MIT/LCS/TR-314

SPECIFICATION AND IMPLEMENTATION OF ATOMIC DATA TYPES

William Edward Weihl

April 1984

This blank page was inserted to preserve pagination.

Specification and Implementation

of

Atomic Data Types

by

William Edward Weihl

S.M., Massachusetts Institute of Technology (1980)

**Submitted in partial fulfillment
of the requirements for the
degree of**

Doctor of Philosophy

at the

Massachusetts Institute of Technology
March 1984

Signature of Author ___*William Edward Weihl*___
Department of Electrical Engineering and Computer Science
March 7, 1984

Certified by ___*Barbara H. Liskov*___
Barbara H. Liskov
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Departmental Committee

Specification and Implementation

of

Atomic Data Types

by

William Edward Weihl

Submitted to the Department of Electrical Engineering and Computer Science
on March 7, 1984 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

Maintaining the consistency of long-lived, on-line data is a difficult task, particularly in a distributed system. This dissertation focuses on *atomicity* as a fundamental organizational concept for such systems. It explores an approach in which atomicity is ensured by the data objects shared by concurrent activities; such objects are called *atomic objects*, and data types whose objects are atomic are called *atomic types*. By using information about the behavior of the shared objects, greater concurrency among activities can be permitted. In addition, by encapsulating the synchronization and recovery needed to support atomicity in the implementations of the shared objects, modularity can be enhanced.

This dissertation addresses three fundamental questions:

What is an atomic type?

How can an atomic type be specified?

How can an atomic type be implemented?

Atomicity of activities is a *global* property of an entire system, while atomicity of types is a *local* property of individual types. This dissertation examines three definitions of atomicity for types, each of which is *optimal*: No strictly weaker definition of (local) atomicity for types suffices to ensure (global) atomicity of activities. The definitions of atomicity discussed encompass both serializability and recoverability, and use user-supplied specifications of objects to permit greater concurrency.

The specification framework presented in this dissertation divides the specification of a data type into two parts: the serial specification, which describes how the type behaves in the absence of concurrency and failures, and the behavioral specification, which describes how the type supports atomicity. This division permits the programmer of an individual activity to ignore how atomicity is achieved, and to focus on the serial behavior of each object. In addition, the definitions of atomicity permit the behavioral specification of a type to be derived systematically from its serial specification.

A novel two-phase locking protocol, covering both synchronization and recovery, is presented and verified. The protocol uses information about the behavior of objects to achieve greater concurrency than can be achieved with protocols based on read and write operations. In addition, the protocol permits the results of an operation, as well as its arguments, to be used in determining the appropriate lock mode for the operation. Furthermore, the protocol permits operations to be both partial and non-deterministic.

Finally, several implementations of atomic types are presented, illustrating how existing techniques for synchronization and recovery can be extended to use information about the behavior of objects to increase concurrency. The dissertation also explores linguistic support for atomic types, analyzing the advantages and disadvantages of alternative approaches.

Acknowledgments

First, I would like to thank my advisor, Barbara Liskov, for her guidance and encouragement, and for her patience as I struggled to develop the ideas presented in this dissertation. In addition, I would like to thank my readers, John Guttag and Nancy Lynch, for their many perceptive comments and suggestions. All three members of my committee read drafts of my dissertation quickly and thoroughly; their suggestions were invaluable in improving its presentation and organization.

Many other people have contributed to the ideas in this thesis. I am indebted to Gene Stark for numerous discussions in the early stages of my work. I am also grateful to Maurice Herlihy and Toby Bloom for many conversations, enlightening and otherwise; to Dan Brotsky for his willingness to listen and for his invaluable comments and suggestions; to Jeannette Wing for her willingness to discuss problems, even as she was finishing her thesis; to Gary Leavens for his comments and questions on my research and for a careful reading of my entire thesis; and to the other members of the Programming Methodology group, especially Bob Scheifler, Paul Johnson, John Goree, Brian Oki, and Sheng-Yang Chiu, for their feedback on my ideas.

I cannot say enough to thank Toby, Dan, Jeannette, Chuck Harm, and Dan Jones for their friendship.

I am deeply grateful to my family for their love, support, and encouragement during the last few years. I wish that my father could have lived to see me complete this dissertation; he and my mother have been an inspiration to me throughout my life.

Finally, I thank my wife, Heather, with all my heart for her unceasing love and support as I have struggled to finish this thesis.

To my father.

Table of Contents

Table of Figures

Chapter One

Introduction

There are many applications in which the manipulation and preservation of long-lived, on-line data is of primary importance. Examples of such applications are banking systems, airline reservation systems, office automation systems, database systems, and various components of operating systems. A major issue in such systems is preserving the consistency of on-line data in the presence of concurrency and hardware failures. In this dissertation we consider how to define data objects that help provide this consistency.

To support consistency it is helpful to make the activities that use and manipulate data *atomic*. Atomic activities are often referred to as *actions* or *transactions*; they were first identified in work on databases [Davies 73, Davies 78, Eswaren *et al.* 76]. Atomic activities are characterized informally by two properties: serializability and recoverability. *Serializability* means that the concurrent execution of a group of activities is equivalent to some serial execution of the same activities. *Recoverability* means that each activity appears to be all-or-nothing: either it executes successfully to completion (in which case we say that it *commits*), or it has no effect on data shared with other activities (in which case we say that it *aborts*).

Nested transactions [Davies 73, Reed 78, Moss 81, Lynch 83] are useful for decomposing activities into smaller units. Nested transactions provide increased failure-tolerance: Subtransactions of a transaction fail independently of each other and independently of the containing transaction. In addition, nested transactions can be used to run parts of the same activity concurrently, while ensuring that their concurrent execution is serializable. As discussed in [Liskov 82], nested transactions permit a simple implementation of a remote procedure call primitive with "at-most-once" semantics: A remote call is executed either zero or one times; partial and multiple executions cannot occur.

Atomicity simplifies the problem of maintaining consistency by decreasing the number of cases that need to be considered. Since aborted activities have no effect, and every concurrent execution is equivalent to some serial execution, consistency is ensured as long as every possible serial execution of committed activities maintains consistency. Even though activities execute concurrently, concurrency can be ignored when checking for consistency.

In this dissertation we explore an approach in which atomicity is achieved through the shared data objects, which must be implemented in such a way that the activities using it appear to be atomic. Objects that provide appropriate synchronization and recovery are called *atomic objects*; atomicity is guaranteed only when all objects shared by activities are atomic objects.

By encapsulating the synchronization and recovery needed to support atomicity in the implementations of the shared objects, we can enhance modularity; in addition, by using information about the specifications of the shared objects, we can increase concurrency among activities.

Atomic objects are encapsulated within *atomic abstract data types.* An abstract data type consists of a set of objects and a set of primitive operations; the primitive operations are the only means of accessing and manipulating the objects [Liskov & Zilles 74]. In addition, the operations of an atomic type ensure serializability and recoverability of activities using the type.

In this dissertation we investigate the semantics of atomic types and the problems involved in implementing them. We address three fundamental questions:

> *What is an atomic type?*
>> We need a precise characterization of the behavior of atomic objects. For example, we need to know how much concurrency can be allowed by an atomic type.

> *How do we specify an atomic type?*
>> What aspects of the type's behavior must appear in the specification of an atomic type, and how should the specification be structured?

> *How do we implement an atomic type?*
>> What problems must be solved in implementing an atomic type, and what kinds of programming language constructs make this task simpler?

The remainder of this chapter is organized as follows: In Section 1.1, we review protocols for implementing atomicity. Next, in Section 1.2, we discuss atomic types in more detail. Then, in Section 1.3, we summarize the contributions of this dissertation. In Section 1.4, we discuss related work. Finally, in Section 1.5, we outline the rest of the dissertation.

1.1 Implementing Atomicity

There are two subproblems that must be solved to implement atomicity: recovery of aborted activities, and scheduling, or synchronization, of concurrent activities. The effects of aborted activities must be undone, and the earlier states of objects recovered, to ensure that aborted activities have no effect on the state of the system. In addition, activities must be synchronized to avoid non-serializable executions. We discuss these two subproblems in the next two subsections.

1.1.1 Recovery

Recovery is accomplished by maintaining redundant information. There are two basic techniques for performing recovery: undo logs [Gray et al. 81, Verhofstad 78], and intentions lists [Lampson 81, Verhofstad 78]. The representation of an object in both cases is divided into two pieces: the actual value of the object, and separate recovery information (either an undo log or an intentions list).

Undo logs work as follows: When an activity invokes an operation on an object, the operation is performed on the value of the object, and sufficient information is recorded in the object's undo log so that the effects of the operation can be undone if the activity that executed the operation later aborts. For example, if an activity executes a write operation on an object, the old value of the object might be saved in the object's undo log. The undo log for an activity is simply discarded if the activity commits.

Intentions lists are used slightly differently: When an activity invokes an operation on an object, the operation is simply recorded in the intentions lists associated with the object; it is not actually performed on the value of the object until the activity commits. For example, if an activity executes a write operation on an object, the new value of the object might be saved in the object's intentions list; the new value will replace the old value only if the activity commits. If the activity aborts, the list of its operations is discarded.

A detailed description of these techniques and alternative storage organizations for them may be found in [Verhofstad 78].

1.1.2 Synchronization

Many protocols have been developed for synchronizing concurrent activities to ensure serializability (or *concurrency control*, as this problem is called in the literature on database systems -- see [Bernstein & Goodman 81] for a survey of a large number of concurrency control protocols.) Most are variations or hybrids of two simple techniques: two-phase locking [Eswaren et al. 76] and multi-version timestamping [Reed 78]. We discuss these two techniques and a hybrid below.

1.1.2.1 Locking Protocols

One of the earliest protocols developed for concurrency control is two-phase locking [Eswaren et al. 76]. Two-phase locking works as follows: Before reading an object X, an activity must acquire a read lock on X. Similarly, before writing an object, an activity must acquire a write lock on the object. An activity can acquire a lock on an object only if no concurrent activity holds a conflicting lock on the object. In addition, once an activity releases one lock, it is not allowed to acquire any additional locks.

Two locks on an object *conflict* if one is a write lock. This definition of conflicting locks ensures that at most one activity is writing an object at a time, but also allows multiple activities to read an object concurrently.

The requirement that an activity not acquire any more locks after it releases a lock means that activities acquire locks in a *two-phase* manner (hence the name of the protocol). During the *growing phase* an activity acquires locks without releasing any locks. When an activity first releases a lock, it enters the *shrinking phase*. During this phase the activity releases its locks, but may not acquire any more locks. As is shown in [Eswaren *et al.* 76, Papadimitriou 79], two-phase locking ensures that activities are serializable in the order in which they first release locks.

A variant of two-phase locking, called *strict two-phase locking*, is more suited to the applications of interest to us. Under strict two-phase locking, activities hold all locks until they commit or abort. This avoids *cascading aborts* [Wood 80], a problem with non-strict two-phase locking: If write locks are released and then an activity aborts, any activities that read the values written by the aborted activity must also be aborted. In addition, strict two-phase locking permits locks to be acquired dynamically as needed. Non-strict two-phase locking may require more advance planning, particularly to determine when an activity can release a lock.

Strict two-phase locking can be extended to nested activities as follows: Nested activities form a natural tree structure, with each activity appearing as the parent of its subactivities. We define the notions of ancestor, proper ancestor, descendant, and proper descendant in the usual way. The locking rules for nested activities are defined as follows: As before, an activity must acquire a read (write) lock before reading (writing) an object. An activity can acquire a lock on an object as long as no concurrent non-ancestor holds a conflicting lock on the object. When an activity aborts, its locks are discarded. When an nested activity commits, its locks are inherited by its parent; when a top-level activity commits, its locks are discarded. Details can be found in [Moss 81, Liskov *et al.* 83]; a proof of the correctness of Moss's algorithm is given in [Lynch 83].

1.1.2.2 Timestamp-based Protocols

The serialization order of activities attained by two-phase locking is determined *dynamically* by the order in which activities lock objects. In contrast, timestamp-based protocols determine the serialization order *statically* by selecting timestamps for activities when they start, and then force the execution of activities to obey this order.

Reed's implementation [Reed 78] of a timestamp-based protocol works as follows: An activity is assigned a unique timestamp when it begins execution. When an activity wants to modify

an object, it creates a new version of the object. A version of an object has two timestamps associated with it: the *write* timestamp, which is the timestamp of the activity that created the version; and the *read* timestamp, which is the maximum of the timestamps of activities that have read the version. When an activity with timestamp t wants to read an object, it selects the version of the object with the largest write timestamp less than t, and changes the read timestamp of the version to the maximum of its current value and t.

Write operations sometimes cannot be executed: Suppose an activity a with timestamp t wants to write an object, and a version v of the object already exists with write timestamp less than t and read timestamp greater than t. In a serial execution in which the activities execute in timestamp order, a must come between the activity that wrote the version v and the activity that read it. If a is allowed to write the object, then in the serial execution the activity that read v should instead read the value written by a. Thus, the write operation must be refused.

To avoid cascading aborts, read operations sometimes must be delayed: If an activity with timestamp t wants to read an object, and the version selected was written by an activity that has not yet committed or aborted, the read operation must wait until that activity completes. Otherwise, if the activity that created the version later aborts, the reader must also be aborted.

More details, and in particular the extension of the protocol to cope with nested activities, can be found in [Reed 78].

1.1.2.3 Hybrid Protocols

Locking and timestamp-based protocols can be combined to yield hybrid protocols that achieve greater concurrency [DuBourdieu 82, Chan *et al.* 82, Bernstein & Goodman 81]. We divide activities into two classes: read-only activities, which never modify objects; and update activities.

Update activities set locks on objects as in strict two-phase locking, but two locks conflict only if one is a read lock and the other is a write lock; a write lock no longer conflicts with another write lock. As with Reed's protocol, each write operation creates a separate version, and versions have two timestamps. Rather than choosing timestamps for update activities when they begin executing, however, we wait until they attempt to commit. Then, using a Lamport clock [Lamport 78], we ensure that the timestamps chosen for updates give a serialization order consistent with the order induced by the locks. The order of conflicting write operations is sorted out using the timestamps.

Updaters that invoke read operations always read the version with the largest timestamp. Read-only activities, however, may read older versions, permitting them to run without interfering with update activities. Timestamps for read-only activities are chosen when they

start executing. When a read-only activity wants to read an object, it simply selects the version with the largest write timestamp less than the activity's timestamp. A proof of the correctness of this protocol can be found in [Bernstein & Goodman 83].

1.2 Atomic Types

Our motivation for focusing on atomic types is two-fold: First, it is important to understand the interactions among independent objects, and to understand what constraints must be satisfied by objects to ensure atomicity of activities. Second, by using information about the behavior of operations provided by types, we can achieve greater concurrency than can be achieved by protocols based on a classification of operations as reads and writes. In this section we discuss these two issues in more detail. In the first part, we illustrate the problems caused by interactions among objects. In the second part, we discuss how user-defined atomic types can be used to increase concurrency.

