chapter we will show that our atomicity definition permits as much concurrency as some non-atomic correctness definitions. On this basis, we will claim that our model of correct behavior is preferable, since in comparison it is equally powerful and easier to understand.

The class of correctness definitions that we use to compare against our atomicity definition is one in which the application defines explicitly pairs of transitions that "conflict." These definitions insure that computations that invoke conflicting transitions are executed in the same order at all objects. A representative of this class of correctness definitions can be found in [50]. A slightly different but similar correctness definition can be found in [38]. We will describe a correctness definition which is slightly more general than the one in [50]. We will call the definition we are going to describe the *consistency* definition.

As we have described earlier, the consistency definition insures that computations executing conflicting transitions are executed in the same order at all sites. For

example, suppose computation **a** executes two transitions **a1** and **a2** and computation **b** executes two transitions **b1** and **b2**. Furthermore, suppose **a1** conflicts with **b1** and **a2** conflicts with **b2**. The consistency definition requires that either **a1** precedes **b1** and **a2** precedes **b2**, or **b1** precedes **a1** and **b2** precedes **a2**. More precisely, the consistency definition can be defined with a graph acyclicity requirement. The nodes in the graph are computations. Two computations are linked by an edge if they execute a pair of conflicting transitions at an object. The direction of the edge is determined by the order of execution of the transitions. A history of transitions is said to be *consistent* if the graph is acyclic. A system that only generates consistent histories is called a *consistent system*.

An equivalent way of stating the same requirement is to require that there exists a total order among the computations in the system: If two computations **a** and **b** execute a pair of conflicting transitions at an object with **a**'s transition executed before **b**'s, then **a** is ordered before **b** in the total order. Notice that this total order is different from a serialization order in an atomicity definition, since only conflicting pairs of transitions are required to be ordered in this total order. Non-conflicting pairs of transitions can be ordered in different orders in different objects. There may be more than one such total order.

An example may help in the understanding of the consistency definition. Consider a banking account with **deposit_x_okay**, **withdraw_y_okay**, **withdraw_y_insuf**, and **read_balance_z** transitions. If the application does not define **read_balance_z** to be conflicting with **deposit_x_okay** or **withdraw_y_okay** transitions, then a transfer between two accounts, composed of a withdrawal and a deposit, can interleave with an audit attempting to find the sum of the balance in two accounts with two *read_balance* operations. In one of the accounts, the **read_balance_z1** transition may be executed before the **withdraw_y_okay** transition, whereas in the other account, the other **read_balance_z2** transition may be executed after the **deposit_y_okay** transition. In this example, the amount being transferred is counted twice by the audit. However, we must assume that this behavior is acceptable to the

application, since it does not choose to exclude it by the definition of conflicting transitions.

This behavior is typical of what real banking systems exhibit in practice. A transfer of funds between two accounts is done in two separate parts, certainly when the two accounts belong to two different banks and often when the accounts belong to different branches of the same bank. In the case of transfer by check, the deposit occurs first, and the withdrawal occurs only after the check has "cleared." The clearing of the check involves physical transport of the check and makes the entire transfer of funds a long computation. During the time the check clears, the money appears to be in two places, which is a way of saying that **read_balance_z** does not conflict with **deposit_x_okay** or **withdraw_y_okay**. People have attempted to take advantage of the inconsistency by investing the double-counted money in various ingenious ways. The banks have not corrected this problem by imposing atomicity across the Federal Reserve System; rather, they tolerate the problem to a degree and control abuses by regulation and law. The builders of banking systems appear to believe, as a practical matter, that the imposition of a total ordering among all the computations would produce intolerable loss of concurrency.

The consistency definition may seem more powerful than atomicity because an application can specify conflicting transitions explicitly. However, we will show that atomicity is at least as powerful as the consistency definition. In the banking example above, we can show that by defining the functionality of the *read_balance*, *withdraw*, and *deposit* operations appropriately, the behavior described above can be modelled by our atomicity definition. Our proof does not make the transfer of funds into a short computation, nor does it enable the audit computation to predict whether a check will clear and to return accurate and up-to-date answers. However, by casting the uncertainty in the answers returned with an atomicity model and providing the same level of concurrency as a consistency system, we provide a simpler model to understand the behavior of an application than the consistency definition. The better understanding in turn provides a better framework for the users to deal with the

inconsistencies that they might observe. Thus the power of atomicity that we show is of more than academic interest.

Our proof is by construction. We show that given any system of objects, their transitions, and a set of conflicting transitions, we can construct a system with an "equivalent" set of objects, an "equivalent" set of transitions, and serial specifications for the equivalent objects, such that the set of consistent histories is identical to the set of "equivalent" atomic histories. Consequently, the two systems have the same behavior and concurrency. The equivalence is defined with mappings from one system to the other. The mappings can be used to "simulate" one system with the other.

The problem with the "equivalent" atomic system that we construct is that its serial specifications are too complicated to maintain our claim that atomicity is easy to understand. Hence our proof only shows that atomicity is at least as powerful, but not always easier to understand. We show a second result in this chapter. We show that for a class of objects atomicity is as powerful *and* easier to understand. We also argue that this class of objects is a large class.

Section 6.1 presents an informal version of our proof that atomicity is at least as powerful as the consistency definition. Section 6.2 defines atomicity and consistency with more formal notations and presents a formal version of the same proof. Section 6.3 defines a class of objects called *accurate* objects and shows that atomicity is as powerful and easier to understand for accurate objects. Although some objects in a system may not be accurate, modelling the behavior of the non-accurate objects with atomicity allows the behavior of the accurate objects to be understood more easily than with a consistency definition. If we abandon atomicity in the non-accurate objects, we abandon atomicity in the accurate objects also.

The correctness requirements to handle situations in which failures can happen are usually not specified clearly in the consistency definitions in the literature. However, failure atomicity can be incorporated into these definitions in a straightforward

manner: only committed transitions are considered in determining whether a history is consistent. We will ignore failure atomicity in our proofs and assume that all transitions will be committed. The addition of failure atomicity, which is orthogonal to the serializability and consistency concepts, does not change our results.

6.1 Informal Proof of Power of Atomicity

Recall that conflicting transitions are required to be executed in a total order of the computations in a consistent system. We will call this total order a *consistent order*. Similar to an atomic system, a concurrency control algorithm is needed to determine a global consistent order followed by every object in the consistent system. Also, just as in an implementation of an atomic system, *conflicts* can be created when, for example, there is insufficient knowledge of the consistent order. Using the terminology of the conflict model developed in this thesis, a conflict is created by a new transition when there are other transitions that have the following properties:

1. these transitions are conflicting with respect to the new transition, and
2. they are potentially ordered after the new transition according to the global consistent order.

If no conflicts are created, an object can proceed to determine the result to be returned. In a consistent system, the result is computed based on the order in which transitions are executed in an object, which we will call the *local execution order*.

The core of our proof is to construct an equivalent atomic system in which conflicts are created at the same situations and the same results are returned when there are no conflicts. Since conflicts are created at the same situations, the atomic system has the same level of concurrency as the consistent system. Since the same results are returned, the atomic system has the same "behavior." More rigorously, since the conflict conditions and the validity of results in an atomic system are determined by the serial specifications, we need to construct serial specifications that guarantee that a history in the atomic system is atomic if and only if the equivalent history in the consistent system is consistent.

Before describing these serial specifications, we will describe how the atomic objects in the equivalent atomic system can be implemented. Presumably, one can argue that the same implementation that implements the objects in the consistent system can be used to implement the atomic objects. However, we will describe an implementation using the mechanisms that we described in Chapter 4, which may help in understanding the equivalence between the atomic system and the consistent system.

Just as in the implementations in Chapter 4, each transition executed at an atomic object is recorded in a history object. When a new operation is invoked, the history object is queried to determine whether there are previously invoked conflicting transitions that can potentially be serialized after the new transition. If there are, a conflict is created and has to be resolved. If no conflict is created, the implementation has to determine a valid result to return. Since results are computed according to the local execution order in a consistent system, the results in the atomic system should be computed in the same way. In a practical implementation, the transitions in the history object should be merged according to the local execution order, so that the snapshot/projection object can be used to determine the result efficiently. The local execution order has to be encoded in the transitions so that they can be merged accordingly.

We will now describe the serial specifications for the objects in the atomic system that create the same conflicts as the objects in the consistent system. Suppose that in a consistent system a transition **t1** is executed before another transition **t2** in an object **o** and **t1** and **t2** are a pair of conflicting transitions. From our definitions, **t1** must be ordered before **t2** in any consistent order. If we can make sure that, for their equivalent transitions **t1'** and **t2'**, **t1'** must be ordered before **t2'** in any serialization order, then a serialization order exists *only if* a consistent order exists. Also, if the ordering of any such pairs of **t1'** and **t2'** is the only requirement on a serialization order, then a serialization order exists *if* a consistent order exists. To make sure that **t1'** is ordered before **t2'** in a serialization order, we can require the collection of

conflicting transitions that are *executed* before **t2'**, such as **t1'**, to be *serialized* before **t2'**. To express this requirement in the serial specifications, we can encode this collection of transitions in **t2'** and compare this collection with the collection of transitions that are serialized before **t2'**. Since a serialization order exists if and only if a consistent order exists, conflicts are created under the same situations.

An additional requirement on the serial specifications of the equivalent atomic objects is needed. In addition to guaranteeing that conflicts are created under the same situation, we must also require that the results returned in the equivalent atomic system are those returned by the consistent system. In a consistent system, the validity of a result is determined by specifications like the serial specifications. For example, if a *withdraw* operation returns *okay*, then there must be enough deposits executed before the *withdraw* operation to cover the withdrawal. Since the serialization order, though identical with the consistent order, may not be the same as the local execution order, we cannot use the serialization order to compute the results. In other words, the validity of the results in the atomic system should be determined with the local execution order instead of the serialization order. Consequently, an additional requirement on the serial specifications is that each transition should encode the sequence of previously invoked transitions in the local execution order and ensure that this transition's result is valid according to that order.

Since the concurrency levels in the two systems are the same, and the results returned are identical with the exception that a sequence of previously invoked transitions have been encoded in the transitions generated in the atomic system, we claim that the two systems have the same behavior. The same implementations can be used to implement the two systems. The only difference between the two is the modelling of the acceptable behavior of the system.

6.2 Formal Proof of Power of Atomicity

This section presents a more formal version of the argument described in the last section. Atomicity and consistency are defined more formally in sections 6.2.1 and 6.2.2. The formal proof is in section 6.2.3.

6.2.1 Atomicity

Some terminology is needed before presenting the definition of an atomic system. Suppose **h** is a sequence of events, **r** is an object, and **a** is an action. We define **h|r** to be the subsequence of **h** involving **r** and **h|a** to be the subsequence of **h** involving **a** but not **a**'s sub-actions. An event in a sequence **h** is committed if there is a commit event of the same action identifier in **h**. We define **committed(h)** to be the subsequence of **h** that involves only invoke and return events that are committed. **Aborted(h)** is defined similarly. The sign "||" denotes concatenation of sequences. We will omit the concatenation signs for sequences whenever it is convenient. For example, $t_1t_2...$ refers to $t_1||t_2||...$. Also, we will use the "$\in$" sign to refer to an element being part of a sequence. So for example, we say $t_2 \in t_1t_2...$.

A sequence of events **h** is *well-formed* if it satisfies the following conditions:

1. Ignoring commit and abort events, the subsequence **h|a** should have alternating invoke and return events, starting with an invoke event, and with each pair involving the same object.

2. **committed(h)** and **aborted(h)** do not have any common events.

3. If a commit event of an action **a** appears in **h**, then **h|a** consists of an alternating sequence of invoke and return events (starting with an invoke event and ending with a return event) and some commit events at different objects.

A well-formed sequence of events is called a *history*.

We define a function **Serial** which takes a history and a linearization of the actions in that history as inputs, and returns the history rearranged according to the linearization. More formally, if an action **a** or an ancestor of **a** is prior to another

action **b** or an ancestor of **b** in the linearization **L**, then **h|a** precedes **h|b** in **Serial(h, L)**. The order between events of **a** and events of **a**'s sub-actions is preserved in **Serial(h, L)**.

We define **Globally_Atomic_Objects** as the set of globally atomic objects in the system. A history **h** is *globally atomic* iff:

$\exists L \ \forall r_i \in$ **Globally_Atomic_Objects:** $N_i(I_i,$ **Serial(committed(h|**r_i**, L))) ≠** $\perp$
where L is a linearization for actions in h,
N_i **is the state transition function of the serial specification of** r_i**,**
I_i **is the initial state of the state machine.**

A system is atomic if it generates only atomic histories.

To simplify our proofs, we will ignore nested actions. Hence, instead of a linearization, only a total ordering of the computations in a history is needed. We will also limit a history to be a sequence of transitions and commit and abort events. In other words, an invoke event must be followed immediately by the corresponding return event. Transitions from different computations can still be interleaved. The limitation is imposed to simplify the mapping between histories in an atomic system and a consistent system. The simplification does not make any difference to our results as the positions of the invoke events in a history are irrelevant.