1.2.1 Interactions Among Objects

Each of the protocols discussed in the previous section can be shown to ensure atomicity of activities. Thus, an object using one of these protocols should be considered "atomic." However, as the example below illustrates, objects using different protocols cannot necessarily be used together in the same system.

Consider two objects X and Y, each with read and write operations, and each with initial value 0. Suppose that X is implemented using two-phase locking, and Y is implemented using multi-version timestamping. Now suppose there are two activities A and B, with timestamps 1 and 2, respectively. Consider the following execution:

> B reads X, receiving 0.
> B writes 1 into Y.
> B commits.
> A reads Y, receiving 0.
> A writes 1 into X.
> A commits.

This execution is not serializable: In a serial execution, the second activity should see the value written by the first, but both A and B read the initial values of the objects. However, the execution is atomic at each object: At X, B is serializable before A, and at Y, A is serializable before B.

The problem in this example is that X and Y use incompatible protocols to ensure atomicity. If both objects used two-phase locking, or both used multi-version timestamping, the above execution could not occur and atomicity would be guaranteed. As we will discuss in Chapter 4, some information about the protocol used by an atomic type's implementation must be

reflected in the type's specification, and only types using "compatible" protocols can be used in the same system.

1.2.2 Type-specific Concurrency Control

The protocols discussed in the previous section were developed for simple data types, such as files and relations, with read and write operations. To support user-defined types, they must be extended to cope with arbitrary operations. The primary reason for making such an extension is increased concurrency: by using detailed information about the specifications of the operations provided by types, we can allow concurrent executions that must be forbidden if operations are simply characterized as reads and writes.

Consider, for example, a *bank account* data type, with operations to create a new bank account object (with an initial balance of 0), to deposit money in an account, to withdraw money from an account, and to check the current balance of an account. Now consider the following concurrent execution of two activities, A and B:

> A deposits $3 in a bank account X.
> B deposits $2 in X.
> A commits.
> B commits.

This execution is clearly serializable: A and B can execute serially in either order and perform the same steps. It is not permitted, however, by any of the protocols discussed in the previous section: A and B both update X (reading the current balance, and writing a new balance), and so cannot access X concurrently.

The example above is a simple illustration of a general phenomenon: By describing a system in terms of abstract objects (rather than primitive objects with read and write operations), we can permit greater concurrency than would otherwise be possible. This additional concurrency may be essential for achieving adequate performance in an application. Particularly in a distributed system, activities may take a relatively long time to complete; by permitting more concurrent access to objects, we may be able to avoid creating bottlenecks in the system. In the remainder of this dissertation we will provide more examples of this phenomenon, and will show how implementations of objects can permit high levels of concurrency.

Achieving the kind of concurrency illustrated above typically requires a more complex implementation. It may be most effective to implement a system initially permitting little concurrency, for example, by using a protocol based on reads and writes. If certain shared objects can then be identified as bottlenecks, more concurrent (albeit more complex) implementations can be substituted for the types defining those objects. Of course, there are limits to how much concurrency an atomic type can permit. One of the results of this

dissertation is a precise definition of these limits.

1.3 Overview

As mentioned earlier, we address three fundamental questions in this dissertation:

- What is an atomic type?

- How can we specify an atomic type?

- How can we implement an atomic type?

To answer the first two questions we must generalize existing work on concurrency control (or serializability) in three ways:

- Our definition of atomicity is data-dependent: It is based on an explicit specification of the desired behavior for the data objects used by activities. This is crucial in achieving the concurrency required by applications.

- Our definition of atomicity is integrated: We treat both serializability and recoverability. This facilitates the description and verification of implementations of atomic objects, which necessarily must cope with both.

- We focus on modularity issues: We identify local properties of individual objects, and we identify the conditions under which different kinds of objects can be combined in a single system while preserving atomicity of activities.

We explore three local properties, each of which is optimal: No strictly weaker local property suffices to ensure atomicity. The three properties characterize respectively the behavior of the three classes of protocols discussed in Section 1.1.2: two-phase locking protocols, in which the serialization order of activities is determined by the order in which they access objects; multi-version timestamp-based protocols, in which the serialization order of activities is determined by a pre-determined total order; and hybrid protocols, which use a combination of these techniques.

We present a novel locking protocol and verify its correctness. Our protocol generalizes previously existing protocols in two ways: It permits the results of operations, as well as their arguments, to be used in determining the appropriate lock mode, and it handles partial and non-deterministic operations. In addition, we describe and verify the implementation of both synchronization and recovery; descriptions of previously existing protocols are limited to synchronization alone.

Our approach to specifying atomic types permits the programmer of an individual activity to ignore *how* atomicity is achieved. To reason about whether an individual activity preserves consistency, one needs only the serial specification of each object used by the activity, and

the knowledge that activities are atomic; one need not know how objects cooperate to ensure atomicity.

In addition, our specification framework supports an approach that permits the concurrent specification of an object to be derived systematically from a specification of its sequential behavior. This is perhaps the most significant contribution of this dissertation: We reduce the problem of specifying an object to the simpler problem of specifying how it should behave in the absence of concurrency.

Finally, we present several example implementations of atomic types, illustrating how existing techniques for synchronization and recovery can be extended to use information about the specifications of objects to increase concurrency. We also discuss linguistic support for atomic types, analyzing the advantages and disadvantages of several alternative approaches.

Throughout the dissertation we use a model that permits a restricted class of failures: Activities can abort, but objects cannot fail. This model is an abstraction of real systems, and can be approximated arbitrarily closely with a commitment protocol (e.g., two-phase commit [Gray 78, Lampson 81] or three-phase commit [Skeen 82]) and appropriate use of redundant information (e.g., stable storage [Lampson 81]).

There are a number of important issues that we do not address:

- We do not investigate particular specification languages for atomic types; rather, we focus on the underlying formal models of specifications and on identifying desirable semantic properties of atomic types.

- We ignore some issues raised by the distributed nature of applications, such as the distinction between local and remote data.

- We do not consider how to ensure progress in the face of deadlock, starvation, and failures.

1.4 Related Work

Most early work on synchronizing concurrent processes occurred in the context of operating systems. Numerous linguistic mechanisms were developed (e.g., see [Hoare 74, Atkinson & Hewitt 77, Campbell & Habermann 74]), permitting modular implementations of synchronization for individual operations on objects. While similar implementation mechanisms are useful for supporting atomicity, the concurrency rules for atomic actions are different: Synchronization must cope with interference among activities invoking multiple operations, rather than just interference among single operations. Furthermore, there was no systematic approach for coping with failures.

Specification and proof techniques were also explored (e.g., see [Owicki & Gries 76, Owicki & Lamport 82, Pnueli 77]). Much of this work assumed a fixed set of atomic actions, and did not consider how to let the programmer build higher-level atomic actions or new data types. In addition, much of this work did not consider how to specify and verify individual modules.

None of the operating systems work provided a systematic solution to the problem of choosing the specifications for the modules in a system. Our specification approach simplifies the problem of specifying an object by reducing it to the simpler problem of specifying how the object should behave in the absence of concurrency.

The trend in database systems was quite different. Nested atomic actions were first proposed by Davies [Davies 73] (he called them *spheres of control*). Single-level transactions were suggested in [Eswaren *et al.* 76] as a way of ensuring consistency of databases in the presence of concurrency and failures. While Davies discussed atomic actions in very general terms as a concept for controlling concurrency and failures, Eswaren *et al.* presented a practical implementation based on two-phase locking.

Meanwhile, work at the University of Newcastle on recovery blocks [Randell 75] investigated using nested atomic actions as a mechanism for localizing the effects of failures. Recovery techniques were explored in depth, particularly for building user-defined data types [Verhofstad 76, Anderson *et al.* 78]. The problems of concurrency, however, were addressed only for a limited class of data types (e.g., resource managers in [Shrivastava & Banatre 78], and objects with read and write operations in [Best & Randell 81, Best 82]). In addition, the work on concurrency assumed the use of a locking protocol, and did not consider other kinds of protocols.

As work on distributed systems began in earnest in the mid-1970's, attention was focused on atomic actions as a general way of reducing the complexity of coping with concurrency and failures. Work on distributed databases and file systems contributed many new protocols for implementing atomicity (see [Bernstein & Goodman 81] for a survey), including the novel techniques developed by Reed [Reed 78]. Reed's techniques represent the first detailed design for an implementation of *nested* atomic actions.

Since then, several projects (including the Argus project at MIT [Liskov & Scheifler 82], work at CMU [Schwarz & Spector 82], and the Clouds project at Georgia Tech [Allchin & McKendry 83]) have focused on nested atomic actions as a fundamental concept for organizing distributed systems. Moss [Moss 81], as part of the Argus project, developed a locking-based implementation of nested atomic actions. The Argus project has also explored incorporating atomic actions into a programming language. One of the major advantages of this approach is that a language provides a more flexible notion of data object than is supported by a database system or a file system. This dissertation explores how one might take advantage of

this flexibility.

As discussed earlier, our work on specifying atomic types generalizes earlier database-related work on concurrency control (e.g., [Papadimitriou 79]) in three ways: First, we treat both serializability and recoverability. Second, we explore how to analyze user-specified semantic information to achieve greater concurrency. Third, we focus on local properties of objects that ensure atomicity, exploring the conditions under which different kinds of objects can be combined while still ensuring atomicity.

Most of the theoretical work relevant to atomicity has focused on concurrency control, and has ignored problems of recovery. A notable exception is some work by Lynch [Lynch 83]. Lynch defines atomicity for nested actions, presents a formalization of the locking implementation in [Moss 81], and verifies that the (formalization of the) implementation ensures atomicity. While Lynch analyzes serializability and recoverability together, she does not consider data-dependent implementations, nor does she address modularity issues.

A few recent papers on concurrency control [Bernstein *et al.* 81, Korth 81a, Beeri *et al.* 83] address the problem of extending concurrency control protocols to cope with arbitrary user-defined operations. This research is limited in several ways, however. Most important is the lack of consideration of modularity issues. The focus of the work is on locking protocols, and the interactions among different kinds of protocols are not considered. In addition, the papers ignore recovery, and require the operations specified by the user to be functions. Non-determinism, which is often useful to avoid over-specifying abstractions, is not permitted.

We have not considered how to extend all existing protocols to cope with user-defined operations. For example, we have not analyzed optimistic protocols [Kung & Robinson 81] in any detail. The primary reason for this is that optimistic protocols, by their very nature, do not guarantee *internal consistency* [Goree 83], a property that prevents orphans [Nelson 81] from seeing inconsistent states. Our work on specifying atomic types does not rule out optimistic implementations, but we have not considered in detail how to build them.

Other protocols that we have not considered include non-two-phase locking protocols (e.g., see [Silberschatz & Kedem 80, Korth 81b]) and protocols for implementing replicated objects (e.g., see [Gifford 79]). Non-two-phase locking protocols, while useful for achieving greater concurrency, place strong restrictions on how data can be structured. These restrictions may be difficult to satisfy in the general class of applications considered here. Replication techniques, while essential for reliability and availability, are beyond the scope of this dissertation. Researchers at Cornell [Skeen & Birman 83] and MIT [Herlihy 84] are currently investigating how to replicate user-defined data objects.

There is some debate over whether atomicity is too strong a requirement, and whether it

permits adequate performance. Fischer [Fischer & Michael 82] has illustrated how greater concurrency can be achieved for a distributed directory by sacrificing serializability. The Grapevine system [Birrell et al. 82, Schroeder, et al. 84] also violates atomicity in places, primarily for performance reasons. It remains to be seen whether the performance advantages of violating atomicity are worth the resulting increase in complexity, and whether the protocols in [Fischer & Michael 82] and [Birrell et al. 82] can be generalized and applied to other systems.

Lamport [Lamport 76] has also argued that atomicity is too strong a requirement. As evidence for this claim, he presents an example of a banking system, with transfer and audit transactions. Locking implementations of this system do not perform well: audits can run for a long time, preventing transfers from running. Lamport presents an *ad hoc* protocol that ensures that transfers are serializable, and that each audit sees a consistent state of the database. His protocol permits some, though not all, transfers to run concurrently with an audit. Protocols developed since the publication of [Lamport 76], including those in [Reed 78], [DuBourdieu 82], and [Chan et al. 82], ensure serializability of all transactions (not just transfers). Furthermore, the protocols in [DuBourdieu 82, Chan et al. 82] permit all transfers to run concurrently with an audit, thus providing greater concurrency than Lamport's *ad hoc* protocol. Unlike Lamport's protocol, these protocols are easily applied in a large class of similar situations.

A detailed comparison of these and other papers with our work can be found in comments throughout the dissertation.

1.5 Roadmap

In chapters 2 through 4 we focus on specifications of atomic types. In Chapter 2, we present our formal model of computations and specifications. Then, in Chapter 3, we use the model to define atomicity of activities. Finally, in Chapter 4, we define three "local atomicity properties:" properties of individual objects that ensure atomicity of activities. The formal model used in these chapters does not permit activities to be nested. We expect it to be relatively easy to integrate our results with existing formal work on nested activities (e.g., [Lynch 83]), but do not do so in this dissertation.

In chapter 5, we connect the earlier material on specifications with the later material on implementations. In this chapter we focus on one of the local atomicity properties defined earlier, dynamic atomicity. We present an algorithmic description of a novel locking protocol for implementing dynamic atomic objects and prove that it is correct.

In Chapters 6 and 7, and in Appendix A, we discuss implementations of atomic types. In

Chapter 6, we discuss the problems that must be solved by implementations of atomic types, and informally discuss how to extend the material on specifications to cope with nested activities. Then, we present the approach taken in Argus [Liskov *et al.* 83] to cope with the problems involved in implementing atomic types, and illustrate several limitations of this approach. In Chapter 7 .we present an alternative approach that avoids some of the limitations of the Argus approach. The appendix contains several additional examples illustrating how to implement user-defined atomic types, and further illustrating the differences in the two approaches presented in Chapters 6 and 7.

Finally, in Chapter 8, we summarize our results and discuss further work.

An index for the technical terms defined in Chapters 2 through 5 is contained in Appendix B.

Chapter Two

System Model

In this chapter we describe our model of systems. We begin in Section 2.1 by describing the components of a system and defining computations. Then, in Section 2.2, we describe how we model specifications of objects.

2.1 Computations and Observations

We view a system as composed of activities and objects. *Activities* correspond roughly to processes or threads of control: they are the active entities in the system, and perform tasks for users. *Objects* contain the state of the system: they provide operations by which activities can examine and modify the system state, and constitute the sole path by which activities can pass information among themselves. We will typically use the symbols a, b, and c (possibly subscripted) for activities, and the symbols x, y, and z (again possibly subscripted) for objects.

We use an event-based model of computation. In general, the events in which we are interested are events that occur at the interface between objects and activities. For the remainder of this chapter, Chapter 3, and the first part of Chapter 4, we assume that an event is either the invocation of an operation on an object by an activity, the termination of an invocation, the commit (successful completion) of an activity at an object, or the abort (unsuccessful completion) of an activity at an object. A note on terminology: We will use the term "termination" to mean the end of the execution of a single operation, and the term "completion" to mean the end of the execution of an entire activity. Each event identifies the activity and the object that participate in it. If an activity (object) participates in an event, we say that the event *involves* the activity (object). In Chapters 4.2 and 4.3 we will augment our model with additional events that introduce information about timestamps for activities.

For example, suppose x is an object that is intended to behave like a set of integers, with operations to insert an integer in x, to delete an integer, and to check for membership. If a is an activity, example events include the following:

- a invokes insert on x with argument 3 (written $\langle \text{insert}(3),x,a \rangle$)

- an invocation of an operation by a on x terminates with result "ok" (written $\langle \text{ok},x,a \rangle$)

- a invokes member on x with argument 7 (written $\langle \text{member}(7),x,a \rangle$)

- an invocation of an operation by a on x terminates with result "true" (written

⟨true,x,a⟩

- *a* commits at *x* (written ⟨commit,x,a⟩)

A computation is most properly viewed as a partial order of events. For our purposes it suffices to restrict our attention to the observable behavior of a system. We model an observation of a system as a finite sequence of events.

For example, if *a* and *b* are activities, the following event sequence might be a observation of a system containing a set object *x*:

$$⟨insert(3),x,a⟩$$
$$⟨ok,x,a⟩$$
$$⟨member(3),x,b⟩$$
$$⟨commit,x,a⟩$$
$$⟨true,x,b⟩$$
$$⟨commit,x,b⟩$$

If *h* is an event sequence and *X* is a set of objects, we define $h|X$ ("*h* restricted to *X*") to be the subsequence of *h* consisting of all events in which objects in *X* participate. We define $h|A$ similarly for a set of activities *A*. If *x* is an object and *a* is an activity, we write $h|x$ for $h|\{x\}$, and $h|a$ for $h|\{a\}$. We also define *committed(h)* to be the set of activities that commit in *h*, and *aborted(h)* to be the set of activities that abort in *h*. Finally, we define *completed(h)* to be the set of activities that complete (commit or abort) in *h*; i.e., *completed(h)* = *committed(h)* $\cup$ *aborted(h)*.

We will use the following notation for sequences: The symbol "•" denotes concatenation of sequences, and the symbol "Λ" denotes the empty sequence.

We include here two technical lemmas. The first lemma asserts the commutativity of the restriction operators:

Lemma 2-1: Suppose *X* and *Y* are sets of objects, and *A* and *B* are sets of activities. Then

1. $(h|X)|Y = (h|Y)|X = h|(X \cap Y)$

2. $(h|A)|B = (h|B)|A = h|(A \cap B)$

3. $(h|X)|A = (h|A)|X$

The second lemma asserts the independence of the restriction operators and the concatenation operator on sequences:

Lemma 2-2: Suppose *S* is a set of objects, and *h* and *k* are event sequences. Then

$$(h•k)|S = (h|S) • (k|S)$$

The same property holds when *S* is a set of activities.

We will be interested in those event sequences that are complete in the following sense: An event sequence h is *complete* if every activity either completes at every object or does nothing at the object; i.e., for all a and x, either $a \in completed(h|x)$, or $h|a|x = \Lambda$.

Not all event sequences make sense as observations: activities are intended to act like sequential processes. (Concurrency within an activity should be achieved by using nested activities; our formal analysis of atomicity does not cover nesting.) Thus, we restrict our attention to event sequences h satisfying the following conditions:

- An activity must wait until one invocation terminates before invoking another operation. More precisely, let $op\text{-}events(h)$ be the subsequence of h consisting of all invocation and termination events; then $op\text{-}events(h|a)$ must consist of an alternating sequence of invocation and termination events, beginning with an invocation event. In addition, an invocation event and the immediately succeeding termination event must involve the same object.

- No activity both commits and aborts in h (at the same or different objects); i.e., $commit(h) \cap abort(h) = \emptyset$.

- An activity cannot commit if it is waiting for an invocation to terminate, and an activity cannot invoke any operations after it commits. More precisely, if $a \in committed(h)$, then $h|a$ consists of an alternating sequence of invocation and termination events, ending in a termination event, followed by some number of commit events.

Such "well-formed" event sequences will be called *histories*; in the remainder of this dissertation we will be concerned only with histories, not with arbitrary event sequences.

These restrictions on activities are intended to model the typical use of atomic activities in existing systems. An activity executes by invoking operations on objects, receiving results when the operations terminate. Since we disallow concurrency within an activity, an activity is permitted at most one pending invocation at any time. After successful termination of all invocations, an activity can commit at one or more objects.

We make very few restrictions on aborted activities; for example, an activity can continue to invoke operations after it has aborted. We have two reasons for avoiding additional restrictions. First, we have no need for them in our analysis. Second, and most important, additional restrictions might be too strong to model systems with orphans [Nelson 81, Goree 83], and we would like our results to be as generally applicable as possible.

An activity is not allowed to commit at some objects and abort at others; this requirement, called *atomic commitment*, can be implemented using a commitment protocol such as two-phase commit [Gray 78, Lampson 81] or three-phase commit [Skeen 82].

2.2 Specifications

Our specifications take the form of sets of sequences. A set of sequences is like a language, and can be conveniently described by a machine. In Section 2.2.1 we define *state machines*, which we will use for describing specifications. Then, in Section 2.2.2, we define our model for specifications of objects. Finally, in Section 2.2.3, we describe how properties of a system can be inferred from the specifications of its components.

2.2.1 State Machines

Informally, a state machine consists of a collection of states and a collection of transitions. The transitions can be used to change the state of the machine. A step of the machine consists of a single transition; the machine executes steps one at a time.

We begin with some notation. We use the notation $\rightarrow_p$ to denote a partial function. The symbol $\perp$ denotes "undefined," and will be used to indicate when a partial function is not defined for a given set of arguments. The symbol $\bullet$, as mentioned earlier, denotes concatenation of sequences.

Formally, a state machine M consists of: a *state domain* S_M; an *initial state* $I_M \in S_M$; a collection of *transitions* T_M; and a *partial transition function* $N_M: S_M \times T_M \rightarrow_p S_M$.

The transition function N_M can be extended to finite sequences of transitions in the obvious way; i.e., if T is a transition and Tseq is a finite sequence of transitions, then:

$$N_M(S, \Lambda) = S$$

$$N_M(S, \text{Tseq} \bullet T) = N_M(N_M(S, \text{Tseq}), T), \text{ if } N_M(S, \text{Tseq}) \neq \perp$$
$$\perp, \text{ otherwise}$$

If Tseq is a sequence of transitions for machine M, we will sometimes use the notation Tseq(S) for $N_M(S, \text{Tseq})$.

If Tseq(S) $\neq \perp$, we say that Tseq is *defined in* S. We say that a sequence of transitions Tseq is *accepted* by a machine M if Tseq is defined in I_M.

We may easily associate a set of sequences with a machine: Given a machine M, define the *language* of M (denoted L(M)) to be all finite sequences of operations that are accepted by M.

Our definition of a state machine differs slightly from the usual definition of an automaton [Ginsburg 75]. Rather than introducing the notion of an *accepting state*, and then saying that a sequence Tseq is accepted by a machine M if Tseq(I_M) is an accepting state, we have found it convenient for the examples that we will present to define acceptance as above. Not all languages (or even all recursive languages) can be defined by one of our state

machines. however. It follows from our definition of acceptance that the language of a state machine is *prefix-closed*: If a sequence *s* is in the language of a machine M, then every finite prefix of *s* is also in the language of M.

An example of a description of a machine appears in Figure 2-1; the language of the machine consists of all finite alternating sequences of a's and b's. The transition function is described by giving, for each transition, a precondition describing the set of states in which it is defined, and a list of state changes caused by the transition.

States: {a, b, i} **initially i**

Transitions: {a, b}
    N(s,a):
        when s = a or s = i
        changes s to b

    N(s,b):
        when s = b or s = i
        changes s to a

Figure 2-1: An example machine.

The state set of this machine is {a,b,i}. In later examples we will describe *state components*, giving a name and domain for each component. This indicates that the state set is the cartesian product of the sets listed; we will refer to the component *name* of state *s* as *s.name*. In descriptions of machines, we use "**when** <expr>," where <expr> is a boolean expression, to describe preconditions for transitions. If the precondition is omitted from the description, it is assumed to be *true*. We use "**changes** <list>," where each item in <list> has the form "<state component> **to** <expr>," to describe the relationship between the state before a transition and the state after the transition. Components not listed are assumed to be unchanged. We will also use the form "**if** <expr> **then** <changes> **else** <changes>" to describe conditional changes.

2.2.2 Specifications of Objects

Assuming that all executions of a system are atomic, consistency is preserved if every activity preserves consistency when executed in isolation with no failures. To check this property we only need to know how each object and activity behaves in a sequential, failure-free environment. We have found it convenient to specify an object by first describing the object's sequential, failure-free behavior, and then describing how the object controls concurrency and failures to ensure atomicity. We reflect this two-stage process directly in our model: A specification of an object *x* consists of two parts, the *serial specification* (denoted *x.serial*),

and the *behavioral specification* (denoted *x.behavior*). The serial specification of an object is intended to model the acceptable behavior of the object in a sequential, failure-free environment, while the behavioral specification describes how the object supports atomicity.

The behavioral specification of an object *x* describes the acceptable histories involving *x*, and consists of a set of complete histories *h* such that every event in *h* involves *x*. In Section 2.2.3 we will describe how the behavioral specifications of objects constrain the behavior of a system.

It is convenient for the serial specification of an object to be in a slightly different form from its behavioral specification. Instead of a set of complete histories, we will use a set of *operation sequences*. An *operation* is a pair consisting of an invocation and a termination event. In addition, an operation identifies the object on which it is executed. An operation does *not* identify an activity; we have found no need for the serial specification of an object to vary depending on which activity executes an operation, and indeed find it more convenient to describe the serial specification in a way that is independent of activities.

We often speak informally of an "operation" on an object, as in "the insert operation on a set object." An *operation* in our formal model is intended to represent a single execution of an "operation" as used in the informal sense. For example, the following might be an operation (in the formal sense) on a set object *x*:

<div align="center">x:<insert(3),ok></div>

This operation represents an execution of insert on *x* with argument "3" and result "ok."

A state machine is a convenient tool for describing the serial specification of an object. We define the transitions of a machine to be the operations on the object, and choose a transition function such that the language of the machine is the desired set of operation sequences. For example, the state machine in Figure 2-2 describes the serial specification of a set object *x*. A set object provides three operations: *insert*, *delete*, and *member*. *Insert* adds a specified item to the set object. *Delete* removes a specified item from the object. *Member* determines whether a specified item is an element of the set object.

The reader may check that the following operation sequence is in the language of the state machine in Figure 2-2:

<div align="center">x:<insert(3),ok>
x:<member(3),true></div>

The following sequence, however, is not:

<div align="center">x:<insert(3),ok>
x:<member(3),false></div>

Another example, the serial specification of a semiqueue object *y*, appears in Figure 2-3. A semiqueue object provides two operations: *enq* and *deq*. *Enq* adds a specified item to the

States: sets of items initially ∅

Transitions: {x:⟨insert(i),ok⟩, x:⟨delete(i),ok⟩, x:⟨member(i),b⟩: i is an
item and b is a boolean}

N(s,x:⟨insert(i),ok⟩):
changes s to s∪{i}

N(s,x:⟨delete(i),ok⟩):
changes s to s-{i}

N(s,x:⟨member(i),true⟩):
when i∈s

N(s,x:⟨member(i),false⟩):
when i∉s

Figure 2-2:Serial specification of a set object x.

semiqueue object. *Deq* nondeterministically chooses an item in the semiqueue, deletes it, and returns it; *deq* is not defined if the semiqueue is empty. Semiqueues are like multisets, rather than sets, in that an item can appear more than once. We will return to this example in later chapters, first precisely describing the concurrency that can be permitted by an object with this serial specification, and then illustrating how semiqueues can be implemented to achieve this level of concurrency.

States: multisets of items initially ∅

Transitions: {y:⟨enq(i),ok⟩, y:⟨deq,i⟩: i is an item}
N(s,y:⟨enq(i),ok⟩):
changes s to s∪{i}

N(s,y:⟨deq,i⟩):
when i∈s
changes s to s-{i}

Figure 2-3:Serial specification of a semiqueue object y.

2.2.3 Behavior of a System

The behavior of a system is determined by the behaviors of its components and the interconnections among the components. The specification of a component constrains the behavior of the component, and indirectly constrains the behavior of the system. In this section we describe how constraints on the behavior of a system can be inferred from the specifications of the system's components.

The behavioral specification of an object x, as described above, is a set of complete histories involving x. Similarly, we define the behavioral specification of an activity a (denoted $a.behavior$) to be a set of complete histories involving a.

We define the behavior of a system to be all complete histories h such that, for all a and x, $h|a \in a.behavior$ and $h|x \in x.behavior$. In other words, any history of a system must be permitted by the specification of each of the system's components. Notice that the behavioral specification of each component describes how the component constrains the occurrence of events in which it participates, and places no constraints on the occurrence of other events.

Many papers have been published on models for distributed systems. Recently, Stark [Stark 84] has been studying how to model specifications of modules in distributed and concurrent systems. He has developed a general framework incorporating notions of composition and abstraction; in particular, his notion of "observations" is similar to ours, as is his notion of composition of behaviors for components of a system. However, he does not consider well-formedness conditions on observations specific to any particular applications; in addition, his desire to be able to specify and to model eventuality properties forces him to consider infinite observations, complicating his framework. A framework such as his, however, could be used to develop a more rigorous formalization of our model and associated results.

Chapter Three

Global Atomicity

In this chapter we develop a formal definition of atomicity. Our goal is to understand how objects can cooperate to ensure that all complete histories in a system's behavior are atomic. The definitions in this chapter apply to *all* histories, however, not just to *complete* histories. In Chapter 4 we will concentrate on complete histories, and will look at several different ways in which objects can cooperate to ensure atomicity of complete histories. In Chapter 5 we will consider prefixes of complete histories, and illustrate how an implementation of an object can ensure atomicity in an on-line manner.

The remainder of this chapter is divided into two sections. In the first, we develop our definition of atomicity. Then, in the second, we compare our definition to common definitions of serializability in the literature, illustrating some limitations of the model used in those definitions.

3.1 Definitions

Informally, a history of a system is atomic if it is equivalent to a sequential, failure-free execution of the committed activities in the history. The serial specifications of objects describe the acceptable behavior of the system in a sequential, failure-free environment. Since serial specifications are sets of operation sequences, not sets of histories, we need to establish a correspondence between histories and operation sequences. The paragraph below provides the necessary definitions.