Without failure atomicity and nested actions, the set of atomic histories can be re-defined as follows: a history **h** is globally atomic iff

$\exists L \ \forall r_i \in$ **Globally_Atomic_Objects:** $N_i(I_i,$ **Serial(h|**r_i**, L)) ≠** $\perp$
where L is a total ordering for computations in h

Notice that since we assume that every transition is committed, no commit or abort events need to appear in **h**, which becomes a sequence of transitions.

6.2.2 Consistency

To distinguish the objects in a consistent system and an atomic system, we use the symbol r_{CI} to refer to an object in a consistent system, where **C** in the subscript **CI** refers to the set of conflicting transitions pairs.

$C = \{ (t1, t2) \mid t1 \text{ and } t2 \text{ are conflicting transitions of some object } r_{Ci} \}$

We assume that there is some mechanism for an application to define C.

The semantics of each object r_{Ci} is defined with a state machine similar to those used to define serial specifications of atomic objects. The state machine used to define the semantics of r_{Ci} has four components: N_{Ci}, S_{Ci}, I_{Ci}, and T_{Ci}, corresponding to N_i, S_i, I_i and T_i in a serial specification.

A history h_C is consistent iff:

G_{Ch_C} is acyclic and $\forall r_{Ci}: N_{Ci}(I_{Ci}, h_C|r_{Ci}) \neq \perp$

where $G_{Ch_C} = \{ (Comp_{Ca}, Comp_{Cb}) \in Computations(h_C) \times Computations(h_C)$

such that $h_C = ...t_{Ca}...t_{Cb}..., (t_{Ca}, t_{Cb}) \in C, t_{Ca} \in Comp_{Ca}$,

$t_{Cb} \in Comp_{Cb}, Comp_{Ca} \neq Comp_{Cb}\}$

$Computations(h_C) =$ set of computations that appear in h_C

In the definition above, G_{Ch_C} is a graph of edges between the computations that appear in h_C. An edge exists between two distinct computations $Comp_{Ca}$ and $Comp_{Cb}$ iff they have executed a pair of conflicting transitions t_{Ca} and t_{Cb}. To make sure that conflicting transitions executed by different computations are not interleaved, G_{Ch_C} must be acyclic. Furthermore, the transitions must be valid according to specifications of the objects in the system. Notice that there is no global total ordering governing the order in which computations appear in $h_C|r_{Ci}$.

6.2.3 Proof

Suppose a consistent system is defined with a set of conflicting transitions C, the objects r_{Ci}, and the specifications of these objects, which are in turn defined by N_{Ci}, S_{Ci}, I_{Ci}, and T_{Ci}. Our goal is to construct an equivalent atomic system defined with a set of equivalent objects r_i and the serial specifications of these objects, which are defined by N_i, S_i, I_i, and T_i. A 1-1 mapping M will be defined to map histories in the atomic system to those in the consistent system. The set of atomic histories in the

atomic system should map to the set of consistent histories in the consistent system.

We will first show how the serial specifications in the atomic system are defined. Then we prove lemma 1 which states that if a history h_C is consistent then the history $M^{-1}(h_C)$ in the atomic system is atomic, and lemma 2 which states the reverse: if a history h is atomic then the history M(h) in the consistent system is consistent.

Construction of Serial Specifications in Equivalent Atomic System

In our informal version of the proof, we argued that for each transition t_{Ca} that executes at the object r_{Ci} in the consistent system, it is necessary to encode the entire history of transitions that execute at r_{Ci} before t_{Ca} in t_a. The set of equivalent transitions T_i at the equivalent object r_i can be defined as:

$T_i = T_{Ci} \times T_{Ci}^*$

where T_{Ci} is the set of possible transitions in the object r_{Ci},

T_{Ci}^* is the set of all possible sequences of transitions in T_{Ci}

The first component of a transition t_a in T_i corresponds to the equivalent transition t_{Ca} in T_{Ci}. The second component encodes the sequence of transitions that were executed at r_{Ci} previous to t_{Ca}. To make sure that the second component does encode such a sequence and the histories in the atomic system has a 1-1 mapping with those in the consistent system, we constrain the set of histories H in the atomic system to be *coherent*:

1. if $h = ...t_a... \in H$

and $t_a = (t_{Ca}, ts_{Ca})$, $t_{Ca} \in T_{Ci}$, $ts_{Ca} \in T_{Ci}^*$

and $\forall t_d = (t_{Cd}, ts_{Cd})$ such that $h = ...t_d...t_a...: t_{Cd} \notin T_{Ci}$

then $ts_{Ca} = \Diamond$

(i.e., if t_a is the first transition that belongs to r_i in h, then the second component of t_a should be an empty sequence.)

2. if $h = ...t_a...t_b... \in H$

and $t_a = (t_{Ca}, ts_{Ca})$, $t_b = (t_{Cb}, ts_{Cb})$, $t_{Ca}, t_{Cb} \in T_{Ci}$, $ts_{Ca}, ts_{Cb} \in T_{Ci}^*$

and $\forall t_d = (t_{Cd}, ts_{Cd})$ such that $h = ...t_a...t_d...t_b...: t_{Cd} \notin T_{Ci}$

then $ts_{Cb} = ts_{Ca} \| t_{Ca}$

(i.e., if t_a and t_b are consecutive transitions that belong to the same object, then the second component of t_b should be the concatenation of the second and first components of t_a.)

The coherence requirement is an additional requirement that we need to impose on the atomic histories because it can not be expressed with the serial specifications. Since the coherence requirement deals with histories rather than serial histories, it exposes the concurrency in a system. When serial specifications are used to reason about the behavior of a system, concurrency can be ignored. This is not true for the coherence requirement. In section 3.4.3, we have talked about a similar requirement that requires exposing the concurrency in a system. In that section, we described a *lower_bound_balance* operation on an account object. In order to guarantee that an implementation does not return trivial results, such as zero, we require that a result has to be one of the possible results given the many possibilities of serialization orders and operation outcomes. Since this guarantee is a separate requirement from the serial specification, we cannot assume any non-trivial results when we reason about the behavior of *lower_bound_balance* using only the serial specifications.

Given that histories in H are coherent, there is an obvious 1-1 mapping M and its reverse M^{-1} between H and H_C, the set of possible histories in the consistent system:

$M((t_{Ca}, ts_{Ca}) \| (t_{Cb}, ts_{Cb}) \| ...) = t_{Ca}t_{Cb}...$

$M^{-1}(t_{Ca}t_{Cb}...) = (t_{Ca}, \langle\rangle) \| (t_{Cb}, \langle\rangle) \| ...$ if $t_{Ca} \in T_{Ci}, t_{Cb} \in T_{Cj}, i \neq j$

$(t_{Ca}, \langle\rangle) \| (t_{Cb}, t_{Ca}) \| ...$ if $t_{Ca} \in T_{Ci}, t_{Cb} \in T_{Ci}$

We will reuse the symbols M and M^{-1} to stand for the obvious mappings between the computations in h and h_C, or the mappings between G_{Ch_C} and a corresponding

graph in **Computations(h) X Computations(h)**. For notational convenience, we assume:

$$\forall h \in H,\ \forall t_a \in h: t_a = (t_{Ca}, ts_{Ca})$$

Note that if t_{Ca} appears in h_C at the object r_{Ci}, then ts_{Ca} is the concatenation of all the transitions that execute before t_{Ca} at r_{Ci}. In other words, $ts_{Ca} \| t_{Ca}$ is an initial subsequence of $h_C|r_{Ci}$.

We now proceed to finish our definition of the state machine of r_i by defining S_i (the set of states), I_i (the initial state), and N_i (the state transition function).

Let $S_i = T_{Ci}^*$

$I_i = \langle\rangle$

$\mathbf{critical}(t_b, ts_{Cd}) = \{t_{Cx} \in ts_{Cd} \mid (t_{Cx}, t_{Cb}) \in C\}$

$N_i(ts_{Ca}, t_b) = ts_{Ca} \| t_{Cb}$ iff $\mathbf{critical}(t_b, ts_{Ca}) \subseteq \mathbf{critical}(t_b, ts_{Cb})$

$$\text{and } N_{Ci}(I_{Ci}, ts_{Cb} \| t_{Cb}) \neq \perp$$

In the definition of N_i above, two conditions have to be satisfied in order for $N_i(ts_{Ca}, t_b)$ to be defined. The first condition requires that $\mathbf{critical}(t_b, ts_{Ca})$ is a subset of $\mathbf{critical}(t_b, ts_{Cb})$. In other words, all the conflicting transitions that execute before t_{Cb} are serialized before t_b. The second condition requires that $N_{Ci}(I_{Ci}, ts_{Cb} \| t_{Cb})$ is defined. In other words, the transition t_b must be valid according to the local execution order at r_i, since this is required in the consistency definition.

The following two lemmas will show that a history h_C is atomic if and only if the equivalent history $M^{-1}(h_C)$ is consistent.

Lemma 1: if h_C is a consistent history then $M^{-1}(h_C)$ is an atomic history

Proof:

Suppose h_C is a consistent history, let $M^{-1}(h_C) = h$

Let **L** be a total order of all the computations in **Computations(h)**

such that it is consistent with $M^{-1}(G_{Ch_C})$

Suppose **Serial**$(h|r_I, L) = t_a t_b \ldots t_{k-1} t_k$

We will use induction on k to show that $N_I(I_I,$ **Serial**$(h|r_I, L)) \neq \perp$

Basic Step:

From the definition of **critical**, we know: **critical**$(t_a, \langle\rangle) = \emptyset$

$\Rightarrow$ **critical**$(t_a, \langle\rangle) = \emptyset \subseteq$ **critical**(t_a, ts_{Ca})

Also, since $ts_{Ca}\|t_{Ca}$ is an initial subsequence of $h_C|r_{CI}$

and $N_{CI}(I_{CI}, h_C|r_{CI}) \neq \perp$

$\Rightarrow N_{CI}(I_{CI}, ts_{Ca}\|t_{Ca}) \neq \perp$

Hence $N_I(\langle\rangle, t_a) = t_{Ca} \neq \perp$

Induction Step:

Suppose $N_i(l_i, t_a t_b \ldots t_{k-1}) \neq \perp$

From the definition of N_i, we know: $N_i(l_i, t_a t_b \ldots t_{k-1}) = t_{Ca} t_{Cb} \ldots t_{Ck-1}$

Suppose $t_x \in \text{critical}(t_k, t_{Ca} t_{Cb} \ldots t_{Ck-1})$

$\Rightarrow t_{Cx} \in t_{Ca} t_{Cb} \ldots t_{Ck-1}$ and $(t_{Cx}, t_{Ck}) \in C$

$\Rightarrow (\text{Comp}_{Cx}, \text{Comp}_{Ck}) \in G_{Ch_C}$ and $(t_{Cx}, t_{Ck}) \in C$

$\Rightarrow h_C = \ldots t_{Cx} \ldots t_{Ck} \ldots$ and $(t_{Cx}, t_{Ck}) \in C$

$\Rightarrow t_{Cx} \in ts_{Ck}$ and $(t_{Cx}, t_{Ck}) \in C$

$\Rightarrow t_{Cx} \in \text{critical}(t_k, ts_{Ck})$

Hence $\text{critical}(t_k, t_{Ca} t_{Cb} \ldots t_{Ck-1}) \subseteq \text{critical}(t_k, ts_{Ck})$

Also, since $ts_{Ck} \| t_{Ck}$ is an initial subsequence of $h_C | r_{Cl}$

and $N_{Cl}(l_{Cl}, h_C | r_{Cl}) \neq \perp$:

$\Rightarrow N_{Cl}(l_{Cl}, ts_{Ck} \| t_{Ck}) \neq \perp$

$\Rightarrow N_i(t_{Ca} t_{Cb} \ldots t_{Ck-1}, t_k) \neq \perp$

$\Rightarrow N_i(l_i, t_a t_b \ldots t_k) \neq \perp$

Hence h is an atomic history

QED

Lemma 2: if h is an atomic history then M(h) is a consistent history

Proof:

Let $h_C = M(h)$

Suppose h_C is not a consistent history

$\Rightarrow \exists r_{Ci}\ \exists t_{Ca} \in h_C|r_{Ci}: N_{Ci}(I_{Ci}, ts_{Ca} \| t_{Ca}) = \perp$

or a cycle of transitions exists:

$\exists\ (t_{Cm2}, t_{Ca1}), (t_{Ca2}, t_{Cb1}), \ldots, (t_{Ci2}, t_{Cm1}) \in C$

s.t. $h_C = \ldots t_{Cm2} \ldots t_{Ca1} \ldots$, $h_C = \ldots t_{Ca2} \ldots t_{Cb1} \ldots$, ..., $h_C = \ldots t_{Ci2} \ldots t_{Cm1} \ldots$

and $t_{Ca1}, t_{Ca2} \in Comp_{Ca}$; $t_{Cb1}, t_{Cb2} \in Comp_{Cb}$; ...; $t_{Cm1}, t_{Cm2} \in Comp_{Cm}$

Suppose $\exists r_{Ci}\ \exists t_{Ca} \in h_C|r_{Ci}: N_{Ci}(I_{Ci}, ts_{Ca} \| t_{Ca}) = \perp$

$\Rightarrow \exists r_i\ \exists t_a \in h|r_i: N_{Ci}(I_{Ci}, ts_{Ca} \| t_{Ca}) = \perp$

$\Rightarrow \exists r_i\ \exists t_a \in h|r_i: N_i(s, t_a) = \perp$ for all possible $s \in S_i$

$\Rightarrow$ h is not an atomic history, contradiction

Suppose the cycle of transitions exists.

Since h is an atomic history

$\Rightarrow \exists$ a total order L of **Computations(h)** s.t. $\forall r_i\ N_i(I_i, Serial(h|r_i, L)) \neq \perp$

$\Rightarrow \exists (Comp_f, Comp_e) \in L$ s.t. $(t_{Ce2}, t_{Cf1}) \in C$,

$t_{Ce2} \in Comp_{Ce}$, $t_{Cf1} \in Comp_{Cf}$, and $h_C = \ldots t_{Ce2} \ldots t_{Cf1} \ldots$

$\Rightarrow t_{f1} \in$ **Prefix**, where $Serial(h|r_i, L) =$ **Prefix** $\| t_{e2} \|$ **Suffix**

$\Rightarrow t_{Cf1} \in critical(t_{e2}, N_i(I_i, Prefix))$

Since $t_{Cf1} \notin ts_{Ce2}$

$\Rightarrow t_{Cf1} \notin critical(t_{e2}, ts_{Ce2})$

$\Rightarrow N_i(I_i, Prefix \| t_{e2}) = \perp$

$\Rightarrow$ h is not an atomic history, contradiction

Hence h_C is a consistent history

QED

From Lemmas 1 and 2, we know that given any set of objects r_{Ci}, their specifications which are defined with N_{Ci}, S_{Ci}, I_{Ci}, and T_{Ci}, and a set of conflicting pairs of transitions C, we can construct an equivalent set of objects r_i, their serial specifications which are defined with N_i, S_i, I_i, and T_i, so that:

h_C is a consistent history iff $M^{-1}(h_C)$ is an atomic history

6.3 Objects with Simple Serial Specifications

With lemma 1 and lemma 2, we have shown that atomicity is at least as powerful as the consistency definition. However, the serial specifications that we have constructed above are impractical in that they require encoding the entire previous history in a transition. The more complicated a serial specification becomes, the more difficult it is to understand. Thus, although atomicity is as powerful, it is not always easier to understand. In this section, we will argue that the serial specifications can be simplified in many cases and still have the same behavior and concurrency. In particular, we will show that for a particular class of objects in a consistent system, their specifications can be used as the serial specifications for their equivalent atomic objects. No complicated artificial serial specifications have to be constructed. Since the specifications in the two systems are just as easy to understand and the concept of atomicity is easier to understand than the concept of the consistency definition, we will claim that our approach is preferable.

We will first define this class of objects, which we call *accurate objects*. Then we prove a lemma which shows that the set of consistent histories is a subset of the equivalent atomic histories when accurate objects reuse the specifications of their counterparts as serial specifications. Finally we argue that the class of accurate objects is a large class.

6.3.1 Accurate Objects

Ignoring the requirement that a consistent order must exist, the only difference between a consistent system and an atomic system is that the former can execute its transitions in a local execution order, whereas the latter has to make its transitions appear to be executed in a global serialization order. In general, this results in less concurrency for the atomic system. Informally, because a pair of transitions may not "commute", an implementation of the atomic system may create conflicts in the process of making sure that the pair appears to execute in the serialization order. A pair of transitions t_{Ca} and t_{Cb} commutes if:

$\forall\, h_C, h_C' \in T_{Ci}^*$

$$N_{Ci}(I_{Ci}, h_C \| t_{Ca} \| t_{Cb} \| h_C') = \perp \text{ iff } N_{Ci}(I_{Ci}, h_C \| t_{Cb} \| t_{Ca} \| h_C') = \perp$$

Consider an object r_{Ci} in a consistent system with the property that *all non-commutative pairs of transitions are conflicting*. Suppose we construct an equivalent object r_i in an atomic system using the specification of r_{Ci} as its serial specification. Suppose a transition t2 is executed after a transition t1. There are two possible scenarios: either t1 and t2 commute or they do not. In the first scenario, since t1 and t2 commute, no conflicts will be created in either system. Regardless of the serialization order or the consistent order, the transitions t1 and t2 will be valid. In the second scenario, t1 and t2 do not commute. In a consistent system, because t1 and t2 are also conflicting, t2 can only proceed if the implementation is sure that t2 is ordered after t1 in the consistent order. Reusing the consistent order as the serialization order, we can achieve the same concurrency in the atomic system: t2 can only proceed if the implementation is sure that t2 is ordered after t1 in the serialization order.

This property of r_{Ci} can be defined more formally as follows:

$\forall t_{Ca}, t_{Cb} \in T_{Ci}$: if $N_{Ci}(I_{Ci}, h_C \| t_{Ca} \| t_{Cb} \| h_C') \neq \perp$

and $N_{Ci}(I_{Ci}, h_C \| t_{Cb} \| t_{Ca} \| h_C') = \perp$ for some $h_C, h_C' \in T_{Ci}^*$

then $(t_{Ca}, t_{Cb}) \in C$

r_{Ci} has the property that whenever a pair of transitions does not commute, then it is conflicting and belongs in C. We call r_{Ci} an *accurate* object.

Notice that commutativity depends on the definition of N_{Ci}. For example, suppose the specification of a bank account object is defined with the state machine in figure 6-1. This specification is similar to the one we defined in figure 3-1 except that *insufficient_funds* may be returned even when the balance is more than enough to cover the withdrawal. The motivation of this non-determinism is to allow a pessimistic reply to be returned immediately instead of being delayed by tentative updates.

In the state machine in figure 6-1, the only pairs of transitions that do not commute are **(read_balance_x, deposit_y_okay)**, **(deposit_y_okay, read_balance_x)**, **(read_balance_x, withdraw_y_okay)**, **(withdraw_y_okay, read_balance_x)**, and **(deposit_x_okay, withdraw_y_okay)**. The transition pair **(withdraw_y_okay, deposit_x_okay)** commutes since the extra deposit does not invalidate the withdrawal. Also, the transition **withdraw_x_insuf** commutes with all other transitions, even though "normally" we would expect it not to commute with **deposit_y_okay** and **withdraw_y_okay** transitions.

S_{Ci}: real numbers
T_{Ci}: $\langle$deposit(x), r_{Ci}, a$\rangle\langle$okay, r_{Ci}, a$\rangle$ = deposit_x_okay
$\langle$withdraw(x), r_{Ci}, a$\rangle\langle$okay, r_{Ci}, a$\rangle$ = withdraw_x_okay
$\langle$withdraw(x), r_{Ci}, a$\rangle\langle$insufficient_funds, r_{Ci}, a$\rangle$ = withdraw_x_insuf
$\langle$read_balance(), r_{Ci}, a$\rangle\langle$x, r_{Ci}, a$\rangle$ = read_x
where a is a computation, x is a positive real number.
I_{Ci}: 0

N_{Ci}(s, deposit_x_okay) = s + x
N_{Ci}(s, withdraw_x_okay) = s - x if s $\geq$ x
N_{Ci}(s, withdraw_x_insuf) = s
N_{Ci}(s, read_x) = s if s = x

Figure 6-1:Specification of a Bank Account Object in a Consistent System

6.3.2 Specifications of Accurate Objects Can Be Reused

We will show that if r_{Ci} is accurate and the serial specification of the equivalent object r_i is defined as:

$$I_i = I_{Ci}, S_i = S_{Ci}, T_i = T_{Ci}, N_i = N_{Ci}$$

then the set of atomic histories includes the set of consistent histories. An equivalence in behavior and concurrency is achieved without defining artificial serial specifications for r_{Ci}. Rather, the same specification used in the consistent system is used.

The current consistency definition precludes the two sets of histories from being equal. However, the stronger requirement of equality is not necessary as histories that are atomic but not consistent are indistinguishable from the other atomic ones in the sense that all the atomic histories can be generated by some serial execution. Equality can be proved if we use the following more general consistency definition:

h_C is a consistent history iff

G_{CTh_C} is acyclic and $\forall r_i \; \exists L_i$: $N_{Ci}(I_{Ci}, Serial(h_C|r_{Ci}, L_i)) \neq \perp$

where L_i is a total order of the transitions in $h_C|r_{Ci}$

$$G_{CTh_C} = \{ (Comp_{Ca}, Comp_{Cb}) \in Computations(h_C) \times Computations(h_C)$$
$$\text{such that } (t_{Ca}, t_{Cb}) \in L_i \text{ for some } i, (t_{Ca}, t_{Cb}) \in C, t_{Ca} \in Comp_{Ca},$$
$$t_{Cb} \in Comp_{Cb}, Comp_{Ca} \neq Comp_{Cb} \}$$

Using the new definition does not change our previous results except that N_i in section 6.2.3 has to be redefined. In the following proof, we will use the old definition.

Lemma 3: if h_C is a consistent history then $M^{-1}(h_C)$ is an atomic history

(The mappings M and M^{-1} can be extended in the obvious way. For example, suppose t_{Cb} is a transition of an accurate object whereas t_{Ca} and t_{Cc} are not.

$M((t_{Ca}, ts_{Ca}) \| t_{Cb} \| (t_{Cc}, ts_{Cc}) \| \dots) = t_{Ca}t_{Cb}t_{Cc}\dots$

$M^{-1}(t_{Ca}t_{Cb}t_{Cc}\dots) = (t_{Ca}, \langle\rangle) \| t_{Cb} \| (t_{Cc}, \langle\rangle) \| \dots$ if $t_{Ca} \in T_{Ci}, t_{Cc} \in T_{Cj}, i \neq j$

$(t_{Ca}, \langle\rangle) \| t_{Cb} \| (t_{Cc}, t_{Ca}) \| \dots$ if $t_{Ca} \in T_{Ci}, t_{Cc} \in T_{Ci}$

If all the objects in the system are accurate, then **M** and M^{-1} become the identity mapping.)

Proof:

Let **Commutative**$_i \subseteq T_{Ci}^* \times T_{Ci}^*$ s.t. $(h_{Ci}, h_{Ci}') \in$ **Commutative**$_i$ iff

1. $N_{Ci}(I_{Ci}, h_{Ci}) \neq \perp$, and
2. $N_{Ci}(I_{Ci}, h_{Ci}') \neq \perp$, and
3. $h_{Ci} = h\|t_1\|t_2\|h'$, $h_{Ci}' = h\|t_2\|t_1\|h'$ where $t_1, t_2 \in T_{Ci}$, or $h_{Ci} = h_{Ci}'$

Let **Reachable**$_i$ be the transitive closure of **Commutative**$_i$

Suppose h_C is a consistent history, let $M^{-1}(h_C) = h$

Let L be a total order of all the computations in **Computations(h)**

such that it is consistent with $M^{-1}(G_{Ch_C})$

For non-accurate objects, we can show that $N_i(I_i,$ **Serial**$(h|r_i, L)) \neq \perp$ as

before.

For accurate objects r_{Ci}, let $h_C|r_{Ci} = h|r_i = t_1t_2\dots t_{m-1}t_m$

In the rest of the proof we will use induction on k to show that:

(**Serial**$(t_1\dots t_k, L)\|t_{k+1}\dots t_m,\ h_C|r_{Ci}$) $\in$ **Reachable**$_i$ $\forall k = 1,2,..,m$

In particular, since it is true for k = m:

$\Rightarrow$ (**Serial**$(t_1\dots t_m, L), h_C|r_{Ci}) \in$ **Reachable**$_i$

$\Rightarrow N_{Ci}(I_{Ci},$ **Serial**$(t_1\dots t_m, L)) \neq \perp$

$\Rightarrow N_i(I_i,$ **Serial**$(h|r_i, L)) \neq \perp$

$\Rightarrow M^{-1}(h_C)$ is an atomic history

Basic Step: k = 1

It is obvious that $(t_1...t_m, h_C|r_{Ci}) \in$ **Reachable**$_i$ as:

$$t_1...t_m = h_C|r_{Ci} \text{ and } N_{Ci}(I_{Ci}, h_C|r_{Ci}) \neq \perp$$

Induction Step:

Suppose (**Serial**$(t_1...t_k, L)|||t_{k+1}...t_m$, $h_C|r_{Ci}$) $\in$ **Reachable**$_i$

Let **Serial**$(t_1...t_k, L) = u_1...u_k$

Let **Serial**$(t_1...t_{k+1}, L) = u_1...u_j t_{k+1} u_{j+1}...u_k$

From the definition of L, we know: $(t_{k+1}, u_{j+1}), ..., (t_{k+1}, u_k) \notin C$

$\Rightarrow N_{Ci}(I_{Ci}, u_1...u_{k-1}t_{k+1}u_k t_{k+2}...t_m) \neq \perp$ since r_{Ci} is accurate

$\Rightarrow$ ($u_1...u_{k-1}t_{k+1}u_k t_{k+2}...t_m$, $u_1...u_{k-1}u_k t_{k+1}t_{k+2}...t_m$) $\in$ **Reachable**$_i$

...