We say that a history is *serial* if events for different activities are not interleaved. If h is a serial history, and $a_1, ..., a_n$ are the activities in h in the order in which they appear, then we can write h as $h|a_1 \cdot ... \cdot h|a_n$. We say that a history h is *failure-free* if $aborted(h) = \emptyset$. Now, if h is a serial failure-free history, we define $opseq(h)$ as follows: $Opseq(h|a)$, for an activity a, is the operation sequence obtained from $h|a$ by pairing each invocation event with its corresponding termination event, and discarding commit events and pending invocation events. Let $a_1, ..., a_n$ be the activities in h in the order in which they appear; then $opseq(h)$ is defined to be $opseq(h|a_1) \cdot ... \cdot opseq(h|a_n)$. $Opseq(h)$, for a serial failure-free history h, is the operation sequence corresponding to h.

For example, if h is the serial failure-free history

$$\langle insert(3),x,b\rangle$$
$$\langle ok,x,b\rangle$$
$$\langle commit,x,b\rangle$$
$$\langle member(3),x,a\rangle$$
$$\langle true,x,a\rangle$$
$$\langle commit,x,a\rangle$$

then $opseq(h)$ is the operation sequence

$$x:\langle insert(3), ok\rangle$$
$$x:\langle member(3), true\rangle$$

Similarly, if h is the serial failure-free history

$$\langle insert(3),x,a\rangle$$
$$\langle ok,x,a\rangle$$
$$\langle delete(2),x,a\rangle$$

then $opseq(h)$ is the operation sequence

$$x:\langle insert(3), ok\rangle$$

Notice that we do not restrict the domain of $opseq$ to complete histories; we will need to apply it to incomplete histories in Chapter 5.

We say that two histories h and k are *equivalent* if every activity performs the same steps in h as in k; i.e., if $h|a = k|a$ for every activity a. We also say that a serial failure-free history h is *acceptable at x* if $opseq(h|x) \in x.serial$; in other words, if the sequence of operations in h involving x is permitted by the serial specification of x. A serial failure-free history is *acceptable* if it is acceptable at every object x.

For example, suppose that x is an integer set object with a serial specification as given in Figure 2-2. If h is the serial failure-free history

$$\langle insert(3),x,b\rangle$$
$$\langle ok,x,b\rangle$$
$$\langle commit,x,b\rangle$$
$$\langle member(3),x,a\rangle$$
$$\langle true,x,a\rangle$$
$$\langle commit,x,a\rangle$$

then $opseq(h)$ is the operation sequence

$$x:\langle insert(3),ok\rangle$$
$$x:\langle member(3),true\rangle$$

which is in $x.serial$. Thus, h is acceptable. On the other hand, if h is the history

$$\langle insert(3),x,b\rangle$$
$$\langle ok,x,b\rangle$$
$$\langle commit,x,b\rangle$$
$$\langle member(3),x,a\rangle$$
$$\langle false,x,a\rangle$$
$$\langle commit,x,a\rangle$$

then h is not acceptable, since $opseq(h)$ is not in $x.serial$.

If h is a history and T is a total order on activities, we define $serial(h, T)$ to be the serial history equivalent to h in which activities appear in the order T. If $a_1, ..., a_n$ are the activities in h in the order T. then $serial(h, T) = h|a_1 \bullet ... \bullet h|a_n$.

The following lemma asserts the independence of the serializing operator and the restriction operator.

Lemma 3-1: If h is a history, X is a set of objects. and T is a total order on activities, then $serial(h|X, T) = serial(h, T)|X$.

Proof: Let $a_1, ..., a_n$ be the activities in h in the order T. The following equalities show the desired result:

$$
\begin{aligned}
serial(h|X, T) &= (h|X)|a_1 \bullet ... \bullet (h|X)|a_n \\
&= (h|a_1)|X \bullet ... \bullet (h|a_n)|X \\
&= (h|a_1 \bullet ... \bullet h|a_n)|X \\
&= serial(h, T)|X
\end{aligned}
$$

The first line follows from the definition of $serial$. The second line follows from Lemma 2-1, and the third line from Lemma 2-2. The fourth line again follows from the definition of $serial$.

If T is a total ordering of activities, we then say that a failure-free history h is *serializable in the order T* if $serial(h, T)$ is acceptable. We say that a failure-free history h is *serializable* if there exists a total order T on activities such that h is serializable in the order T. In other words, a failure-free history is serializable if it is equivalent to an acceptable serial history.

For example, if h is the failure-free history

<member(3),x,a>
<insert(3),x,b>
<ok,x,b>
<true,x,a>
<commit,x,b>
<commit,x,a>

and T is a total order in which b precedes a, then h is serializable in the order T: $Serial(h,T)$ is the history

<insert(3),x,b>
<ok,x,b>
<commit,x,b>
<member(3),x,a>   .
<true,x,a>
<commit,x,a>

and, as illustrated above, this history is acceptable.

Note that serializability is defined for all failure-free histories, not just those that are complete. We will consider incomplete histories in Chapter 5.

Lemma 3-2 describes the relation between serializability of a history and serializability of its subhistories at each object.

> **Lemma 3-2:** If *h* is a history and *T* is a total order on activities, *h* is serializable in the order *T* if and only if, for every object *x*, *h|x* is serializable in the order *T*.
>
> **Proof:** Follows easily from the definitions and Lemma 3-1.

Now, define *permanent(h)* to be *h|committed(h)*. We then say that *h* is *atomic* if *permanent(h)* is serializable. Thus, we formalize recoverability by throwing away events for non-committed activities, and requiring that the committed activities be serializable.

For example, if *x* is an integer set as above and *h* is the history

<center>
<member(3),x,a>
<insert(3),x,b>
<ok,x,b>
<true,x,a>
<commit,x,b>
<delete(3),x,c>
<ok,x,c>
<commit,x,a>
<abort,x,c>
</center>

then *permanent(h)* is the failure-free history

<center>
<member(3),x,a>
<insert(3),x,b>
<ok,x,b>
<true,x,a>
<commit,x,b>
<commit,x,a>
</center>

which, as illustrated above, is serializable. Thus, *h* is atomic.

On the other hand, the history

<center>
<member(2),x,a>
<true,x,a>
<commit,x,a>
</center>

is not atomic, since *x.serial* does not contain the sequence

<center>
x:<member(2), true>
</center>

The final lemma asserts that *permanent* commutes with restriction to an object for complete histories.

> **Lemma 3-3:** If *h* is a complete history and *x* is an object, *permanent(h)|x* = *permanent(h|x)*.
>
> **Proof:** We show that *permanent(h)|x* is a subsequence of *permanent(h|x)*; a similar argument shows that *permanent(h|x)* is a subsequence of *permanent(h)|x*. Since both histories are subsequences of *h*, it suffices to show that every event appearing in *permanent(h)|x* also appears in *permanent(h|x)*. Suppose *e* is an

event in *permanent(h)|x*. Let *a* be the activity that participates in *e*. By the definition of restriction, *x* must also participate in *e*. Thus, *e* also appears in *h|x*. Since *e* appears in *permanent(h)|x*, *a* must commit in *h*; since *h* is complete, and *h|a|x* ≠ Λ, *a* must commit in *h|x*. Thus, *e* appears in *permanent(h|x)*.

Our definition of atomicity is similar to the definition of serializability in [Papadimitriou 79], where it is assumed that some underlying recovery mechanism handles aborts of activities, and the formal analysis considers only events for committed activities. It is different in that we include events for aborted and active activities in our formal model; as we will discuss in Chapter 5, this facilitates the precise description of online support for recoverability. It also differs in that the definition of serializability is based on user-supplied specifications of the acceptable serial behavior of objects, rather than a free interpretation as in [Papadimitriou 79]. As we will discuss in Section 4.4.1, this enables us to achieve more concurrency.

3.2 Limitations of the Scheduler Model

Our model is slightly different from that used in much of the literature on concurrency control, including those papers that consider user-specified operations (e.g., [Papadimitriou 79, Bernstein *et al.* 81, Korth 81a, Beeri *et al.* 83]). That model, which we will call the *scheduler model*, is pictured in Figure 3-1.
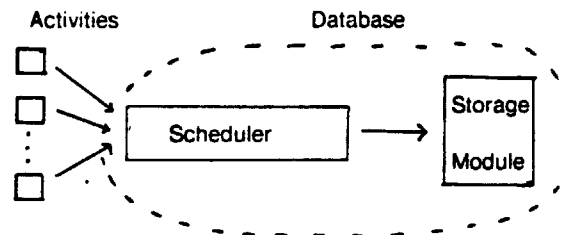
Figure 3-1: The scheduler model.

The boxes on the left represent transactions, which submit invocations to the scheduler in the middle. The scheduler determines the order in which to run operations invoked by transactions, and submits the invocations in that order to the storage module on the right, which processes the operations and returns their results to the transactions. (The storage module behaves similarly to the state machines that we use to describe serial specifications.) The problem addressed in the papers cited above is to analyze the properties of the scheduler module. The problem that we address is slightly different: we analyze the properties of the interface represented by the dotted line.

The scheduler model imposes unnecessary limitations on the problem, as the following example illustrates. Let z be a first-in-first-out queue object with a serial specification as described by the machine in Figure 3-2. A FIFO queue object provides two operations: *enq* and *deq*. *Enq* appends a specified item to the back of the queue. *Deq* removes and returns the item at the front of the queue; if the queue is empty, *deq* signals *empty*. Now consider the following history:

$$\langle enqueue(1),z,a \rangle$$
$$\langle ok,z,a \rangle$$
$$\langle enqueue(2),z,b \rangle$$
$$\langle ok,z,b \rangle$$
$$\langle commit,z,a \rangle$$
$$\langle commit,z,b \rangle$$
$$\langle dequeue,z,c \rangle$$
$$\langle 2,z,c \rangle$$
$$\langle dequeue,z,c \rangle$$
$$\langle 1,z,c \rangle$$
$$\langle commit,z,c \rangle$$

Note that this history is atomic: The equivalent history with a, b, and c in the order b-a-c is acceptable.

States: sequences of items initially Λ

Transitions: $\{z:\langle enq(i),ok \rangle, z:\langle deq,i \rangle, z:\langle deq,empty \rangle: i$ is an item$\}$

    $N(s,z:\langle enq(i),ok \rangle)$:

        changes s to s•i

    $N(s,z:\langle deq,i \rangle)$:

        when s = i•s'

        changes s to s'

    $N(s,z:\langle deq,empty \rangle)$:

        when s = Λ

Figure 3-2: Serial specification of a FIFO queue object z.

Now consider what happens in the scheduler model. This history cannot be produced using the scheduler model (assuming that the order of termination events corresponds to the scheduling order). If it were, the state of the storage module after a and b commit would be 1•2 (reading from front to back); then c would have to receive 1 before 2, not 2 before 1. Thus, the scheduler model restricts the histories that can occur, and indeed rules out some histories that appear "atomic."

The cause of this limitation of the scheduler model is its fixed internal structure. It imposes a fixed interpretation on all histories, based on the interface between the scheduler and the

storage module. In contrast. we do not interpret events in terms of some lower-level model of execution. *Opseq* provides a kind of "operational" interpretation, but we use it only for serial histories, not for all histories.

The scheduler model was intended to be used to study the concurrency control problem, which is but one aspect of the more general problem of ensuring atomicity. Our model was designed to be used to study atomicity in as general a setting as possible: thus, we needed to make our model as abstract as possible. This means that we avoid the limitations of the scheduler model illustrated above, but also means, since our model incorporates less fixed structure, that it may be more difficult to verify implementations.

Chapter Four

Local Atomicity Properties

We are interested in ways of ensuring that all possible histories of a system are atomic. As discussed in Chapter 2, the histories of a system are constrained by the specifications of the components of the system. In this chapter we investigate several properties of individual objects that ensure atomicity of activities using the objects. We call such properties local atomicity properties. More precisely, a *local atomicity property* is a property P of specifications of objects such that the following is true: If the specification of every object in a system satisfies P, then every history in the system's behavior is atomic.

The problem that must be solved in designing a local atomicity property is to ensure that the objects agree on at least one serialization order for the committed activities. Solving this problem can be difficult because each object is aware of only the events in which it participates. In other words, each object has purely *local* information; no object has complete information about the *global* computation of the system.

As discussed in Chapter 1, there are many different protocols that ensure agreement among objects, and these protocols are not always compatible. In this chapter we present three different local atomicity properties, highlighting the way in which agreement is reached. We also show that each property is optimal, in a sense to be defined below. Our optimality results imply that no local property is both necessary and sufficient for global atomicity.

The three properties presented in this chapter provide formal characterizations of the behavior of three different classes of protocols, exemplified by the three types of protocols (two-phase locking, multi-version timestamping, and hybrid methods) discussed in Chapter 1. Each of the properties is based on user-supplied specifications of the acceptable serial behavior of objects, thus permitting implementations that achieve greater concurrency than is possible when operations are simply characterized as reads and writes.

The remainder of this chapter is divided into four sections. In each of the first three we present a different local atomicity property. In the fourth we conclude with some remarks on classes of atomic types and related work.

4.1 Dynamic Atomicity

Two-phase locking protocols [Eswaren *et al.* 76, Bernstein *et al.* 81, Korth 81a] determine a serialization order for activities *dynamically*, based on the order in which activities invoke operations and obtain locks on objects. Our first local atomicity property characterizes the behavior of protocols, including two-phase locking protocols, which are dynamic in this sense. We call this property *dynamic atomicity*.

Informally stated, the fundamental property of protocols characterized by dynamic atomicity is the following: If the sequence of operations executed by one committed activity conflicts with the operations executed by another committed activity, then some of the operations executed by one of the activities must occur after the other activity has committed. Locking protocols (and all pessimistic protocols) achieve this property by *delaying* conflicting operations; optimistic protocols [Kung & Robinson 81] achieve this property by allowing conflicts to occur, but *aborting* conflicting activities to prevent conflicts among committed activities.

The remainder of this section is divided into two subsections. In the first, we present dynamic atomicity and prove that it is a local atomicity property. In the second, we define optimality, and show that dynamic atomicity is optimal. In Chapter 5, we will examine dynamic atomicity in greater detail, describing locking-based implementations and verifying that dynamic atomicity indeed characterizes their behavior. We will also illustrate the limitations of locking, showing that there are useful non-locking protocols that are characterized by dynamic atomicity.

4.1.1 Definition of Dynamic Atomicity

We can describe dynamic atomicity precisely as follows. If h is a history, define *precedes*(h) to be the following relation on activities: $\langle a,b \rangle \in precedes(h)$ if and only if there exists an operation invoked by b that terminates after a commits. The events need not occur at the same object. The relation *precedes*(h) captures the concept of an operation being delayed: If $\langle a,b \rangle \in precedes(h)$, then some operation executed by b was delayed in h until after a committed.

For example, if x is a set object as before, and h is the history

$$\langle insert(2),x,a \rangle$$
$$\langle ok,x,a \rangle$$
$$\langle member(3),x,b \rangle$$
$$\langle false,x,b \rangle$$
$$\langle commit,x,b \rangle$$
$$\langle commit,x,a \rangle$$

then *precedes*(h) is the empty relation, while if h is the history

$$\langle insert(2),x,a\rangle$$
$$\langle ok,x,a\rangle$$
$$\langle member(3),x,b\rangle$$
$$\langle commit,x,a\rangle$$
$$\langle false,x,b\rangle$$
$$\langle commit,x,b\rangle$$

then *precedes*(h) contains the pair $\langle a,b\rangle$. Note that, for any history h, *precedes*(h) is a partial order.