$\Rightarrow$ ($u_1...u_j t_{k+1}u_{j+1}...u_k t_{k+2}...t_m$, $u_1...u_{k-1}u_k t_{k+1}t_{k+2}...t_m$) $\in$ **Reachable**$_i$

$\Rightarrow$ (**Serial**$(t_1...t_{k+1}, L)|||t_{k+2}...t_m$, $h_C|r_{Ci}$) $\in$ **Reachable**$_i$

QED

6.3.3 There Are Many Accurate Objects

There are three possible kinds of pairs of non-commutative transitions:

1. mutator - observer
2. mutator - mutator
3. observer - mutator

Notice that case 3 is different from case 1 because a mutator transition and an observer transition can be defined as conflicting if they execute in one order but non-conflicting in the other order. We will argue that in most cases, an application

would define the three kinds of non-commutative transitions as conflicting. Hence most objects are accurate.

Mutator - Observer

The main reason for a mutator-observer pair to be conflicting is that there is no concurrency gained by making them non-conflicting. Typically, when a mutator-observer pair does not commute, the validity of the result returned by the observer also depends on the outcome of the mutator. Consequently, because the observer has to be delayed in any case, making them conflicting does not cause any loss in concurrency.

The bank account object with its N_{CI} defined in figure 6-1 can be used to illustrate this argument. Suppose the account object has the following pairs of transitions in C:

(read_balance_x, deposit_y_okay), (read_balance_x, withdraw_y_okay), (deposit_y_okay, read_balance_x), (withdraw_y_okay, read_balance_x)

These conflicting transition pairs in C prevent audit computations from interleaving with fund transfer computations. However, because **(deposit_x_okay, withdraw_y_okay)** is not in C, the account object is not accurate. We will show that no concurrency is gained by making the account object not accurate.

Consider an implementation of a consistent system in which an algorithm similar to a dynamic concurrency control algorithm is used to guarantee that a consistent order exists. An incoming transition t is delayed until any previously executed transition t' is finalized if $(t', t) \in C$. Also, to guarantee that $N_{CI}(I_{CI}, h_C) \neq \perp$, a **withdraw_x_okay** transition is generated only when previous committed deposits in h are sufficient to cover the unaborted withdrawals. A **withdraw_x_insuf** transition can be generated anytime without creating any conflicts.

The same implementation can be used if we define the account object as atomic with N_{CI} as its serial specification and use a dynamic concurrency control algorithm. This

is true despite the fact that successful withdraw transitions and deposit transitions are not commutative. Two factors contribute to this equivalence. First, the implementation has the property that the conflicting transition pairs in a history generated by the implementation are ordered by their commit timestamps. Second, if a withdraw transition depends on some previous deposit transitions, it must be committed only after they are committed. Consequently, if we compare the actual execution order and the serialization order, a successful withdraw transition is ordered after a deposit transition in both orders if the withdrawal depends on the deposit.

To present our arguments more rigorously, consider a sequence of transitions $s = u_1 \ldots u_r d_x w_y v_1 \ldots v_s$ such that

$N_{Cl}(I_{Cl}, u_1 \ldots u_r d_x w_y v_1 \ldots v_s) \neq \perp$

where d_x is a deposit_x_okay transition,

w_y is a withdraw_y_okay transition.

Consider the sequence with the two transitions d_x and w_y reversed: $s' = u_1 \ldots u_r w_y d_x v_1 \ldots v_s$.

Since $N_{Cl}(I_{Cl}, s) \neq \perp$

$\Rightarrow N_{Cl}(I_{Cl}, u_1 \ldots u_r) \neq \perp$

Also, *if* $N_{Cl}(I_{Cl}, u_1 \ldots u_r w_y) \neq \perp$

then $N_{Cl}(I_{Cl}, s') \neq \perp$, since the v_i's are not affected by the order of the withdraw and deposit transitions

If the system is implemented with the dynamic algorithm that we described above, we know that the order in which the computations commit, L, is consistent with G_{Ch_C}. Obviously, either w_y is committed after d_x or d_x is committed after w_y. If the former is true, we know that w_y is serialized after d_x according to L and we would not have to "switch" w_y in front of d_x during the induction step in lemma 3. In other words, we do not have to worry about the validity of s'.

If d_x is serialized after w_y, it must be uncommitted when w_y is executed. Furthermore, due to the property of the concurrency control algorithm, all the committed deposits at the time w_y is executed must be represented in $u_1 ... u_r$. Assuming that the bank object cannot predict whether uncommitted deposits will commit, it implies that:

$$N_{Cl}(I_{Cl}, u_1 ... u_r w_y) \neq \perp$$

Consequently, we know that for any consistent h_C generated by the implementation that we described above, **Serial**$(h_C|r_{Cl}, L)$, $h_C|r_{Cl}) \in$ **Reachable**$_i$ despite the fact the account object r_{Cl} is not accurate. Making **(deposit_x_okay, withdraw_y_okay)** non-conflicting does not gain any concurrency.

Mutator - Mutator

Before describing the reasons why a mutator-mutator pair should be conflicting, we should observe that there are many mutator-mutator pairs that commute. For example, all the mutators in the bank account example commute with one another because increments and decrements commute. Similarly, in an airline reservation system, increments and decrements of seat counts commute with one another. The concurrency problem that we encounter in these applications is usually due to conflicts between observers and mutators.

Nevertheless, there are also many examples in which two mutators do not commute. One of them involves an "overwrite" transition, such as resetting the value of a counter, which does not commute with other mutator transitions. In a calendar application, changing the meeting place of a meeting appointment does not commute with another transition that changes the meeting place of the same appointment. In a FIFO-queue, the order in which items are enqueued determines the order in which items are dequeued. Two enqueue transitions do not commute.

There are several reasons why these non-commutative transition pairs should be conflicting. First, making them conflicting is the only means to maintain consistency

within a set of objects. For example, in a replicated object, if a computation that performs an "overwrite" operation at each replica can interleave with other mutator computations, the state that results at each replica is no longer consistent. This is probably not acceptable to the application. Similarly, if two computations that change the meeting place of a meeting appointment are executed concurrently, the desirable behavior is to serialize the mutators at each participant calendar in the same order, so that at least all the participants would go to the same place for the meeting. Making the transitions that change the meeting place conflicting is the only way to guarantee such a behavior. The question of why there are two such computations initiated concurrently in the first place should be left for arbitration at a higher level.

Second, making two mutators non-conflicting does not improve concurrency in many cases. In the implementations that we have described in previous chapters, the validity of the results of two mutator transitions does not depend on the outcome of other transitions or the serialization order. For example, both inserting an item x and removing x from a set object return *okay* in any case. It is only when there are other observer transitions whose validity depends on the serialization order or outcomes of these mutator transitions that conflicts may be created. For example, in the implementation of a set object in figure 4-3 on page 100, the only condition under which a conflict is created by a *delete(x)* operation is when the *delete(x)* operation may be serialized between an *insert(x)* operation and a *member(x)* operation that had returned *true*. If the implementation uses a dynamic concurrency control algorithm, the only situation that such a condition can be met is when the *insert(x)* operation is committed and the *member(x)* operation tentative. In an implementation of a consistent system, whether a conflict would also be created under such a condition depends on whether *member(x)* and *delete(x)* are conflicting, which we will discuss below.

Observer - Mutator

In an atomic system, a conflict condition depends on the functionality of the application. In particular, whether a conflict is created by a mutator that executes after an observer depends on the functionality of the mutator and observer. For example, in the bank account example described in figure 6-1, no conflicts are created by any mutator that executes after the transition **withdraw_x_insuf** because *insufficient_funds* does not guarantee that the balance is less than the amount to be withdrawn.

Similarly, the relaxed semantics of *insufficient_funds* can be used to increase concurrency in a consistent system. A pessimistic answer can be returned by *withdraw* if there are tentative mutators. Given that *insufficient_funds* has a relaxed functionality, defining **withdraw_x_insuf** and **deposit_y_okay** as conflicting does not increase concurrency over an atomic system. In other words, defining an observer-mutator pair to be conflicting may not increase concurrency because the functionality of the observer may have been relaxed to avoid conflict between a mutator-observer pair of transitions.

In summary, since defining each of the three possible type of non-commutative transition pairs as non-conflicting is unlikely to increase concurrency, defining them as conflicting does not decrease concurrency either. Consequently, the set of accurate objects is likely to be a large set.

6.4 Conclusion

In this chapter we have shown that atomicity is at least as powerful as a consistency definition that is similar to some other correctness definitions proposed in the literature. By allowing serial specifications to be defined by an application, a programmer can construct an atomic system equivalent to a consistent system in terms of its concurrency and behavior. However, the serial specifications of the equivalent atomic system are too complicated to sustain our claim that our atomicity definition is easier to understand than the consistency definition. We showed that for

a class of *accurate* objects the specification used in a consistency system can be used as the serial specification in the equivalent atomic system. Since the specifications in the two systems are as easy to understand and the concept of serializability is easier to understand than the concept of consistency, we claim that atomicity is at least as powerful *and* easier to understand in the case of accurate objects. We argued that the class of accurate objects is a large class because it is unlikely to have non-conflicting non-commutative transition pairs.

This chapter finishes our discussion of concurrency. In the next chapter we will turn our attention to resilience problems in a system with long computations.

Chapter Seven

Resilience

When the execution of a computation spans a long period of time, the probability of its encountering some transient failure increases. After a failure, a computation may have lost its program state (e.g. local variables) before the failure and be unable to resume its execution. Unless precautions are taken to guard against these transient failures, a computation becomes more and more unlikely to be completed successfully when its length increases. Other than site crashes, transients failures also include deadlocks and invalid assumptions in concurrency control algorithms.

Two kinds of resilience problems are dealt with in this chapter. The first kind of resilience problems is concerned with limiting the amount of lost work when a failure occurs. The use of nested actions is a partial solution: aborting a sub-action in progress does not undo the sibling actions or the parent action. However, using sub-actions alone is not sufficient. If a sub-action is aborted after it had finished and the abort is not initiated by the parent action, the parent action has to be aborted also. Since the execution of the sub-action may be non-deterministic and have affected the subsequent execution of the parent action, a mere re-execution of the sub-action is inadequate. Storing the modifications of the sub-action in stable memory only helps occasionally, as aborts may be caused by deadlocks and invalid assumptions in concurrency control algorithms, as well as by site crashes.

Conversely, when an action is aborted, all its sub-actions have to be aborted also. Significant delay can be added to the response time when these sub-actions are executed at remote sites. Re-executing the aborted action but not the sub-actions is not acceptable in general. The execution of the aborted action can be non-deterministic such that a different set of sub-actions may be created in the re-

execution.

The second kind of resilience problems is related to communication. In a communication network where partitions are frequent, a message may never reach the destination site if resending from the origin site is the only measure to mask partitions. Consider the communication path between two sites to consist of *switches* linked by direct communication links. If the receiver or one of these switches or links is non-operational, a partition is created. Even though individual partitions disappear over time, and the sender site can resend the message repeatedly, the system may be partitioned in such a manner that the sender and receiver sites never establish a connection along which all the components would be operational simultaneously (figure 7-1). A special case of this situation is when the sender and receiver sites are connected to the communication network at non-overlapping periods of time.

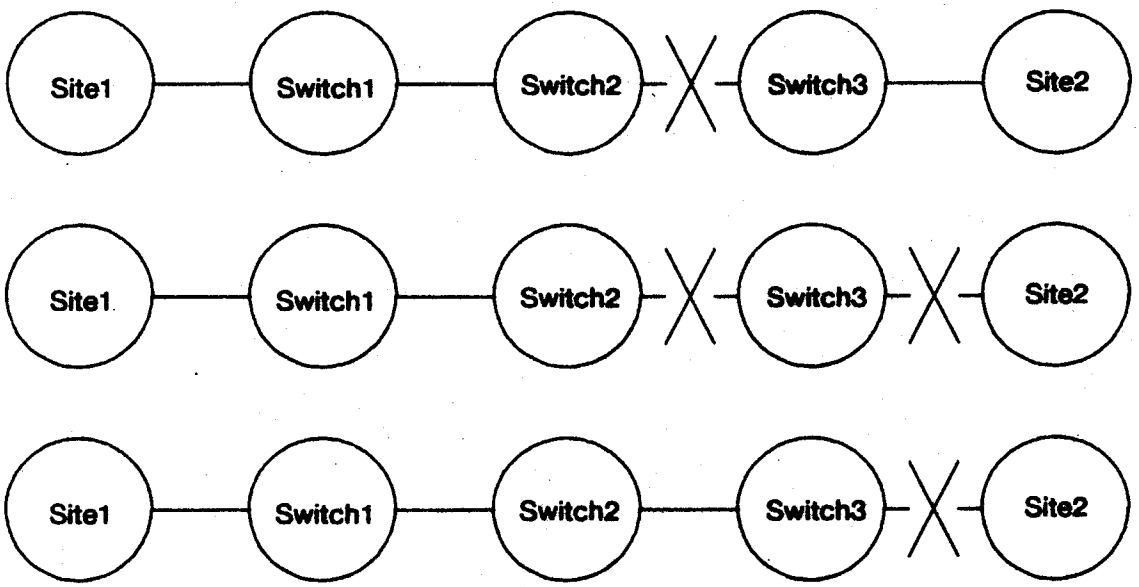

Figure 7-1:Partitions that Prevent Communication

With most current communication protocol implementations, an end-to-end connection from sender to receiver is assumed. While switches may resend to recover from a transient failure, they currently do not have the capability to buffer messages for an extended period of time, so that the ultimate resending responsibility falls back on the sender. If partitions develop, these assumptions prevent successful communication.

In section 7.1, we describe a checkpointing mechanism which allows a program interrupted by failures to restart itself at the last checkpoint. A "program" can be equated with a **procedure** in a resource manager. Checkpointing has been suggested in the literature [41, 53] to increase the resilience of a computation; our goal is to work out a checkpointing mechanism compatible with the implementation paradigm described in this thesis. In addition, because of our assumption that communication delays can be significantly long, we will discuss how to minimize aborting remote sub-actions by coordinating the checkpoints with remote invocations. Another difference between our work and other work on checkpointing mechanisms relates to the amount of information stored in a checkpoint. In order to avoid checkpointing every piece of information accessible to a program, we will describe how the program can specify a subset of its state to be preserved across checkpoints.

In section 7.2, we describe how messages can be relayed through *message transfer agents (MTA's)*. The protocol between two MTA's or an MTA and its client is simple, minimizing the state that needs to be kept on both sides. MTA's are capable of buffering messages as well as storing messages in stable memory so that messages are not lost with site crashes while waiting for partitions to disappear.

7.1 Checkpoints

This section describes how a program can checkpoint its state during execution. At a checkpoint, all the updates to the shared objects accessed or created by this

program should be stored in stable memory. These shared objects include all the objects accessible from the **permanent state** of the resource manager. In addition, any objects local to this program (e.g., local variables) must have their updates remembered in a known location in stable memory. Since it may be too expensive to copy all the accessible local state into stable memory, we will describe how the application program can specify a subset of the local state. Only objects in this subset are accessible after the checkpoint.

Due to our decision that only a subset of the state accessible to a program is preserved by a checkpoint, and because a **procedure** is a more convenient unit than a process to specify the subset, we will equate a program with a **procedure**. Obviously, checkpointing only the state of a program is not sufficient. To guard against site crashes, all the ancestor programs on the call stack at the same site must also be checkpointed. It may also be appropriate to extend the checkpointing beyond this site.

Our approach may provide less availability than a system in which the checkpointed state is replicated in another site with relatively independent failure characteristics. To determine the appropriate trade-off, availability should be evaluated against the cost and complexity of replication. Complexity can be reduced at the cost of special hardware support (e.g., dual-ported disks).

In the remainder of this section we describe our checkpoint mechanism in greater detail. We will describe the actions taken at checkpoint time and failure occurrences.

7.1.1 Checkpoint Time

Our discussion of the actions taken at checkpoint time will start with a brief description of the local state that needs to be stored by a checkpointing program. Storing the local state accessed by a program is not enough to guarantee resilience, however. We will also discuss how the objects accessed by previously invoked sub-programs can be stored in stable memory, and how checkpoints can be propagated

to ancestor programs.

7.1.1.1 Checkpointing a Program

At a checkpoint, a program can specify a collection of local variables in a *checkpoint record*. Together with the **permanent state** of the resource manager, a checkpoint record constitutes the accessible state after the checkpoint.

Since abstract atomic objects of an application are eventually implemented using globally atomic objects or locally atomic objects supported by the language system, storing the accessible state requires storing these system-level objects into stable memory. For concreteness, we will assume that the system-level objects are implemented using read/write locks and storing the objects into stable memory requires writing log information that contains new values of modified objects into stable memory [44]. Other algorithms are possible [48, 17].

When the log records that contain the values of modified objects are written out, they are associated with the corresponding checkpoint so that a consistent set of values can be restored after a failure. The order in which log records are stored can be used to determine the order of different checkpoints taken by a computation. The creation and preparation of a sub-action can be regarded as special checkpoints and ordered with other regular checkpoints in the log. When a restart is needed later, the ordering in the log can be used to determine the latest checkpoint to rollback to. To model checkpoints taken by parallel actions, an acyclic directed graph instead of a total order can be used to model the order.

When a checkpoint is taken, an object checkpointed may be locked or a previously acquired lock may have been released. If the object is still locked, this can be indicated in the log record so that the lock can be retained when the object is restored. If the lock is released, it is because the object is a locally atomic object and the local computation that acquired the original lock had committed. If any changes made by the locally atomic computation had been written out to stable memory, no

further work needs to be done. Otherwise, any changes made by the locally atomic computation, including the decision to commit the locally atomic computation, can be flushed to stable memory.

One complication remains. If a locally atomic object is checkpointed while a lock is held and the lock is subsequently released, it may not be possible to rollback to that checkpoint because some other locally atomic computation could have accessed the object and possibly committed. One of the solutions is to disallow checkpointing a locally atomic object when it is locked. This is not a severe restriction because we expect checkpoints to be taken between, and not during, short locally atomic computations. Linguistically, a checkpoint can be taken as the end of a locally atomic computation, which forces locks to be released at the checkpoint. Another possibility is to discard the checkpoint as if it had never been done when locks are released later. The decision to discard a checkpoint can be written to stable memory together with the decision to commit the locally atomic computation and release locks.

Log records about a checkpoint can be discarded when the action in which the checkpoint is executed is finalized[25].

Linguistically, in order to enforce the scope of the local variables so that the program after the checkpoint can only access those objects contained in the checkpoint record or **permanent state**, we require the program to continue in a separate program module after a checkpoint. We call this program module a *continuation procedure*, the name of which is stored in stable memory and associated with the checkpoint. The permanent state is accessible to all program modules in the **resource manager**. The checkpoint record can be made accessible to the continuation procedure as its "arguments." See figure 7-2 for an example.

---

[25] If the only source of failures is site crashes, a checkpoint can be discarded once the action executes a later checkpoint or is prepared.

```
calendar = resource manager is ...
        permanent state is
                a: table[slot]
        ...
        ...
        make_appointment = procedure(...)
                local1: integer
                ...
                ...
                checkpoint(local1, ...)
                continue at cont1
                end make_appointment

        cont1 = procedure(clocal1: integer, ...)
                ...
                ...clocal1...
                ...
                ...a...
                ...
                end cont1
```

Figure 7-2:A Program Using Checkpoints

7.1.1.2 Propagating a Checkpoint to Previously Invoked Sub-Programs

In addition to the local objects accessed by this program, other objects accessed by the sub-programs previously invoked by this program should also be stored in stable memory. Since these sub-programs had already returned, no local variables need to be stored. Only the objects in the **permanent state** of the resource managers in which these sub-programs executed have to be written out to stable memory. If a sub-program and its parent execute at the same site, a single stable memory access can be used to write out all the log records. If they execute on different sites, the parent has to send messages to inform the sub-program of the checkpoint.

To simplify our discussion, we assume that all remote sub-programs are executed in sub-actions. If these remote sub-actions have already prepared, no extra work is needed. Otherwise, *prepare* messages should be sent to the remote sub-actions. If a *no* vote is returned by a sub-action, this action has to be rolled back to a checkpoint taken before the sub-action is created. We will discuss rollbacks in the next section.

It is not necessary for the parent to wait for a remote sub-action to prepare before proceeding. However, when the parent prepares later, it has to make sure that the sub-action has also prepared.

7.1.1.3 Two Kinds of Checkpoints

Two kinds of checkpoints are allowed in this proposal. The first kind of checkpoints is associated with a procedure call. Under our model, the length of a computation is attributed to communication delays. Consequently, if a program expects a long delay in the return of a remote procedure call, it should execute a checkpoint immediately after evaluating any arguments but before the call. If the site in which the caller resides crashes during the wait, any previous work, such as calling some other remote procedures, *and* the ongoing call would not have to be aborted. Executing the checkpoint before the call minimizes the possibility that the caller will be aborted. By associating the procedure call with the checkpoint, we guarantee that the checkpoint will be immediately before the call and the deterministic processing in between would not invalidate the invoke message.

The second kind of checkpoints is not associated with any procedure calls. These checkpoints are executed when a program arrives at some "logical breaks." At these logical breaks, the remaining tasks in the program are relatively independent of previous tasks. Little or no local state is required to be stored for the continuation procedure. However, if we assume that a program spends relatively little time between remote calls, there is less motivation for these checkpoints.

When a checkpoint associated with a procedure call is executed, the arguments and a unique *frame identifier* of the callee will be stored along with other information in stable memory. A frame identifier uniquely identifies a program. We assume that frame identifiers are unique over the lifetime of a system. Storing the frame identifier of a callee ensures that a program is aware of its waiting for another program to return when it is restarted. The continuation procedure will only be invoked when the procedure call finally returns. A handle can be provided to access the results of the

call in the continuation procedure. The use of frame identifiers will be discussed further in the next section.

A program can *anticipate* the delay in calling a remote procedure and execute a checkpoint at the time of the call. On the other hand, a program can also delay the checkpoint until it is informed by the system of the difficulty in communicating with the remote site. We expect the system to convey such difficulties through some system-defined exceptions. In the discussion below, we assume that an `unavailable` exception is raised at a remote call when communication with the remote site is not possible. It is possible that the invoke message might have been delivered and the remote call is actually executing.

The alternatives available to a program when an `unavailable` exception is raised depends on the exception model. With a resumption model [36], a program can execute a checkpoint and resume the outstanding call. With a termination model [29], the outstanding call is abandoned. The resumption model has the advantage that the call will not be aborted if it had been, or will be, started. The program also has the choice of abandoning the call, and pursuing some other alternatives, in which case the sub-action associated with the call will be aborted if it is ever going to be started. After the checkpoint and resumption, the state of the program is as if the checkpoint had been anticipated.

7.1.1.4 Propagating a Checkpoint to Ancestor Programs

In the discussion above, we have ignored the interaction between a program that executes a checkpoint and its ancestor programs. In fact, the resilience of the computation is not much improved if only the current program is checkpointed. In order to notify the caller of a checkpointing program, executing a checkpoint statement will also cause a special exception to be raised inside the caller. At the risk of a slight misnomer, we can reuse the name `unavailable` for the special exception. Unless the caller had anticipated the delay by a previous checkpoint, the caller has to provide a handler for the exception. To handle the exception, the caller can decide

to checkpoint its state and resume the callee. The exception can be avoided if the callee knows that the caller has a checkpoint associated with the call.

If the caller did not anticipate the checkpoint and decides to checkpoint when it receives the exception, it would in turn cause an exception to be raised in its own caller. Thus, checkpoints are propagated along the call chain (see figure 7-3). This propagating of checkpoints can be thought of as translating volatile stack frames into a chain of "stable stack frames," each of which consists of the following:

1. a checkpoint record,
2. the frame identifiers of this program and its caller,
3. a continuation procedure
4. the frame identifier of the callee and the arguments of the call if the checkpoint is associated with a procedure call.

During a checkpoint, storing updated objects and the stable stack frame into stable memory, notifying the caller, and executing the continuation procedure can all proceed in parallel. If the caller does not resume this program, the current action can be aborted asynchronously. The parallelism is needed as the caller may be from a remote site, creating a long delay in notification. If the caller and callee are at the same site, their checkpoints can be synchronized in such a manner that the storing of their states into stable memory can be buffered in a single access to stable memory. On the other hand, there may be applications that may prefer to minimize the probability of rollbacks before starting the continuation procedure. A synchronous checkpoint can be provided; the continuation procedure will only be invoked after the following has happened:

1. the caller has resumed this program,
2. the objects updated by this program and its sub-programs have been stored in stable memory,
3. the procedure call associated with the checkpoint has returned.