The following lemma provides the key to our definition of dynamic atomicity.

Lemma 4-1: If h is a history and x is an object, then *precedes*(h|x) ⊆ *precedes*(h).

If h is a history of the system, each object has only partial information about *precedes*(h). However, if each object x ensures local serializability in *all* orders consistent with *precedes*(h|x), we are guaranteed global serializability in all orders consistent with *precedes*(h). To be precise, we have the following definition of dynamic atomicity: We say that a history h is *dynamic atomic* if *permanent*(h) is serializable in every total order consistent with *precedes*(h). In other words, every serial history equivalent to *permanent*(h), with the activities in an order consistent with *precedes*(h), must be acceptable.

For example, the following history h is atomic, but not dynamic atomic:

$$\langle member(3),x,a\rangle$$
$$\langle insert(3),x,b\rangle$$
$$\langle ok,x,b\rangle$$
$$\langle false,x,a\rangle$$
$$\langle member(3),x,c\rangle$$
$$\langle commit,x,b\rangle$$
$$\langle true,x,c\rangle$$
$$\langle commit,x,a\rangle$$
$$\langle commit,x,c\rangle$$

Permanent(h), which is the same as h, is equivalent to the following acceptable serial history:

$$\langle member(3),x,a\rangle$$
$$\langle false,x,a\rangle$$
$$\langle commit,x,a\rangle$$
$$\langle insert(3),x,b\rangle$$
$$\langle ok,x,b\rangle$$
$$\langle commit,x,b\rangle$$
$$\langle member(3),x,c\rangle$$
$$\langle true,x,c\rangle$$
$$\langle commit,x,c\rangle$$

and thus is serializable in the order a followed by b followed by c (written a-b-c). However, since *precedes*(h) contains only the single pair $\langle b,c\rangle$, *permanent*(h) must also be serializable in the orders b-a-c and b-c-a for h to be dynamic atomic. This is not the case; for example, the serial history

<center>

⟨insert(3),x,b⟩
⟨ok,x,b⟩
⟨commit,x,b⟩
⟨member(3),x,a⟩
⟨false,x,a⟩
⟨commit,x,a⟩
⟨member(3),x,c⟩
⟨true,x,c⟩
⟨commit,x,c⟩

</center>

is not acceptable.

As another example, the history

<center>

⟨member(2),x,a⟩
⟨insert(3),x,b⟩
⟨ok,x,b⟩
⟨false,x,a⟩
⟨member(3),x,c⟩
⟨commit,x,b⟩
⟨true,x,c⟩
⟨commit,x,a⟩
⟨commit,x,c⟩

</center>

is dynamic atomic. *Precedes(h)* contains the single pair ⟨*b,c*⟩, and *permanent(h)* is serializable in the orders *a-b-c*, *b-a-c*, and *b-c-a*.

We say that an object *x* is *dynamic atomic* if every history in *x.behavior* is dynamic atomic. The following theorem justifies our claim that dynamic atomicity is a local atomicity property:

Theorem 4-2: If every object in a system is dynamic atomic, then every history in the system's behavior is atomic.

Proof: Suppose every object in a system is dynamic atomic, and let *h* be a history of the system. *Precedes(h)* is a partial order, so let *T* be a total order of the activities in *h* that is consistent with *precedes(h)*. By Lemma 4-1, *precedes(h|x)* ⊆ *precedes(h)*, so *T* is also consistent with *precedes(h|x)* for every *x*. Since each object is dynamic atomic, *permanent(h|x)* is serializable in every total order consistent with *precedes(h)*; in particular, it is serializable in the order T. By Lemmas 3-3 and 3-2, *permanent(h)* is serializable in the order T. Thus, *h* is atomic.

4.1.2 Optimality

Dynamic atomicity is *optimal*: there is no other local atomicity property that is strictly more permissive. The paragraphs below state this result precisely and provide a proof. We caution the reader, however, that "optimal" does not mean "best." As we will see in Section 4.2, there are other local atomicity properties that are also optimal, yet are different: Each allows specifications not allowed by the others. Also, the optimality of a local atomicity property depends on the events in the underlying model: a property that is optimal in one model may

be suboptimal in a model with more events. if the additional events provide objects with useful information about the execution of the system. We will see an example of this latter situation in Section 4.3. With these caveats in mind. we now present the theorem.

As defined earlier, a local atomicity property P is any property P of specifications of objects such that the following is true: If the specification of every object in a system satisfies P, then every history in the system's behavior is atomic. We say that a property P is *weaker* (or *more permissive*) than a property Q if every specification that satisfies Q also satisfies P; in other words, if Q implies P. A property P is *strictly weaker* than a property Q if P is weaker than Q, and Q is not weaker than P. Note that if P is strictly weaker than Q, then there exists some specification that satisfies P and does not satisfy Q. If we equate the level of concurrency permitted by a property with the behaviors permitted by the property for a given serial specification, then if P is strictly weaker than Q, P permits more concurrency than Q.

We say that a local atomicity property P is *optimal* if no local atomicity property is strictly weaker than P. The following theorem shows that dynamic atomicity is optimal:

> **Theorem 4-3:** For systems using the events described in this section, dynamic atomicity is optimal.
>
> **Proof:** The proof proceeds by contradiction: Suppose dynamic atomicity is not optimal. Then there exists a local atomicity property P that is strictly weaker than dynamic atomicity. We will exhibit a system composed of objects satisfying P, and show that there is a non-atomic history of that system, thus contradicting the claim that P is a local atomicity property.

Since P is more permissive than dynamic atomicity, there must be a specification S_x of an object x such that S_x satisfies P but x is not dynamic atomic. In particular, there must be at least one history h_x in $x.behavior$ that is not dynamic atomic; that is, such that $permanent(h_x)$ is not serializable in at least one total order T consistent with $precedes(h_x)$. We will construct an object y whose specification contains a history h_y involving the committed activities in h_x. Now, consider a system containing x, y, and all the activities in h_x. We will choose h_y so that h_y is serializable only in the order T, and there is a history h of this system such that $h|x = h_x$ and $h|y = h_y$. Since $permanent(h_y)$ is only serializable in the order T, and $permanent(h_x)$ is not serializable in that order, it follows from Lemma 3-2 that $permanent(h)$ is not serializable. Thus, h is not atomic.

Construction of y:

The construction of y is as follows: Let y have a single operation called increment. Y is intended to behave as a counter: Its state is initially zero, and each invocation of the increment operation increments the state of y and returns the resulting

value. A state machine describing the serial specification of *y* appears in Figure 4-1.

States: integers initially 0

Transitions: {y:<increment,i>: i is an integer}
    N(s,y:<increment,i>):
        when i = s + 1
        changes s to s + 1

Figure 4-1: Serial specification of a counter object *y*.

The operation sequences in *y.serial* have the following form:

$$y:<increment,1>$$
$$y:<increment,2>$$
$$...$$
$$y:<increment,n>$$

Let *y.behavior* be the largest set of histories such that *y* is dynamic atomic; i.e., let *y.behavior* contain all dynamic atomic histories *h* such that $h = h|y$. Since *P* is weaker than dynamic atomicity, S_y satisfies *P*.

Choice of h_y:

Now, let *a1, a2, ..., an* be the committed activities in h_x in the order *T*, and let h_y be the following serial history in *y.behavior*:

$$<increment,y,a1>$$
$$<1,y,a1>$$
$$<commit,y,a1>$$
$$<increment,y,a2>$$
$$<2,y,a2>$$
$$<commit,y,a2>$$
$$...$$
$$<increment,y,an>$$
$$<n,y,an>$$
$$<commit,y,an>$$

Note that h_y is serializable only in the order *T*.

Construction of h:

It now suffices to show that there exists a history *h* such that $h|x = h_x$ and $h|y = h_y$. We note several facts about h_x and h_y:

    1. *T* is consistent with *precedes(h_x)*.

    2. h_y is serial, and activities appear in the order *T*.

3. *T* is consistent with *precedes(h_y)*.

4. *committed(h_y)* = *committed(h_x)*.

5. *aborted(h_y)* = $\varnothing$.

Now consider the following algorithm, which we will show "merges" h_x and h_y to get a history *h* satisfying our requirements:

```
s := Λ
sx := hx
sy := hy
while committed(sx) ≠ ∅ do
      let s1, s2, and a be such that sx = s1•<commit,x,a>•s2,
            and committed(s1) = ∅
      s := s • s1
      sx := s2
      if sy|a ≠ Λ then
            let s3 and s4 be such that sy = s3•<commit,y,a>•s4
            s := s • s3 • <commit,y,a>
            sy := s4
            end
      s := s • <commit,x,a>
      end
s := s • sx • sy
let h = s
```

It is clear that the algorithm terminates, since each iteration of the loop shortens *sx*. In addition, it is obvious that the final value of *h* contains the initial values of h_x and h_y as subsequences, as desired. It remains to be shown that the event sequence *h* is a history, that is, that it is well-formed. This we do below.

Well-formedness of h:

We repeat here the conditions on well-formed sequences from Chapter 2:

- An activity must wait until one invocation terminates before invoking another operation. More precisely, let *op-events(h)* be the subsequence of *h* consisting of all invocation and termination events; then *op-events(h|a)* must consist of an alternating sequence of invocation and termination events, beginning with an invocation event. In addition, an invocation event and the immediately succeeding termination event must involve the same object.

- No activity both commits and aborts in *h* (at the same or different objects); i.e., *commit(h)* $\cap$ *abort(h)* = $\varnothing$.

- An activity cannot commit if it is waiting for an invocation to terminate, and an activity cannot invoke any operations after it commits. More precisely, if *a* $\in$ *committed(h)*, then *h|a* consists of an alternating sequence of

invocation and termination events. ending in a termination event, followed by some number of commit events.

The second condition follows from the fact that $committed(h) = committed(h_x) \cup committed(h_y)$ and $aborted(h) = aborted(h_x) \cup aborted(h_y)$, and from the previously stated facts about h_x and h_y.

We argue the first condition as follows: If $b \notin committed(h)$ then $h|b = h_x|b$, and the first condition follows from well-formedness of h_x. Suppose $b \in committed(h)$, and suppose that the first condition does not hold for $h|b$; we will derive a contradiction. Note that if an invocation event is in the subsequence $s3$ moved from sy to s, then the corresponding termination event immediately follows it in $s3$, and hence in h.

Let $h1$ be the longest prefix of $h|b$ such that $h1$ is an alternating sequence of invocation and termination events, beginning with an invocation event. Since the first condition does not hold for $h|b$, h is longer than $h1$. We claim that $h1$ ends in an invocation event from h_x:

- $h1$ cannot end in an invocation event from h_y, since the corresponding termination event immediately follows it in h, implying that $h1$ is not the longest prefix of alternating invocation and termination events.

- $h1$ cannot end in a termination event from h_x, since the next event in $h|b$ would then have to be a termination event; this second termination event cannot come from h_y, since it immediately follows its corresponding invocation event, and it cannot come from h_x, since otherwise $h_x|b$ would contain two adjacent termination events.

- $h1$ cannot end in a termination event from h_y: If it did, the next event in $h|b$ would have to be a termination event, and would have to come from h_x; the next-to-last event in $h1$ would have to be the invocation event from h_y, and the event preceding it would have to be an invocation event from h_x, contradicting the assumption that $h1$ is an alternating sequence.

Since $h1$ ends in an invocation event from h_x, the next event in $h|b$ must be an invocation event from h_y.

Now consider the iteration of the main loop during which the events for b in h_y are moved in $s3$ from sy to s. s already contains $h1$ as a subsequence before $s3$ is appended to s, and $s2$ contains a termination event for b. Let a be the activity whose commit event is moved from sx to s during that iteration. Since $s2$ contains a termination event for b, a termination event for b follows a commit event for a in h_x, so $\langle a,b \rangle \in precedes(h_x)$. But events for b at y appear in $s3$, and so appear in h_y before the commit event for a; by the construction of h_y, $\langle b,a \rangle \in precedes(h_y)$. This is a contradiction, since T is consistent with $precedes(h_x)$ and $precedes(h_y)$.

Thus, the first condition holds for $h|b$.

Now consider the third condition. Suppose $a \in committed(h)$. By the first condition, $op\text{-}events(h|a)$ consists of an alternating sequence of invocation and termination events; since h_x and h_y are well-formed, it suffices to show that no events involving a except commit events occur in h after the first commit event for a. As shown above, when the commit event for a at y is moved to s, only commit events remain in sx for a. When the first commit event for a at x is moved to s, either no events remain for the activity in sy, or the events for the activity in sy are moved to s, on the same iteration of the main loop, before the commit event from sx is moved to s.

Thus, h is well-formed. Since $permanent(h_x)$ is not serializable in the order T, and $permanent(h_y)$ is serializable only in the order T, h is not atomic. This is the desired contradiction, showing that dynamic atomicity is optimal.

The locking protocols of [Bernstein et al. 81] and [Korth 81a] are suboptimal: while sufficient to ensure atomicity (given their assumptions about the underlying recovery mechanism), they achieve strictly less concurrency than permitted by dynamic atomicity. We will illustrate this point with detailed examples in Chapter 5.

4.2 Static Atomicity

Protocols characterized by dynamic atomicity determine a serialization order for activities based on the dynamics of the execution of activities. In contrast, timestamp protocols (see, e.g., [Reed 78]) determine a serialization order for activities *statically*, based on timestamps chosen when activities start. Our second local atomicity property characterizes the behavior of protocols which are static in this sense. We call this property *static atomicity*.

The remainder of this section is divided into three subsections. In the first, we define static atomicity and prove that it is a local atomicity property. Then, in the second, we show that static atomicity is optimal. Finally, in the third, we compare static and dynamic atomicity.

4.2.1 Definition of Static Atomicity

Static atomic objects ensure that activities are serializable in a pre-determined order. We model this behavior as follows: Let ACT be the set of all activities, and let TS be a fixed total order on ACT. Let h be a history containing initiation, invocation, termination, commit, and abort events. We say that h is *static atomic* if $permanent(h)$ is serializable in the order TS.

For example suppose x is a set object as before. Suppose that a and b are activities, and that

TS orders *b* before *a*. Then the following history *h* is atomic, but not static atomic:

<center>

<member(3),x,a>
<false,x,a>
<commit,x,a>
<insert(3),x,b>
<ok,x,b>
<commit,x,b>

</center>

Permanent(h) is a serial history, and is serializable in the order *a-b*. However, *permanent(h)* is not serializable in the order *b-a*, which is the order specified by TS.

As another example, the history

<center>

<insert(3),x,a>
<ok,x,a>
<commit,x,a>
<member(3),x,b>
<false,x,b>
<commit,x,b>

</center>

is static atomic: *Permanent(h)* is serializable in the order TS.