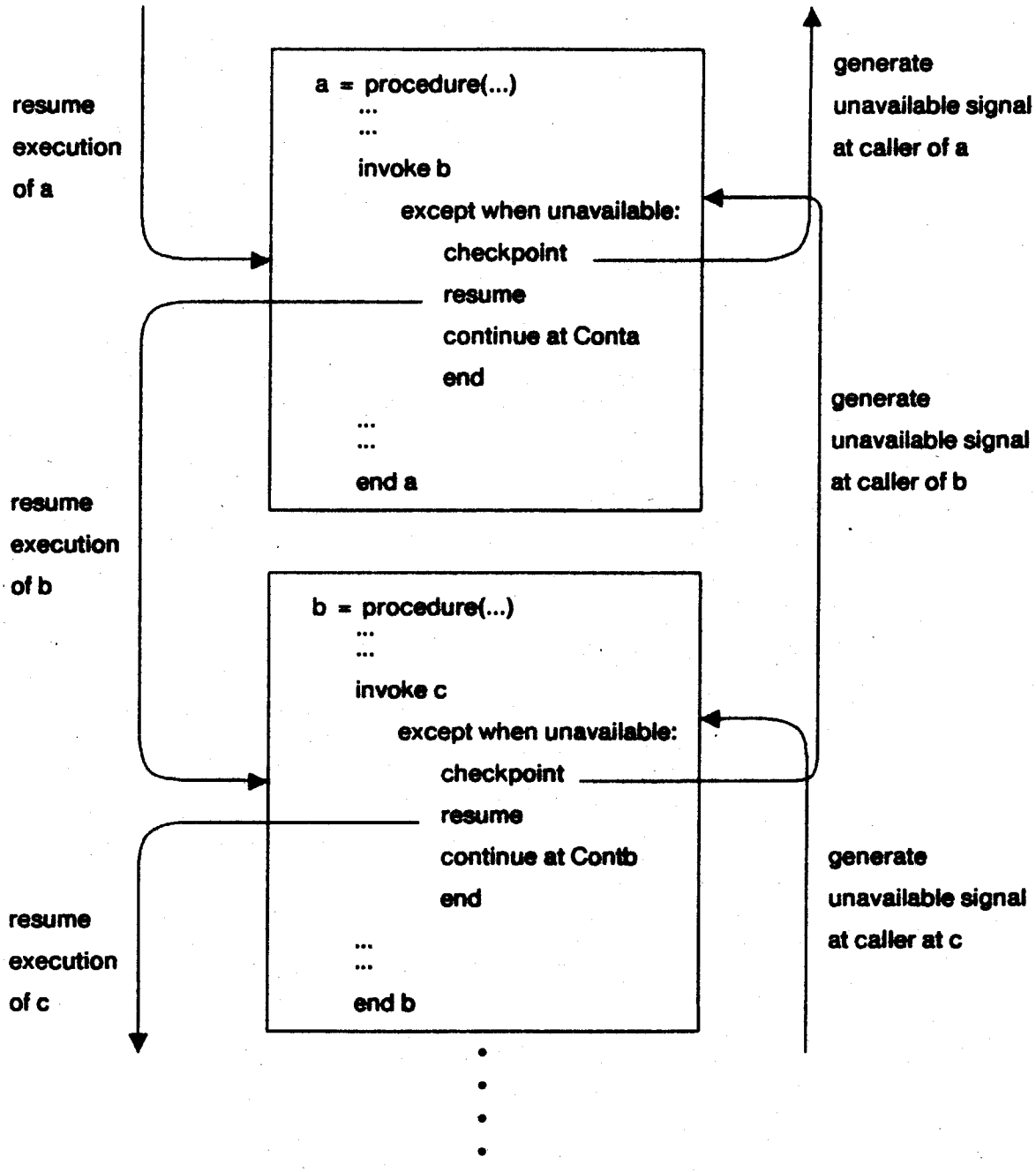

Figure 7-3:Propagating Checkpoints to Ancestor Programs

7.1.1.5 Checkpointing Parallel Sub-Actions

Consider when a checkpointing program is one of the parallel sub-actions invoked by a parent action. Like other checkpoints, the program has to supply a continuation procedure and a checkpoint record. The creator of these parallel sub-actions is also notified so that it can checkpoint if it had not anticipated the delay. In its checkpoint, it will remember the sub-actions that have not yet finished. Its continuation procedure will be invoked only after all the remaining sub-actions are finished.

Parallel sub-actions can be used to specify an application time-out. Figure 7-4 describes a scenario in which a parent action creates two parallel sub-actions: one of them sends out requests to set up a meeting, the other contains a checkpoint statement and remembers a deadline. The continuation procedure of the timer sub-action will sleep until the deadline is reached. When the timer sub-action is awakened, it will abort the sibling action or perform other necessary tasks. If the sibling action is finished before the deadline, it will abort the timer sub-action and return. We assume that there are mechanisms to abort sibling actions.

7.1.2 Restart Time

This section describes the process of restoring the state of a program to a checkpoint. First, the restartable programs have to be identified. This is not a straightforward operation as checkpoints can be asynchronous at different sites. Then the states of the sites involved have to be restored to those recorded by the checkpoints and the programs associated with the checkpoints are restarted. We will focus on the case where the failure is caused by a site crash. Later we will describe variations to handle other types of failures.

7.1.2.1 Identifying the Restartable Program

After a failure, the system should consult the record of the checkpoints. The goal is to identify the last checkpoint executed by a program whose caller is still expecting the program to return. If the failure is caused by a site crash, the system can retrieve

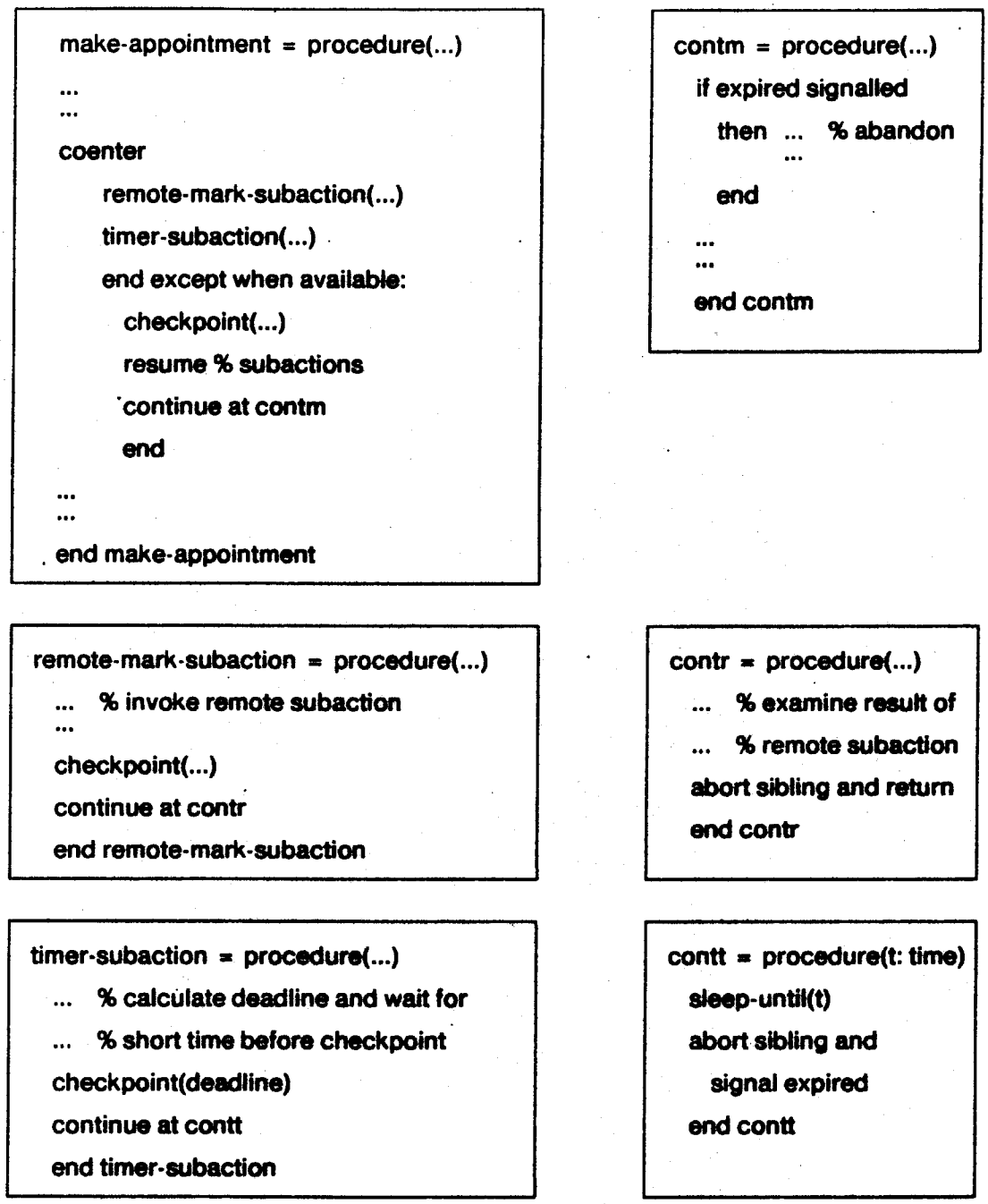

Figure 7-4:Using Parallel Sub-Actions to Specify Application Time-Out

all the checkpoint records that belong to unprepared actions from stable memory. Recall that the checkpoints created by a program are ordered in their execution order and that sub-action creation and preparation can be regarded as special checkpoints. Only programs that were executed by unprepared actions need to be restarted. Programs that had returned before an ancestor program executed a checkpoint need not be restarted either.

A program can be top-level if it executes the top-level action, in which case, the state of its caller, if the program has any, is irrelevant for recovery purposes. For the non-top-level programs that potentially need to be restarted, the frame identifiers recorded during a checkpoint can be used to identify their callers. A caller can be in a remote site and not necessarily checkpointed. If the caller is local, one of the checkpoints of the caller should be associated with a procedure call and expecting this program to return.

To determine whether the caller of a program has a checkpoint at the call or is still waiting for the call to return, a message has to be sent to a remote site if the caller is executing remotely. If a caller neither has a checkpoint at the call nor is it waiting for the call to return, the callee should be asked to abort. If the caller is still waiting for the call to return, no more work needs to be done and the callee can restart. If the caller is not waiting for the call to return but has a checkpoint at the call, the caller can continue up the chain and determine whether the caller itself can restart at that checkpoint. If the caller can restart at that checkpoint, the callee can restart also.

7.1.2.2 Restarting a Program

In order to restart a program as quickly as possible, two optimizations can be introduced. First, the sending of an inquiry message to a remote caller and a restart can proceed in parallel. This is crucial as there may be a long delay before an answer is returned. Second, a call message that invokes a remote callee can indicate whether the caller is checkpointed at the call. If it is, no inquiry messages are needed later. Also, positive replies of an inquiry message can be saved and later

inquiry messages directed to the same caller can be omitted.

When a program is restarted at a checkpoint, all the work performed after the checkpoint, including any changes to the local objects and any sub-action created, should be undone or aborted. The values of the local objects are restored according to the values recorded by the checkpoint. See [54, 35] for a discussion of detecting orphan sub-actions that are still running even when they are supposed to be aborted. To avoid committing supposedly aborted sub-actions, the return, *prepare*, and *commit_computation* messages should contain the tree of action identifiers that ought to be committed. An action should refuse preparation if the action tree contains sub-actions that should have been aborted.

To restart a program on a crashed site, the continuation procedure associated with the checkpoint can be invoked directly if the checkpoint is not associated with a procedure call. Otherwise, the program can re-invoke its callee.

7.1.2.3 Other Types of Failures

Dealing with other types of failure is similar. If an operation a is the victim of a deadlock, or a has made an invalid assumption in an optimistic concurrency control algorithm, the checkpoint before a can be considered as the "last" checkpoint before a "crash" (see figure 7-5). All work performed after the "last" checkpoint has to be undone. Determining this checkpoint requires remembering the ordering of the checkpoints and the points at which operations occur. If this is too expensive, the beginning of the action that a is executed in can be used as the last checkpoint.

7.2 Message Transfer Agents

In the introduction, we described a communication problem due to the improbability of having all the components along a communication path operational at the same time. This section describes how to alleviate the problem with *Message Transfer Agents (MTA's)*.