We say that an object is *static atomic* if every history permitted by the object's behavioral specification is static atomic. The following theorem verifies that static atomicity is a local atomicity property:

> **Theorem 4-4:** If every object in a system is static atomic, then every history in the system's behavior is atomic.
>
> **Proof:** Suppose that every object in a system is static atomic, and let *h* be a history of the system. By the definition of static atomicity, *permanent(h|x)* is serializable in the order TS. By Lemmas 3-3 and 3-2, *permanent(h)* is also serializable in the order TS, so *h* is atomic.

4.2.2 Optimality

Static atomicity, like dynamic atomicity, is optimal. The paragraphs below provide a precise proof of this result. We begin with some definitions, and then prove the theorem.

If *h* is a history, define the relation *commit-order(h)* on activities to contain all pairs <*a,b*> such that *a,b* $\in$ *committed(h)* and the first commit event for *a* in *h* occurs before the first commit event for *b* in *h*. Note that *commit-order(h)* is a partial order on activities, and that it totally orders *committed(h)*.

> **Theorem 4-5:** For systems using the events described in this section, static atomicity is optimal.
>
> **Proof:** The proof proceeds by contradiction, along the lines of the proof of Theorem 4-3. Suppose *P* is a local atomicity property that is strictly weaker than static atomicity. We will exhibit a system composed of objects satisfying *P*, and

show that there is a non-atomic history of that system. thus contradicting the claim that *P* is a local atomicity property.

Since *P* is more permissive than static atomicity. there must be a specification S_x of an object *x* such that S_x satisfies *P* but *x* is not static atomic. In particular, there must be at least one history h_x in *x.behavior* that is not static atomic; that is. such that *permanent*(h_x) is not serializable in the order TS. We will construct an object *y* whose specification contains a history h_y involving the committed activities in h_x. Now, consider a system containing *x*, *y*, and all the activities in h_x. As in the proof of Theorem 4-3, h_y will be chosen so that it is serializable only in the order TS, and there is a history *h* of this system such that $h|x = h_x$ and $h|y = h_y$. Since *permanent*(h_y) is only serializable in the order TS, and *permanent*(h_x) is not serializable in that order, it follows from Lemma 3-2 that *permanent*(*h*) is not serializable. Thus, *h* is not atomic.

Construction of y:

The serial specification of *y* is the same as in the proof of Theorem 4-3, and is described by the machine in Figure 4-1. Let *y.behavior* be the largest set such that *y* is static atomic. Notice that while the serial specification of *y* is the same as in the proof of Theorem 4-3, the behavioral specification is different. Since *P* is weaker than static atomicity, S_y satisfies *P*.

Choice of h_y:

Now, let *a1*, *a2*, ..., *an* be the activities in *committed*(h_x) in the order *commit-order*(*h*). Let *order*: {*a1*,...,*an*} → [1,2,...,n] map the *ai* to the integers between 1 and n such that *order*(*ai*)<*order*(*aj*) if and only if TS orders *ai* before *aj*. Let h_y be the following serial history in *y.behavior*:

<increment,y,a1>
<order(a1),y,a1>
<commit,y,a1>
<increment,y,a2>
<order(a2),y,a2>
<commit,y,a2>

...

<increment,y,an>
<order(an),y,an>
<commit,y,an>

Note that h_y is serializable only in the order TS.

Existence of h:

By the same arguments as in the proof of Theorem 4-3, there exists a history h such that $h|x = h_x$ and $h|y = h_y$. This gives the desired contradiction.

4.2.3 Discussion

Dynamic atomicity and static atomicity are different: each permits operations to be interleaved in ways that the other does not. This implies that optimal does not mean "best," but rather that nothing else is strictly better.

Which of these two local atomicity properties is better for a given application will depend on the patterns of operations invoked by activities. For example, dynamic atomicity works poorly for long read-only activities such as audits. If dynamic atomicity is implemented using a locking protocol, a read-only activity, once it has a lock on an object, will cause other activities that need conflicting locks to wait. Because of the need to wait for locks, long read-only activities can be quite prone to deadlock. Static atomicity, however, works reasonably well for long read-only activities. In the implementation proposed by Reed [Reed 78], read-only activities are never forced to abort (the analog of deadlock in a locking system), and are rarely delayed by other activities. On the other hand, static atomicity works poorly for update activities with old timestamps. For example, in the implementation proposed by Reed, if an activity attempts to write an object after another activity with a later timestamp has already read the object, the former activity must be aborted. Using dynamic atomicity, the writer might be delayed until the reader committed, but would then be able to proceed.

4.3 Hybrid Atomicity

Our final local atomicity property, which we call *hybrid atomicity*, characterizes the behavior of protocols that exhibit some of the characteristics of dynamic atomic protocols and some of the characteristics of static atomic protocols. Examples of such protocols include the mixed methods of [Bernstein & Goodman 81, Section 5.3.2] and the multi-version scheme proposed in [DuBourdieu 82] and formally analyzed in [Bernstein & Goodman 83].

Hybrid atomicity is based on two ideas. First, we partition activities into two classes: read-only activities, and update activities. Intuitively, a read-only activity is one that does not invoke any operations that change the state of an object. Formally, if the serial specification of an object is described by a state machine M, then an operation O on the object is *read-only* if $N_M(S,O) = S$ for all states S in which O is defined. An activity is *read-only* if every operation executed by the activity is read-only. All other activities are considered to be *update activities*

.

As in Reed's implementation of static atomicity, timestamps for read-only activities are chosen

when they begin execution. Timestamps for update activities, however, are chosen dynamically as they commit. The system ensures that the timestamp order on updates is consistent with the *precedes* order; objects can use this property to ensure that updates are serializable in timestamp order.

When a read-only activity with timestamp *t* invokes an operation on an object, the answer to its query is computed by including the effects of all operations executed by committed update activities with timestamps less than *t*. The system also ensures that update activities that commit later choose a timestamp greater than *t*, ensuring that the results returned to the read-only activity are not invalidated by an update that commits later.

The remainder of this section is divided into three subsections. In the first, we describe the additional events needed to define hybrid atomicity. In the second, we define hybrid atomicity and prove that it is a local atomicity property. We also claim that hybrid atomicity is optimal. Finally, in the third, we compare hybrid atomicity to static atomicity and dynamic atomicity.

4.3.1 Additional Events

To define hybrid atomicity precisely, we need to introduce some new events that describe the timestamps chosen by activities. In addition, we must partition the set of activities into two subsets: the updates (written *a*, *b*, and *c*), and the read-only activities (written *r* and *s*). Timestamps for updates are chosen when they commit, and an object learns of an update's timestamp when the update commits at the object. We write the event corresponding to the commitment of an update *a* at object *x* with timestamp *t* as ⟨commit(t),x,a⟩. Timestamps for read-only activities are chosen when they start, and an object learns of a read-only activity's timestamps when the activity invokes an operation on the object. We model this by including initiation events: Before invoking an operation on an object, a read-only activity must *initiate* at the object to let the object know its timestamp. We write the event corresponding to the initiation of activity *r* at object *x* with timestamp *t* as ⟨initiate(t).x,r⟩. We use the term *timestamp events* to denote the set of all commit events for updates and all initiation events for read-only activities.

We assume that timestamps are taken from some countable, totally ordered set; in our examples we will use natural numbers.

In addition to the well-formedness constraints on event sequences stated in Chapter 2, we have the following constraints:

- A read-only activity must initiate at an object before invoking any operations at the object; i.e., for every read-only activity *r* and every object *x*, $h|r|x$ begins with an initiation event.

· Any two timestamp events for distinct activities have distinct timestamps.

· Any two timestamp events for the same activity have the same timestamp.

For example, the following sequence is well-formed:

$$\langle insert(3),x,a\rangle$$
$$\langle ok,x,a\rangle$$
$$\langle commit(2),x,a\rangle$$
$$\langle initiate(1),x,r\rangle$$
$$\langle member(3),x,r\rangle$$
$$\langle false,x,r\rangle$$
$$\langle commit,x,r\rangle$$

The following sequence h, however, is not:

$$\langle insert(3),x,a\rangle$$
$$\langle ok,x,a\rangle$$
$$\langle commit(2),x,a\rangle$$
$$\langle member(3),x,b\rangle$$
$$\langle true,x,b\rangle$$
$$\langle commit(1),x,b\rangle$$
$$\langle initiate(2),x,r\rangle$$

since r and a use the same timestamp, violating the uniqueness property of timestamps.

We extend the definition of *opseq* to histories including timestamp events by having *opseq* throw away timestamp events in addition to completion events. Acceptability, serializability, and atomicity for histories are then defined as before in terms of *opseq*.

4.3.2 Definition of Hybrid Atomicity

Let h be a history. We say that h is *hybrid atomic* if *permanent*(h) is serializable in timestamp order.

For example, let x be a set object as before; then the following history h is atomic:

$$\langle insert(3),x,a\rangle$$
$$\langle ok,x,a\rangle$$
$$\langle insert(4),x,b\rangle$$
$$\langle ok,x,b\rangle$$
$$\langle commit(1),x,a\rangle$$
$$\langle commit(3),x,b\rangle$$
$$\langle initiate(2),x,r\rangle$$
$$\langle member(3),x,r\rangle$$
$$\langle true,x,r\rangle$$
$$\langle member(4),x,r\rangle$$
$$\langle true,x,r\rangle$$
$$\langle commit,x,r\rangle$$

since it is serializable in the order *a-b-r*. However, it is not hybrid atomic, for the following reason. *Permanent*(h) in timestamp order is the history

$$\langle insert(3),x,a \rangle$$
$$\langle ok,x,a \rangle$$
$$\langle commit(1),x,a \rangle$$
$$\langle initiate(2),x,r \rangle$$
$$\langle member(3),x,r \rangle$$
$$\langle true,x,r \rangle$$
$$\langle member(4),x,r \rangle$$
$$\langle true,x,r \rangle$$
$$\langle commit,x,r \rangle$$
$$\langle insert(4),x,b \rangle$$
$$\langle ok,x,b \rangle$$
$$\langle commit(3),x,b \rangle$$

which is not an acceptable serial history.

As another example, the history

$$\langle insert(3),x,a \rangle$$
$$\langle ok,x,a \rangle$$
$$\langle insert(4),x,b \rangle$$
$$\langle ok,x,b \rangle$$
$$\langle commit(1),x,a \rangle$$
$$\langle commit(3),x,b \rangle$$
$$\langle initiate(2),x,r \rangle$$
$$\langle member(3),x,r \rangle$$
$$\langle true,x,r \rangle$$
$$\langle member(4),x,r \rangle$$
$$\langle false,x,r \rangle$$
$$\langle commit,x,r \rangle$$

is hybrid atomic.

We say that an object is *hybrid atomic* if every history permitted by the object's behavioral specification is hybrid atomic. The following theorem verifies that hybrid atomicity is a local atomicity property:

Theorem 4-6: If every object in a system is hybrid atomic, then every history of the system is atomic.

The proof is identical to that for static atomicity.

Hybrid atomicity is optimal; the proof again is by contradiction.

4.3.3 Discussion

At first glance hybrid atomicity might not seem very different from static atomicity: both work by establishing a single global ordering on activities and ensuring that activities are serializable in that order. Static atomicity uses a pre-determined order, chosen before activities begin executing. Hybrid atomicity, in contrast, uses an order determined by the order in which updates commit. This difference is substantial: it raises a number of

interesting implementation issues, and results in some useful properties.

One simple way to implement hybrid atomicity is to use dynamic atomicity for update activities, generating timestamps for them so that the timestamp order on updates is consistent with the *precedes* order, and to compute the results of an operation invoked by a read-only activity with timestamp t by including the effects of all committed updates with timestamps less than t. (See, e.g., [DuBourdieu 82, Bernstein & Goodman 83] for a detailed description of this approach for simple objects with read and write operations.) This simple implementation, like implementations of static atomicity, permits read-only activities to run without interfering with update activities; by using dynamic atomicity for updates, however, the problems of static atomicity are avoided.

More complex implementations of hybrid atomicity can allow more concurrency among update activities than is permitted by dynamic atomicity. Consider, for example, a first-in-first-out queue object z with a serial specification as described by the machine in Figure 3-2. Hybrid atomicity allows activities to enqueue items in parallel, as is illustrated by the following hybrid atomic history:

$$\langle enqueue(1),z,a \rangle$$
$$\langle enqueue(2),z,b \rangle$$
$$\langle ok,z,a \rangle$$
$$\langle ok,z,b \rangle$$
$$\langle commit(3),z,b \rangle$$
$$\langle commit(5),z,a \rangle$$
$$\langle dequeue,z,c \rangle$$
$$\langle 2,z,c \rangle$$
$$\langle dequeue,z,c \rangle$$
$$\langle 1,z,c \rangle$$
$$\langle commit(17),z,c \rangle$$

Dynamic atomicity allows activities to enqueue items in parallel, but the items enqueued cannot be dequeued by another activity. Consider the following history h:

$$\langle enqueue(1),z,a \rangle$$
$$\langle enqueue(2),z,b \rangle$$
$$\langle ok,z,a \rangle$$
$$\langle ok,z,b \rangle$$
$$\langle commit,z,b \rangle$$
$$\langle commit,z,a \rangle$$
$$\langle dequeue,z,c \rangle$$
$$\langle ?,z,c \rangle$$
$$\langle commit,z,c \rangle$$

If c dequeues 1 in place of the "?," then h is not serializable in the order b-a-c; if c dequeues 2, then h is not serializable in the order a-b-c. Since *precedes*(h) contains only the pairs $\langle a,c \rangle$ and $\langle b,c \rangle$, h is not dynamic atomic, regardless of which item is dequeued by c. (In Chapter 5, we will discuss this problem with dynamic atomicity in more detail, and will explain how to

solve it.)

Hybrid atomicity can allow more concurrency than dynamic atomicity because the objects have more information, namely the timestamps assigned to update activities as they commit. This does not contradict our optimality results, but rather serves to emphasize their dependence on the information available to objects. (Cf. the work in [Kung & Papadimitriou 79] on the "optimality" of concurrency control protocols, and the dependence of "optimality" on the amount of information available to the protocols.)

Lamport [Lamport 76] suggested that atomicity is too strong a requirement because it permits too little concurrency, and presented the example of a banking system to support his claim. The system contains transfer activities (which move money between two accounts) and audit activities (which print out the current balances of all accounts). Lamport noted the performance problems of locking implementations, and suggested that the solution to these problems is to allow non-atomic executions. He defined a correctness property, namely that the view of the database seen by an audit must be consistent, and described an implementation that guarantees this property while permitting more concurrency than a locking implementation of atomicity. His correctness property does not ensure, however, that the view seen by an audit bears any relation to the actual state of the database. In addition, audits under his implementation still interfere with some updates. The implementations of hybrid atomicity discussed above solve the problem addressed by Lamport, namely the performance problems with read-only activities under dynamic atomicity. In contrast to Lamport's solution, hybrid atomicity ensures atomicity; this means that the view seen by an audit can be related to the updates performed by transfers and to the views seen by other audits. In addition, hybrid atomicity can be implemented so that audits do not interfere with any updates, and the techniques can be applied easily to other situations involving read-only activities.

Our primary motivation for developing hybrid atomicity was to solve the problems of dynamic atomicity with long read-only activities. In addition, hybrid atomicity permits more concurrency among update activities than does dynamic atomicity. For example, hybrid atomicity permits a FIFO queue that allows concurrent use of enq operations. Hybrid atomicity also appears useful for replication: Herlihy [Herlihy 84] has developed novel replication techniques, based on hybrid atomicity, for user-defined objects. Hybrid atomicity permits greater freedom in choosing quorums for operations than dynamic atomicity, providing the potential for increased availability.

4.4 Remarks

In this section we discuss a number of issues related to the material presented earlier. The section is divided into three subsections. In the first, we return to the issue of type-specific concurrency control discussed in Chapter 1. In the second, we discuss the meaning of the term "atomic type," and explain how the local atomicity properties discussed above serve to classify different kinds of atomic types. Finally, in the third, we discuss the structure of specifications of objects.

4.4.1 Type-specific Concurrency Control

Let us consider the example of a semiqueue object discussed earlier. The serial specification of a semiqueue object y was given in Figure 2-3. Consider the following history h:

$$\langle enq(1),y,a \rangle$$
$$\langle enq(2),y,b \rangle$$
$$\langle ok,y,b \rangle$$
$$\langle ok,y,a \rangle$$
$$\langle enq(3),y,a \rangle$$
$$\langle ok,y,a \rangle$$
$$\langle enq(4),y,b \rangle$$
$$\langle ok,y,b \rangle$$
$$\langle commit,y,a \rangle$$
$$\langle commit,y,b \rangle$$
$$\langle deq,y,c \rangle$$
$$\langle 3,y,c \rangle$$
$$\langle commit,y,c \rangle$$

Permanent(h) is the same as h, and *precedes*(h) contains the pairs $\langle a,c \rangle$ and $\langle b,c \rangle$. The reader may check that *permanent*(h) is serializable in the orders a-b-c and b-a-c, and thus that h is dynamic atomic.

Protocols based on a free interpretation of the operations (as used in [Papadimitriou 79]) cannot achieve this kind of concurrency: each operation is assumed to update the object's state, implying that activities cannot access an object concurrently. For example, two-phase locking using exclusive locks is shown in [Kung & Papadimitriou 79] to be optimal, given that no information is available about the semantics of operations. The example above, and others throughout this dissertation, illustrate how a specification of the acceptable serial behavior of objects can be used to achieve greater concurrency.

4.4.2 Atomic Types

As discussed in the introduction, a data type consists of a collection of objects with associated operations. We extend properties of objects to types as follows: We say that a type satisfies a property P if all of its objects satisfy P. For example, if a type's objects are all dynamic atomic, we say that the type is dynamic atomic.

Each of the three local atomicity properties discussed in this chapter defines a class of "atomic" types. The types within a single class are "compatible," in the sense that as long as all types in a system belong to a single class, activities are guaranteed to be atomic. Types in different classes are not necessarily compatible; for example, as illustrated in Chapter 1, atomicity is not guaranteed if dynamic atomic and static atomic objects are used together.

Indeed, a recent paper on concurrency control [Beeri et al. 83] claims to "verify" an implementation of an atomic type taken from [Weihl & Liskov 82]. The type happens to be dynamic atomic. However, all that is shown in [Beeri et al. 83] is that every history permitted by the implementation is atomic. As we illustrated in Chapter 1, if the implementation were placed in a system containing incompatible types, atomicity could be violated. Our results show that there are many different definitions for the term "atomic type," and that it is necessary to check that the different types in a system are all compatible.

4.4.3 Structure of Specifications

We separated the serial specification of an object from its behavioral specification to reflect the stages of our informal design process. As the following example illustrates, this separation also enhances the modularity of a system, by reducing the amount of information about an object needed by the programmer of an activity.

Consider the plight of the programmer of a single activity who is given a behavioral specification (and no separate serial specification) for each object and the assurance that every system history is atomic. The programmer would like to ignore the non-serial histories in each object's specification, since atomicity is supposed to ensure that all histories are "serializable." Now consider the following serial history in the specification of an object x which is informally documented as a set of integers:

$$\langle insert(3),x,a\rangle$$
$$\langle ok,x,a\rangle$$
$$\langle commit,x,a\rangle$$
$$\langle member(3),x,b\rangle$$
$$\langle false,x,b\rangle$$
$$\langle commit,x,b\rangle$$

Without knowing the local atomicity property satisfied by x, the programmer will find it difficult to place a reasonable interpretation on this history. If x is static atomic, and TS orders b before a, then the history makes sense for a set object, since it is serializable in the order TS. If, however, x is dynamic atomic, the history is rather odd: One would expect b to find 3 in the set after it has been inserted by a. By separating the serial specification from the behavioral specification, we avoid the need for the programmer of an activity to understand the details of the local atomicity property used by objects. Instead, the programmer needs to know only that every system history is atomic; he or she can then check that an activity preserves consistency using only the serial specification of each object.

.

.

.

Chapter Five

Locking

In this chapter we look at how two-phase locking protocols can be used to implement dynamic atomic objects. We begin in Section 5.1 by reviewing the locking protocols in the literature [Bernstein et al. 81, Korth 81a]. These protocols are limited in several ways, particularly in their inability to handle non-deterministic operations. Then, in Section 5.2, we describe a general locking protocol that avoids some, but not all, of the limitations of existing protocols. Our description covers both synchronization and recovery. Next, in Section 5.3, we verify that the protocol described in Section 5.2 ensures dynamic atomicity. Finally, in Section 5.4, we discuss a number of related issues.

5.1 Existing Protocols

The two-phase locking protocol of [Eswaren et al. 76] has been extended in [Bernstein et al. 81] and [Korth 81a] to use user-specified information about objects to increase concurrency. These extended protocols restrict invocations to be total and deterministic: If M is a machine describing a serial specification, S_M is the state domain of M, and TERM is the set of termination events, then for every invocation event i we can define a total function $perform_i$: $S_M \rightarrow S_M \times$ TERM, where $perform_i(s) = \langle s',r \rangle$ and $N_M(s,\langle i,r \rangle) = s'$. In other words, for every invocation i, and every state s of M, there exists exactly one termination event r such that $N_M(s,\langle i,r \rangle) \neq \perp$. In contrast, we permit partial invocations, for which there may be no termination event in some states, and non-deterministic invocations, for which there may be more than one termination event in some states. The serial specification of the semiqueue object in Figure 2-3 illustrates both: The deq invocation is partial (there are states in which no execution of deq is defined) and non-deterministic (there are states in which more than one execution of deq is defined).

Like the read-write locking protocol discussed in Chapter 1, these extended protocols partition the set of operations on an object into classes, and use a different lock mode for each class. Classes are relatively coarse-grained; for example, all operations with the same invocation event are in the same class.

Informally, we say that a lock mode for one class is *compatible* with a lock mode for another class if all operations in the first class "commute" with all operations in the second class. Two lock modes *conflict* if they are not compatible.

An execution of an operation must first acquire a lock in the mode defined for its class. A lock

can be acquired if no concurrent activity holds a conflicting lock. Locks are released when activities complete. If an operation is unable to acquire its lock, it waits until conflicting activities complete.

The proof that this extended protocol ensures dynamic atomicity is relatively simple: since operations executed by concurrent activities "commute," they can be reordered without affecting their results or the final state. By appropriately reordering operations, we can obtain a serial execution equivalent to any concurrent execution that obeys the locking rules. This proof depends, however, on the requirement that operations be total and deterministic. In the next two sections we define and verify a general locking protocol that permits partial and non-deterministic operations. The protocols of [Bernstein *et al.* 81] and [Korth 81a] are special cases of our protocol. In our remarks at the end of the chapter we discuss the differences between our protocol and the protocols of [Bernstein *et al.* 81, Korth 81a], and illustrate how their requirements enable the proof to be simplified.

5.2 A General Locking Protocol

Like the locking protocols of [Eswaren *et al.* 76, Bernstein *et al.* 81, Korth 81a], our protocol is based on the notion of "commutativity" of operations. We begin our description of the protocol by precisely defining commutativity.

5.2.1 Definition of Commutativity

It is convenient to define commutativity quite generally, so that it covers sequences of operations, not just individual operations. If M is a state machine, and T and U are two sequences of transitions of M, we say that T and U *commute* if, for every state s in which T and U are both defined, $T(U(s)) = U(T(s))$ and $T(U(s)) \neq \perp$.

The obvious definition of commutativity is simply that T and U commute if, for all states s, $T(U(s)) = U(T(s))$. Our definition differs in two respects from this simple definition: First, we only care about those states in which T and U are both defined. Second, we require that the state resulting from application of both T and U be defined. The simple definition permits $T(U(s)) = U(T(s)) = \perp$. As should be clear from the proof in the next section, the more complex definition is necessary for the protocol to work.

We illustrate our definition of commutativity with some examples taken from the serial specification of a set object x in Figure 2-2. If $T = x:\langle insert(i),ok \rangle$ and $U = x:\langle insert(j),ok \rangle$, then T and U commute: Both are defined in all states, and $T(U(s)) = U(T(s)) = s \cup \{i\} \cup \{j\}$. On the other hand, if $V = x:\langle delete(i),ok \rangle$, then T and V do not commute: $T(V(s)) = s \cup \{i\}$, while $V(T(s)) = s - \{i\}$. U and V commute, however, as long as $i \neq j$.

Suppose W = x:<member(i),true>. W and V do not commute: V and W are both defined only in states containing i. If s contains i, V(W(s)) = V(s) = s-{i}. However, W(V(s)) = W(s-{i}) = ⊥, so W and V do not commute. On the other hand, W and T commute: If s contains i, T(W(s)) = W(T(s)) = sU{i} = s.

5.2.2 The Protocol

Our description of our protocol is designed to emphasize the general strategy followed by the protocol, and to highlight the differences with other locking protocols. We do not address in this chapter the problem of designing an efficient implementation of the protocol for a particular object. An efficient implementation of this protocol for a "map" data type can be found in the appendix.

We assume that the serial specification of an object x is given by a state machine SERIAL. We let the behavioral specification of x consist of all dynamic atomic complete histories. An implementation of x is described by the state machine LOCK in Figure 5-1. The language of the machine LOCK consists of a set of event sequences involving x; in the next section we will show that LOCK is correct in the sense that every complete history in L(LOCK) is dynamic atomic.

The definition of LOCK uses some new notation: ACT is the set of all activities, INV is the set of all invocation events, and sequence(S), where S is a set, is the set of all sequences of elements of S. Also, the expression $m[x{\rightarrow}y]$, where m is a (possibly partial) function from X to Y, $x{\in}X$, and $y{\in}Y$, denotes that function identical to m except at x, which it maps to y. Recall that the symbol $\rightarrow_p$ is used to denote a partial function.

The machine LOCK works roughly as follows: Recovery is accomplished using intentions lists. When a termination event occurs for an activity, the operation (consisting of the termination event and its corresponding invocation event) is appended to an intentions list for the activity. When an activity commits, the operations on its intentions list are applied to the current value of x, producing a new current value. Thus, the current value of x reflects all changes made by committed activities.

Each activity has its own "view" of the current value of x, defined by applying the operations in the activity's intentions list to x's current value. An activity's view reflects all changes made by committed activities, plus changes made by the activity itself. A termination event is allowed to occur for an activity only if the operation (again consisting of the termination event and its corresponding invocation event) is defined in the activity's view. This is clearly necessary: If the activity then commits, and all other activities abort, the activity must be serializable after the other committed activities.

State Components

$current \in S_{SERIAL}$ initially I_{SERIAL}

$pending$: ACT $\rightarrow_p$ INV initially $\perp$

$intentions$: ACT $\rightarrow$ sequence(T_{SERIAL}) initially Λ

$committed \subseteq$ ACT initially $\varnothing$

$aborted \subseteq$ ACT initially $\varnothing$

Transitions: events involving x

$N_{LOCK}(s, \langle commit,x,a\rangle)$:

if $a \in s$.committed **then no change else changes**

s.current to N_{SERIAL}(s.current, s.intentions(a))

s.committed to s.committed $\cup$ {a}

$N_{LOCK}(s, \langle abort,x,a\rangle)$:

changes

s.pending to s.pending[a$\rightarrow\perp$]

s.aborted to s.aborted $\cup$ {a}

$N_{LOCK}(s, \langle i,x,a\rangle)$:

when i is an invocation event

changes s.pending to s.pending[a$\rightarrow$i]

$N_{LOCK}(s, \langle r,x,a\rangle)$:

when

r is a termination event

and s.pending(a) $\neq \perp$

and N_{SERIAL}(s.current, s.intentions(a)·x:$\langle$s.pending(a),r$\rangle$) $\neq \perp$

and x:$\langle$s.pending(a),r$\rangle$ commutes with every operation in

s.intentions(b), for every b in

ACT-s.committed-s.aborted-{a}

changes

s.pending to s.pending[a$\rightarrow\perp$]

s.intentions to s.intentions[a$\rightarrow$s.intentions(a)·x:$\langle$s.pending(a),r$\rangle$]

Figure 5-1:The machine LOCK.

In addition, a termination event is allowed to occur for an activity only if the operation commutes with all operations executed by concurrent activities. This is sufficient to ensure serializability in all orders consistent with the *precedes* order.

A more precise explanation of LOCK follows. The states of LOCK have five components. *Current* $\in S_{SERIAL}$ is the current serial state, derived from the initial serial state by performing the operations executed by committed activities. *Pending*: ACT $\rightarrow_p$ INV maps each activity to its current pending invocation, if any. *Intentions*: ACT $\rightarrow$ sequence(T_{SERIAL}) maps each activity to the sequence of operations it has executed. *Committed* $\subseteq$ ACT is the set of committed activities, and *aborted* $\subseteq$ ACT is the set of aborted activities. We denote the components of a state $s \in S_{LOCK}$ by *s.current*, *s.pending*, *s.intentions*, *s.committed*, and *s.aborted*.

When an activity commits for the first time, LOCK updates the current serial state by applying the activity's intentions list to the state. (Multiple commit events may occur for an activity at an object; we only want to apply the activity's changes to x's current value once.) When an activity aborts, the current serial state is not changed.

Each active activity has its own "current view" of the current serial state, defined by applying the activity's intentions list to the current serial state. Invocation events are recorded in *pending*. Termination events can occur for an activity whenever three conditions are satisfied:

1. The activity has a pending invocation.

2. The resulting operation (obtained by pairing the pending invocation with the termination event) is defined in the activity's current view.

3. The operation commutes with all operations executed by concurrent activities (those that have not yet committed or aborted).

When a termination event occurs, the corresponding operation is appended to the intentions list for the activity, and the record of the pending invocation is discarded.