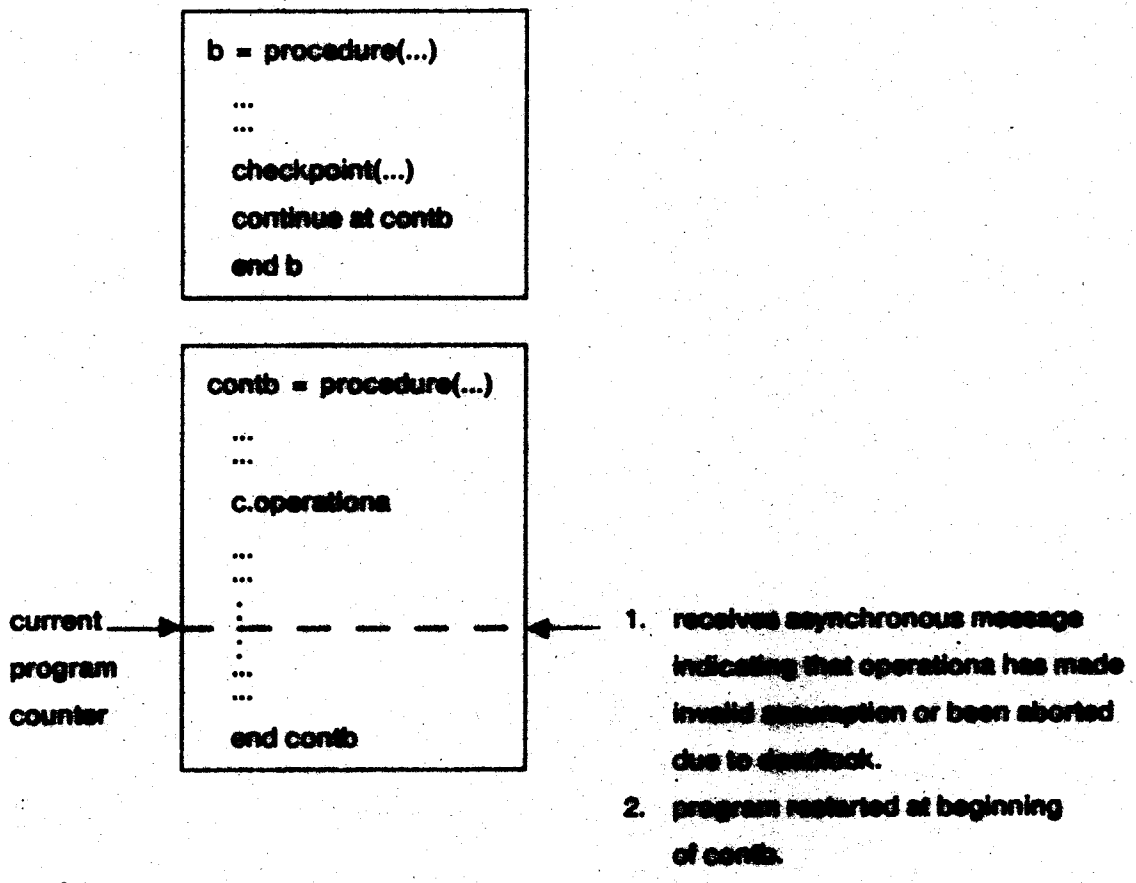

Figure 7-5:Rollbacks due to Deadlocks or Invalid Dependency Assumptions

An MTA provides buffering of messages when the sender of a message cannot communicate with the receiver directly. A message can be relayed through several MTA's before reaching its final destination. Sending the message through several relays is more likely to succeed than requiring all the components in the path to be operational simultaneously, assuming that each relay would get the message "closer" to its destination. It also avoids wasting network resources in trying to send a message over a portion of the path repetitiously. We assume that there is some routing algorithm to select relay MTA's given the MTA closest to a destination.

If a destination resource manager has a fixed network address, the system can determine which MTA is "closest" simply by some table lookup. However, a resource manager can occasionally be relocated from one address to another. For example, a resource manager can be reincarnated in a different machine when a previous one crashes, and portable computers can be carried around and reconnected to the network at different locations. If the new address has not been propagated in the system, the table lookup may not return the closest MTA.

This problem can be alleviated in two ways. First, the source and destination of a message can be expressed in resource manager identifiers, instead of network addresses. Each relaying MTA can perform a table lookup for the best MTA to send to. Another possible solution is to allow each resource manager to specify a set of MTA's as its *home MTA*'s. For example, a user may specify MTA's which are closest to his home or office as the home MTA's for his portable calendar resource manager. Messages can be replicated and sent to each of these home MTA's. Although extra resources are required for replication, these replicated messages are otherwise harmless because they are detected by the destination resource manager. A home MTA that receives a message will try to send the message to the destination resource manager periodically. A resource manager can also poll its home MTA's periodically or when it is conscious of its being reconnected to the network.

To avoid keeping messages in an MTA for an extended period of time and employing complicated algorithms to inform an MTA when messages can be deleted, an MTA assumes that it can delete a message when its delivery has been acknowledged by the destination resource manager or the next MTA on the path. If the delivery is not acknowledged (e.g., the acknowledgment message is lost), the MTA can try another path without having to worry about a possible replicated message which is harmless. In fact, a message can be replicated intentionally and relayed through different routes to increase reliability and minimize delay even when there is only one home MTA. To avoid lost messages, messages can be stored in stable memory along the route. To avoid an MTA being "stuck" with a message, each message is associated

with an expiration time and the message can be dropped when it expires. The sender of a message is responsible for resending when the message expires.

Several other protocols [47, 22, 57] provide a similar relaying service. A Simple Mail Transfer Protocol which provides a relaying service across transport service environments for mail is described in [47]. Sites that are connected to different transport services are chosen as relaying points. An asynchronous data distribution service for general distributed applications for the SNA architecture is describe in [22]. A similar service for the CCITT standard is described in [57].

The protocol we described above is not meant to be a complete specification but rather an outline of the main features. One of the features in our protocol is our assumption that a recipient can detect and discard duplicate messages. It allows us to simplify our protocol and increase reliability by replicating messages. Also, an MTA can discard messages when they expire. It allows the resources of an MTA to be reclaimed easily.

7.3 Conclusion

This chapter described the resilience problems that a computation may encounter when partitions in the network are frequent. In addition to the increased possibility of site crashes during the long execution of a computation, there is also a higher likelihood of deadlocks. To avoid a computation being aborted whenever a failure occurs, a program can execute checkpoints from which it can be restarted. We have described how the state of the program can be specified at these checkpoints. In view of the possible long communication delay between two sites, we have shown how their checkpoints can be coordinated. A program can execute a checkpoint in anticipation of or in response to a long delay in communication. A program can also inform its caller when it is performing a checkpoint.

A different resilience problem arises when it is unlikely for the sender and receiver of a message to communicate synchronously. We described a relaying service which

has a simple protocol due to its assumption that duplicate messages can be detected by the receiver.

Chapter Eight

Conclusion

This chapter summarizes our work and suggests future work.

8.1 Summary

As the size and complexity of a system grow, it becomes more difficult to understand the behavior of the system. Atomicity provides a useful tool to handle this problem. In this dissertation we have investigated how long atomic computations can be supported.

There are several questions that we tried to answer:

1. How to improve the concurrency of a system with long atomic computations?
2. Given that answers to the previous question may require application-dependent synchronization and recovery, how can the process of implementing an application be simplified?
3. Is atomicity the right model for long computations after all?
4. How can a long computation be resilient to transient failures?

Two solutions to the concurrency problem have been proposed in this thesis. The first solution involves the use of application semantics, which is not a new idea. The basis of the solution is to define atomicity using the serial specifications of abstract objects, which are specifications of the abstract objects' behavior in an environment without concurrency or failures. As long as the external behavior of an abstract object appears to be atomic, how the object masks the internal concurrency and failures is immaterial. This approach of defining atomicity naturally leads to a trade-off between functionality and concurrency. By relaxing serial specifications, concurrency is increased. Being able to trade off functionality for concurrency is an important requirement in a system with long computations. Given that an

implementation cannot predict whether tentative computations will commit and that computations can be initiated asynchronously and interleave, a concurrency problem is unavoidable unless a "weak" functionality is used.

The ability to define atomicity based on objects' serial specifications also makes atomicity at least as powerful as other correctness definitions that abandons atomicity. We have shown that given a consistent system [50], an equivalent atomic system can be defined such that the set of atomic histories is identical to the set of equivalent consistent histories. We have also argued that in many cases, the serial specifications in the equivalent atomic system are identical to the specifications used in the consistent system. Consequently, atomicity is at least as powerful *and* easier to understand. This result assures us that our atomicity definition is a useful tool.

In implementing an application, an application programmer is confronted with two problems. First, how can the serial specification of an object be defined such that there is "enough" concurrency? Second, how can abstract objects that behave atomically be implemented? We introduced a conflict model that measures the level of concurrency with how frequent conflicts are created. We have described a process with which a programmer can derive conflict conditions from the serial specification of an object. Since a conflict condition is a useful indication of the level of concurrency in an implementation, the serial specification of the object can be designed accordingly. An important characteristic of the conflict model is the masking of the underlying concurrency control algorithm. Hence, the designer of a serial specification does not have to be knowledgeable or aware of details of the underlying concurrency control algorithm.

The implementation paradigm that we suggested for the implementation of an atomic object follows the conflict model closely. When an operation is invoked, it first tests whether a conflict is created. If a conflict is created, it must be resolved. Otherwise the operation can proceed. We emphasize simplicity in our implementation paradigm. Not only do programs become easier to write, their correctness can also

be argued more easily. History objects are used to capture the necessary information that determines whether a conflict is created. We described two recovery paradigms that govern how recovery is achieved.

An important feature of a history object is that, similar to the conflict model, it masks the underlying concurrency control algorithm from the application programmer. An application programmer can write programs without having to know the underlying concurrency control algorithm and its details. The programs written can also be ported on systems with different concurrency control algorithms. This portability is important when systems with different algorithms may be merged. It is also helpful when little actual experience is available to determine the optimal concurrency control algorithm. We have shown how the programming interface can be implemented with different concurrency control algorithms.

Another implementation mechanism suggested is the concept of local atomicity versus global atomicity. By executing (short) portions of a globally atomic computation as locally atomic computations, the programming of application-dependent synchronization and recovery is simplified. A parallel with recursion can be drawn. The implementation of long atomic computations is simplified by making portions of them atomic to one another. The power of the atomicity concept is reused at a different level.

The motivation for these implementation mechanisms is to provide a stylized and well-understood way of implementing atomic objects. By using the history objects to derive conflict conditions, the recovery paradigms to perform recovery, and local atomicity to decompose synchronization and recovery, globally atomic objects can be implemented easily.

The second solution that we provide to the concurrency problem is a limited one. We have designed two novel concurrency control algorithms that minimize the occurrences of costly conflicts. These algorithms provide a limited solution because they are effective only under special conditions. For example, for the hierarchical

algorithm, costly restarts and long delays can be avoided if distributed computations and computations that both observe and mutate are rare.

Finally, we have discussed a checkpointing mechanism and a reliable message delivery service that alleviate some of the resilience problems. In view of the possible long delay to communicate between two sites, we have shown how the checkpoints within a computation can be coordinated. A program invoking another possibly remote program can execute a checkpoint in anticipation of, or in response to, a long delay in communication. It can also inform its own caller so that its caller can in turn prepare for the delay. Due to the possibly long communication delay and cost in accessing stable memory, the checkpointing process proceeds asynchronously at each site.

8.2 Future Work

In this section we will discuss a number of areas for further investigation.

8.2.1 Other Communication Primitives

In this thesis, we have chosen RPC as the communication primitive. Although it has its limitations, such as in dealing with interactions that resembles coroutines, RPC is relatively more understood and familiar to programmers. The tree of call and returns also fit nicely with the nested action tree. However, the requirement that each call must be paired with a return may pose some efficiency problem in an environment with long communication delays. It is not uncommon to have computations consisted of work that need to be done sequentially at several (more than two) sites. The arrangement that requires that shortest communication delay will have the first site invoke the second, the second invoke the third, and so on, until the last return to the first. This is not possible within the RPC paradigm.

Another type of communication primitives that has been proposed is broadcast messages [12]. Communication cost can be reduced when implementing, say,

replicated objects. In particular, the messages that need to be relayed through the MTA's described in Chapter 7 can be minimized.

Incorporating new communication primitives requires much rethinking of the design and implementation of a system. For instance, it is unclear how a nested action tree can be defined when the control structure of the computation does not follow a nested tree of invokes and returns.

There is also the problem of language design. A simple semantics of the communication primitives should be presented to the programmers. When the communication primitives are implemented on an unreliable network, the implementation should be efficient and yet conform to the semantics.

8.2.2 Hardware Configuration and Reliability

We have assumed in this thesis that each site is equipped with stable memory. This is not necessarily true for most personal workstations. One solution is to provide *stable memory servers* shared by the sites without stable memory. The protocol between the sites and the stable memory servers must not only be efficient, but also provide a reliable service seldom interrupted by site crashes. For example, if the site on which a resource manager resides crashes, one should be able to reincarnate the resource manager on a different site with the help of the stable memory server, without waiting for the original site to be recovered. By concentrating the stable memory of a system into fewer stable memory servers, better maintenance can be provided to these machines and the system becomes more reliable as a result.

A more difficult requirement is for a resource manager to be able to continue its service using another stable memory server when the original server crashes, with or without aborting ongoing computations. The problem is difficult as the resource manager may not have a copy of its entire state. A less ambitious goal is to provide some limited service, such as only allowing prepared actions to be committed. Since prepared actions have their changes written in the crashed stable memory server, the

new stable memory server can record the commitment and retrieve the changes from the crashed server later.

Instead of stable memory servers, a system can replicate the state of a resource manager on multiple sites. If these sites have relatively independent failure characteristics, the storage reliability may be as high as that provided by stable memory. Similar to the stable memory servers described above, the replicated state information can also be used to increase availability when the resource manager crashes.

A natural extension of this scheme is to replicate not only the state that needs to be stored in stable memory, but also that on volatile memory. Long computations interrupted by site crashes are not aborted and they can resume execution as soon as one of the "backup" sites where the state information is replicated is chosen as the "primary." Obviously, a resource manager cannot afford to broadcast every memory update to its backups. A checkpointing scheme not unlike the one described in Chapter 7 can be used to coordinate the updates at the backups.

8.2.3 Replication

A different form of replication can be used to reduce communication delay and increase availability of the system. The replication in the previous section can be regarded as the replication of system-level objects. Replication can also be implemented at the application level. Conceivably, an application-level object can be replicated in several sites with different representations.

Replicating at the application level has the advantage that the semantics of the application can be utilized to reduce the number of replicas that have to be accessed. Herlihy [20] discusses using the type of an operation to determine the quorums of replicas that need to accessed. Different kinds of semantic information can be used. For example, the state of an airline reservation database can be replicated in several sites. Each site can sell tickets and update their own replica.

The updates can be propagated to other sites *after* they are committed. The number of tickets sold can be kept under a ceiling as long as each site is limited to sell only a portion of the total tickets left. Periodically the number of tickets left can be recalculated.

The implementation of replicated objects in our system would present an interesting (but not mutually exclusive) alternative to the solutions we have proposed for long computations. Long communication delays can be avoided if only nearby replicas are accessed. Implementing the replicated objects with the programming paradigms and mechanisms proposed in this thesis would be an interesting test for these ideas.

8.2.4 Implementation Experience

Because the ideas proposed in this thesis have not been implemented, many of the system issues are not discussed. There is no doubt that much fine tuning of the system is needed to produce a practical implementation. For example, the scheduler of the system has to be "fair" and efficient, since there may be many pending processes waiting to be scheduled, some of them having been delayed for a long time.

Another critical component of the system is the stable memory manager. In many of our arguments, we have relied on the piggybacking of stable memory accesses to make the costs of our algorithms acceptable. Careful coding is required. If the stable memory manager is implemented with a remote stable memory server, the system performance becomes even more sensitive to the frequency of stable memory accesses.

The implementation of the communication subsystem is also left unspecified in this thesis. In particular, the timeout interval is an important parameter. Too short an interval leads to wasted effort in checkpointing. Too long an interval may jeopardize an uncheckpointed computation and delay the application from taking other appropriate actions, such as informing the user.

Finally, the usefulness of the ideas proposed in this thesis cannot be fully tested unless some applications are implemented. For example, the use of the serial specifications to specify the behavior of "large" applications would help us evaluate the practicality of our correctness definition. An implementation would also provide useful data in determining the merits of the different concurrency control algorithms, recovery paradigms, and other implementation strategies discussed in this thesis.

8.3 Conclusion

Atomicity is a powerful concept that masks concurrency and failures in a distributed system. Long computations are called for in many applications due to long delays in communication or other types of I/O events. The proposals in this thesis provide solutions to the concurrency or resilience problems that a system with long atomic computations may encounter.

References

[1]

J. E. Allchin.
An Architecture for Reliable Decentralized Systems.
PhD thesis, Georgia Institute of Technology, September, 1983.

[2]

J. E. Allchin and M. S. McKendry.
Synchronization and Recovery of Actions.
In *Proceedings of ACM Second Annual Symposium on Principles of Distributed Computing*, pages 31-44. ACM, 1983.

[3]

J. F. Barlett.
A NonStop Kernel.
In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 22-29. ACM, 1981.

[4]

R. Bayer et al.
Dynamic Timestamp Allocation for Transactions in Distributed Systems.
North-Holland, 1982, pages 9-20.

[5]

C. Beeri et al.
A Concurrency Control Theory for Nested Transactions.
In *Proceedings of ACM Second Annual Symposium on Principles of Distributed Computing*, pages 45-62. ACM, 1983.

[6]

P. A. Bernstein, D. W. Shipman, and W. S. Wong.
Formal Aspects of Serializability in Database Concurrency Control.
IEEE Transactions on Software Engineering SE-5(3):203-215, May, 1979.

[7]

P. A. Bernstein and N. Goodman.
Concurrency Control in Distributed Database Systems.
Computing Surveys 13(2):185-221, June, 1981.

[8]
P. A. Bernstein.
Two Part Proof Schema for Database Concurrency Control.
In *Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 71-84. IEEE, February, 1981.

[9]
P. A. Bernstein and N. Goodman.
Multiversion Concurrency Control - Theory and Algorithms.
ACM Transactions on Database Systems 8(4):465-483, December, 1983.

[10]
K. P. Birman.
Replication and Fault-Tolerance in the ISIS System.
In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 79-86. ACM, 1985.

[11]
A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder.
Grapevine: An Exercise in Distributed Computing.
Communications of ACM 25(4):260-274, April, 1982.

[12]
D. R. Boggs.
Internet Broadcasting.
PhD thesis, Stanford University, October, 1983.

[13]
A. Borg, J. Baumbach, and S. Glazer.
A Message System Supporting Fault Tolerance.
In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 90-99. ACM, 1983.

[14]
K. P. Eswaran et al.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of ACM 19(11):624-633, November, 1976.

[15]
D. K. Gifford and J. Donahue.
Coordinating Independent Atomic Actions.
In *Proceedings of COMPCON*. IEEE, 1985.

[16]
D. K. Gifford.
Weighted Voting for Replicated Data.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*. ACM SIGOPS, December, 1979.

[17]
J. N. Gray.
Notes on Data Base Operating Systems.
In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science Vol. 60*, pages 393-481. Springer-Verlag, 1978.

[18]
J. N. Gray et al.
The Recovery Manager of the System R Database Manager.
Computing Surveys 13(2):222-242, June, 1981.

[19]
M. Hammer and D. Shipman.
Reliability Mechanisms for SDD-1: A System for Distributed Databases.
ACM Transactions on Database Systems 5(4):431-466, December, 1980.

[20]
M. P. Herlihy.
Replicated Methods for Abstract Data Types.
PhD thesis, Massachusetts Institute of Technology, May, 1984.

[21]
C. A. R. Hoare.
Monitors: An Operating System Structuring Concept.
Communications of ACM 17(10):549-557, October, 1974.

[22]
B. C. Housel and C. J. Scopinich.
SNA Distribution Service.
IBM Systems Journal 22(4):319-343, 1983.

[23]
Z. Kedem and A. Silberschatz.
Non-Two-Phase Locking Protocols with Shared and Exclusive Locks.
In *Proceedings of Sixth International Conference on Very Large Data Bases*, pages 309-317. ACM, 1980.

[24]
H. F. Korth.
A Deadlock-Free, Variable Granularity Locking Protocol.
In *Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 105-121. IEEE, February, 1981.

[25]
H. F. Korth.
Locking Primitives in a Database System.
Journal of the ACM 30(1):55-79, January, 1983.

[26]
H. T. Kung and J. T. Robinson.
On Optimistic Methods for Concurrency Control.
Communications of ACM 6(2):213-226, June, 1981.

[27]
L. Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of ACM 21(7):558-565, July, 1978.

[28]
B. Lampson.
Atomic Transactions.
In *Distributed Systems: Architecture and Implementation, Lecture Notes in Computer Science Vol. 100*, chapter 11. Springer-Verlag, 1980.

[29]
B. H. Liskov and A. Synder.
Exception Handling in CLU.
IEEE Transactions on Software Engineering SE-5(6):546-558, November, 1979.

[30]
B. H. Liskov and R. Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(7):381-404, July, 1983.

[31]
B. H. Liskov.
The Argus Language and System.
In Goos and Hartmanis, editors, *Distributed Systems: Methods and Tools for Specification; An Advanced Course, Lecture Notes in Computer Science Vol. 190*, pages 343-430. Springer-Verlag, Berlin, 1985.

[32]
B. H. Liskov and J. Guttag.
Abstraction and Specification in Program Development.
MIT Press, 1986.

[33]
B. H. Liskov and W. Weihl.
Specifications of Distributed Programs.
Distributed Computing 1(2), April, 1986.

[34]
N. A. Lynch.
Concurrency Control for Resilient Nested Transactions.
In *Proceedings of ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 166-181. ACM, 1983.

[35]
M. S. McKendry and M. Herlihy.
Time-Driven Orphan Elimination.
In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*. IEEE, 1986.

[36]
Mesa Language Manual, Version 5.0.
1979.

[37]
C. Mohan and B. Lindsay.
Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions.
Operating Systems Review 19(2):40-52, April, 1985.

[38]
H. Garcia-Molina.
Using Semantic Knowledge for Transaction Processing in a Distributed Database.
ACM Transactions on Database Systems 8(2):186-213, June, 1983.

[39]
W. A. Montgomery.
Robust Concurrency Control for a Distributed Information System.
PhD thesis, Massachusetts Institute of Technology, December, 1978.

[40]
J. E. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
Technical Report TR-260, MIT Laboratory for Computer Science, 1981.

[41]
J. E. Moss.
Checkpoint and Restart in Distributed Transaction Systems.
In *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems*, pages 85-89. IEEE, 1983.

[42]
J. E. Moss, et al.
Abstraction in Concurrency Control and Recovery Management.
Technical Report 86-20, COINS, University of Massachusetts, Amherst, 1986.

[43]
R. Obermarck.
Global Deadlock Detection Algorithm.
ACM Transactions on Database Systems 7(2):187-208, June, 1982.

[44]
B. M. Oki.
Reliable Object Storage to Support Atomic Actions.
Technical Report TR-308, MIT Laboratory for Computer Science, 1983.

[45]
D. C. Oppen and Y. K.Dalal.
The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment.
ACM Transactions on Office Information Systems 1(3):230-253, July, 1983.

[46]
C. H. Papadimitriou.
The Serializability of Concurrent Database Updates.
Journal of the ACM 26(4):631-653, October, 1979.

[47]
J. B. Postel.
Simple Mail Transfer Protocol.
Technical Report RFC 821, ISI, University of Southern California, 1982.

[48]
D. P. Reed.
Naming and Synchronization in a Decentralized Computer System.
PhD thesis, Massachusetts Institute of Technology, 1978.

[49]
F. B. Schneider.
Byzantine Generals in Action: Implementing Fail-Stop Processors.
ACM Transactions on Computer Systems 2(2):145-154, May, 1984.

[50]
P. M. Schwarz and A. Z. Spector.
Synchronizing Shared Abstract Types.
ACM Transactions on Computer Systems 2(3):223-250, August, 1984.

[51]
L. Sha.
Modular Concurrency Control and Failure Recovery - Consistency, Correctness and Optimality.
PhD thesis, Carnegie-Mellon University, March, 1985.

[52]
D. Skeen.
Non-Blocking Commit Protocols.
In *Proceedings of ACM/SIGMOD International Conference on Management of Data*, pages 133-142. ACM/SIGMOD, 1981.

[53]
L. Svobodova.
Resilient Distributed Computing.
IEEE Transactions on Software Engineering SE-10(3):257-267, May, 1984.

[54]
E. F. Walker.
Orphan Detection in the Argus System.
Technical Report TR-326, MIT Laboratory for Computer Science, 1984.

[55]
W. E. Weihl.
Specification and Implementation of Atomic Data Types.
PhD thesis, Massachusetts Institute of Technology, 1984.

[56]
R. Williams et al.
R*: An Overview of the Architecture.
In *Proceedings of Second International Conference on Databases: Improving Usability and Responsiveness.* ACM, 1982.

[57]
CCITT VIIIth Plenary Assembly - Document 68, Study Group VII - Report R 38. 1984.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MIT/LCS/ 377	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Long Atomic Computations		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Pui Ng		8. CONTRACT OR GRANT NUMBER(s) Office of Naval Research Contract N00014-83-K-0125
9. PERFORMING ORGANIZATION NAME AND ADDRESS MIT Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA/DOD 1400 Wilson Blvd. Arlington, VA 22217		12. REPORT DATE September, 1986 13. NUMBER OF PAGES 221
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for Public Release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

distributed systems, atomicity, concurrency control, long computations, recovery, fault tolerance, reliability, programming methodology.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Distributed computing systems are becoming commonplace and offer interesting opportunities for new applications. In a practical system, the problems of synchronizing concurrent computations and recovering from failures must be dealt with effectively. Atomicity has been suggested as a tool that masks concurrency and failures from the users of a system. With synchronization and recovery mechanisms, atomic computations appear to execute indivisibly. This dissertation addresses the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

issues in implementing long atomic computations, such as computations that last for hours or even days. Long computations make synchronization more difficult because their execution is more overlapped. They are also more likely to encounter failures in their execution. Three issues are raised:

1. Should long computations be executed automatically? Or should atomicity be replaced with other correctness criteria to increase the concurrency of a system?
2. If long atomic computations can be implemented practically, are there implementation paradigms that application programmers can follow to simplify the programming effort?
3. How can long atomic computations be made resilient to transient failures?

This dissertation shows that by using the semantics of an application, a system that supports atomic computations can be made as concurrent as other systems that do not. Since atomicity is easier to understand than other correctness criteria, systems that support long atomic computations are preferable.

Using the semantics of an application requires application-dependent synchronization and recovery code, which can be complicated and introduce subtle errors easily. Several synchronization and recovery paradigms are investigated in this dissertation. The paradigms divide synchronization and recovery into levels so that the task at each level is simpler. A programming interface taht hides the concurrency control algorithm used by a system implementation is also presented.

Finally, this dissertation discusses the use of checkpoints and buffered messages to increase the resilience of long atomic computations.