Note that the "intentions list" for an activity is never discarded. A real implementation of this protocol would not keep intentions lists for aborted activities, or for committed activities once the intentions list had been applied to the current value of the object. The intentions lists for completed activities have no effect on the behavior of the machine; we keep them around solely for convenience in the proof of correctness in the next section. Similarly, if intentions lists for completed activities are discarded, there is no need in a real implementation to remember the sets of committed and aborted activities. Again, we keep them for convenience. These parts of the state of LOCK are not unlike the "ghost variables" of [Owicki & Gries 76].

5.3 Correctness Proof

We wish to prove the following theorem:

Theorem 5-1: Let SERIAL be a machine, and let x be an object with $x.serial$ = L(SERIAL), and with $x.behavior$ defined to be the set of all dynamic atomic complete histories involving x. Let LOCK be a machine as defined in the previous section. Then every complete history in L(LOCK) is in $x.behavior$.

We note that there may be histories in $x.behavior$ that are not in L(LOCK); i.e., dynamic atomicity permits more histories than can be achieved by this locking protocol. We will discuss the differences between dynamic atomicity and locking-based implementations of it in Section 5.4.2.

Our proof of the theorem consists of the verification of a collection of invariants relating the state of LOCK with the histories it accepts. We present our proof in three parts. First, in Section 5.3.1, we present some technical lemmas about commutativity. Then, in Section 5.3.2, we describe our main invariant. Finally, in Section 5.3.3, we present a series of lemmas that complete the proof.

5.3.1 Commutativity

We begin with some notation. In a diagram such as:

$$s_1 \searrow^{T} s_2$$
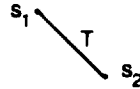
an arc leading downward indicates that $T(s_1) = s_2$, and hence that T is defined in s_1. If T and U commute, and we have the diagram:

$$T \diagup \diagdown U$$

then by the definition of commutativity we can complete the diagram to:

$$\begin{array}{c} T \diagup \diagdown U \\ U \diagdown \diagup T \end{array}$$
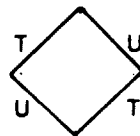
The first lemma provides an inductive technique for proving that two sequences commute:

Lemma 5-2: Let A_i and B_j, for $i,j \in \{0,1\}$, be sequences of transitions. If A_i commutes with B_j for all i and j, then $A_0 \cdot A_1$ commutes with $B_0 \cdot B_1$.

Proof: Let s be such that $A_0 \cdot A_1(s) \neq \perp$ and $B_0 \cdot B_1(s) \neq \perp$. We need to show that

$A_0 \cdot A_1(B_0 \cdot B_1(s)) = B_0 \cdot B_1(A_0 \cdot A_1(s))$ and that $A_0 \cdot A_1(B_0 \cdot B_1(s)) \neq \perp$. We have the following diagram:
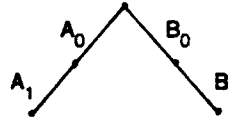
which by the hypotheses of the lemma can be completed to:
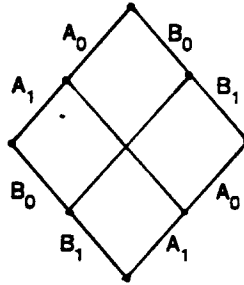
which shows the desired result.

The following corollary extends the lemma to sequences composed of more than two parts.

Corollary 5-3: Let A_i, $1 \leq i \leq m$, and B_j, $1 \leq j \leq n$, be sequences of transitions, and let $A = A_1 \cdot ... \cdot A_m$ and $B = B_1 \cdot ... \cdot B_n$. If A_i commutes with B_j for all i and j, then A commutes with B.

The following lemma addresses the situation when we have a collection of more than two sequences that commute pairwise.

Lemma 5-4: Let S be a state, and let T_i, $1 \leq i \leq n$, be sequences of transitions such that:

1. T_i is defined in S for all i.

2. T_i commutes with T_j for all i and j, $1 \leq i < j \leq n$.

Let $i1, i2, ..., in$ be a permutation of $1, 2, ..., n$. Then:

1. $T_1 \cdot T_2 \cdot ... \cdot T_n(S) \neq \perp$

2. $T_{i1} \cdot T_{i2} \cdot ... \cdot T_{in}(S) = T_1 \cdot T_2 \cdot ... \cdot T_n(S)$

Proof: The proof proceeds by induction on n. The case when $n = 1$ is trivial. The case when $n = 2$ follows directly from the definition of commutativity.

For the induction step, assume that the lemma holds for fewer than n sequences. Let j be such that $ij = 1$ (so $T_{ij} = T_1$). By the induction hypothesis, $T_2 \cdot T_3 \cdot ... \cdot T_n(S) \neq \perp$, and $T_{i1} \cdot ... \cdot T_{i(j-1)} \cdot T_{i(j+1)} \cdot ... \cdot T_{in}(S) = T_2 \cdot ... \cdot T_n(S)$.

Now let $S1 = T_{i1} \cdot ... \cdot T_{i(j-1)}(S)$. Since $T_{i1} \cdot ... \cdot T_{i(j-1)} \cdot T_{i(j+1)} \cdot ... \cdot T_{in}(S) \neq \perp$, it follows

that $S1 \neq \bot$ and $T_{i(j+1)} \cdot \ldots \cdot T_{in}$ is defined in S1.

By Corollary 5-3. T_1 commutes with $T_2 \cdot \ldots \cdot T_n$, $T_{i1} \cdot \ldots \cdot T_{i(j-1)}$, and $T_{i(j+1)} \cdot \ldots \cdot T_{in}$. Since T_1 is defined in S, and so is $T_2 \cdot \ldots \cdot T_n$, $T_2 \cdot \ldots \cdot T_n(T_1(S)) \neq \bot$. This proves the first half of the lemma.

Now, since T_1 is defined in S, and so is $T_{i1} \cdot \ldots \cdot T_{i(j-1)}$, it follows from the definition of commutativity that T_1 is defined in S1. Therefore $T_1(T_{i(j+1)} \cdot \ldots \cdot T_{in}(S1)) = T_{i(j+1)} \cdot \ldots \cdot T_{in}(T_1(S1))$.

The following equalities show the second half of the lemma:

$$T_1 \cdot T_2 \cdot \ldots \cdot T_n(S) = T_2 \cdot \ldots \cdot T_n(T_1(S))$$
$$= T_1(T_2 \cdot \ldots \cdot T_n(S))$$
$$= T_1(T_{i1} \cdot \ldots \cdot T_{i(j-1)} \cdot T_{i(j+1)} \cdot \ldots \cdot T_{in}(S))$$
$$= T_1(T_{i(j+1)} \cdot \ldots \cdot T_{in}(S1))$$
$$= T_{i(j+1)} \cdot \ldots \cdot T_{in}(T_1(S1))$$
$$= T_{i(j+1)} \cdot \ldots \cdot T_{in}(T_1(T_{i1} \cdot \ldots \cdot T_{i(j-1)}(S))$$
$$= T_{i(j+1)} \cdot \ldots \cdot T_{in}(T_{ij}(T_{i1} \cdot \ldots \cdot T_{i(j-1)}(S))$$
$$= T_{i1} \cdot T_{i2} \cdot \ldots \cdot T_{in}(S).$$

The second line follows since T_1 commutes with $T_2 \cdot \ldots \cdot T_n$. The remaining lines follow from equalities argued above.

5.3.2 On-line Dynamic Atomicity

In this section we define the main invariant to be proved about the histories in L(LOCK). We actually define two properties: *strong dynamic atomicity*, and *on-line dynamic atomicity*.

Let x be an object and M a machine, with $x.serial = L(M)$. We say that a history h is *strong dynamic atomic at x* if the following conditions are satisfied:

- $h|x$ is dynamic atomic.

- if *T1* and *T2* are total orders on activities consistent with *precedes(h|x)*, then
  $N_M(I_M, opseq(serial(permanent(h|x), T1))) =$
  $N_M(I_M, opseq(serial(permanent(h|x), T2)))$.

In other words, the final state of M resulting from executing the operations in *serial(permanent(h|x), T)* does not depend on *T*.

We say that a history h is *strong dynamic atomic* if it is strong dynamic atomic at all objects x. The following lemma follows easily from the definitions:

Lemma 5-5: If h is strong dynamic atomic, h is also dynamic atomic.

The following example illustrates how strong dynamic atomicity and dynamic atomicity differ. In Section 4.3.3 we illustrated a problem with dynamic atomicity, namely that concurrent activities can enqueue items on a FIFO queue, but that the items cannot be dequeued later. We repeat the example history here:

$$\langle enqueue(1),z,a \rangle$$
$$\langle enqueue(2),z,b \rangle$$
$$\langle ok,z,a \rangle$$
$$\langle ok,z,b \rangle$$
$$\langle commit,z,b \rangle$$
$$\langle commit,z,a \rangle$$
$$\langle dequeue,z,c \rangle$$
$$\langle ?,z,c \rangle$$
$$\langle commit,z,c \rangle$$

The problem is that the result returned by the dequeue operation depends on the order in which a and b are serialized. Consider the following prefix h of this history:

$$\langle enqueue(1),z,a \rangle$$
$$\langle enqueue(2),z,b \rangle$$
$$\langle ok,z,a \rangle$$
$$\langle ok,z,b \rangle$$
$$\langle commit,z,b \rangle$$
$$\langle commit,z,a \rangle$$

This prefix is not strong dynamic atomic: The state of the machine M resulting from serializing a before b is the sequence $1 \cdot 2$, while the state resulting from serializing b before a is the sequence $2 \cdot 1$.

This example is an instance of a general problem with dynamic atomicity. If an implementation allows a history that is not strong dynamic atomic, it may have difficulty responding to later invocations. There are two solutions to this problem. One is to use hybrid atomicity, as discussed in Section 4.3.3: Once activities commit, they can be totally ordered by their timestamps. The other is to strengthen dynamic atomicity, requiring strong dynamic atomicity instead. This latter solution results in a loss of concurrency; however, the lost concurrency does not seem useful, since it leads to situations where an activity invokes an operation and never gets a response. All implementations of dynamic atomicity that we have studied produce histories satisfying strong dynamic atomicity.

Our second property, *on-line dynamic atomicity*, seems fundamental to all pessimistic implementations of dynamic atomicity. We say that a history h is *on-line dynamic atomic at* x if, for every set C of activities such that $committed(h|x) \subseteq C \subseteq ACT$ - $aborted(h|x)$, the following conditions are satisfied:

- $h|x|C$ is serializable in every total order consistent with $precedes(h|x)$.

- if $T1$ and $T2$ are total orders consistent with $precedes(h|x)$, then $N_M(I_M, opseq(serial(h|x|C, T1))) = N_M(I_M, opseq(serial(h|x|C, T2)))$.

Pessimistic implementations have the property that once an activity executes an operation, it can commit at any time without violating dynamic atomicity. Consider a history h and all extensions of h obtained by appending commit events for some of the active activities in h; on-line dynamic atomicity simply requires that all these histories be strong dynamic atomic.

We say that a history h is *on-line dynamic atomic* if it is on-line dynamic atomic at all objects x.

The following lemma is immediate:

Lemma 5-6: If h is on-line dynamic atomic, h is also strong dynamic atomic.

5.3.3 Verification of LOCK

We will now present a series of lemmas describing properties of the machine LOCK and the histories accepted by it. The final lemma completes the proof of Theorem 5-1.

Recall that *commit-order(h)* is the partial order on activities containing all pairs $\langle a,b \rangle$ such that the first commit event for a in h occurs before the first commit event for b.

The first lemma describes a number of simple relationships between a history accepted by LOCK and the final state of LOCK after accepting the sequence.

Lemma 5-7: Suppose h is a history in the language of the machine LOCK. Let s = $h(I_{LOCK})$. Then:

1. $opseq(h|a) = s.intentions(a)$

2. $h|a$ ends in the invocation event $\langle i,x,a \rangle \Leftrightarrow s.pending(a) = i$

3. $a \in committed(h) \Rightarrow a \in s.committed$

4. $a \in aborted(h) \Rightarrow a \in s.aborted$

5. $a \neq b \land a,b \notin committed(h) \cup aborted(h) \Rightarrow s.intentions(a)$ commutes with $s.intentions(b)$

6. Let T be a total order consistent with *commit-order(h)*, and let *permanent* = $opseq(serial(h|committed(h), T))$. Then $s.current$ = $N_{SERIAL}(I_{SERIAL}, permanent)$.

Proof: The proof is by induction on the length of h. For illustration we will prove the fifth property, showing that the sequences of operations executed by two active activities commute.

The basis case, when $h = \Lambda$, is trivial.

For the induction step, suppose $h \neq \Lambda$, and assume that the fifth property holds for

all histories in L(LOCK) that are shorter than h. Then $h = k \cdot e$ for some history k in L(LOCK) and some event e. Since k is shorter than h, the fifth property holds for k. Note that if $a \notin committed(h) \cup aborted(h)$, then $a \notin committed(k) \cup aborted(k)$, and similarly for b. There are two cases, depending on the type of e.

If e is a commit, abort, or invocation event, then $s.intentions = k(I_{LOCK}).intentions$, and the result follows from the induction hypothesis.

If e is a termination event $\langle r,x,c \rangle$, $s.intentions$ differs from $k(I_{LOCK}).intentions$ only at c. If $a \neq c$ and $b \neq c$, then the result again follows from the induction hypothesis. Suppose without loss of generality that $a = c$. From the definition of LOCK it follows that $s.intentions(a) = k(I_{LOCK}).intentions(a) \cdot \langle k(I_{LOCK}).pending(a),r \rangle$, and by the induction hypothesis $k(I_{LOCK}).intentions(a)$ commutes with $s.intentions(b)$. By Corollary 5-3 and the precondition on e, $\langle k(I_{LOCK}).pending(a),r \rangle$ commutes with $s.intentions(b)$. The result then follows from Lemma 5-2.

The next lemma shows that the sequence of operations executed by an active activity is always defined in the current serial state.

Lemma 5-8: Suppose h is a history in the language of the machine LOCK. Let $s = h(I_{LOCK})$. Then:

> $a \notin committed(h) \cup aborted(h) \Rightarrow s.intentions(a)$ is defined in $s.current$

Proof: The proof proceeds by induction on the length of h. The basis case, when $h = \Lambda$, is trivial.

For the induction step, suppose $h \neq \Lambda$, and assume that the lemma holds for all histories in L(LOCK) that are shorter than h. Then $h = k \cdot e$ for some history k in L(LOCK) and some event e. Since k is shorter than h, the lemma holds for k. Note that if $a \notin committed(h) \cup aborted(h)$, then $a \notin committed(k) \cup aborted(k)$. There are three cases, depending on the type of e.

If e is an abort event or an invocation event, then $s.intentions = k(I_{LOCK}).intentions$ and $s.current = k(I_{LOCK}).current$. The result follows from the induction hypothesis.

If e is a termination event, the precondition for e ensures that the lemma holds for h if it holds for k.

If e is a commit event $\langle commit,x,b \rangle$, then $s.intentions = k(I_{LOCK}).intentions$. (Note that $b \neq a$, since $a \notin committed(h)$.) If $b \in committed(k)$ then $k(I_{LOCK}) = h(I_{LOCK})$, and the result follows from the induction hypothesis. Suppose $b \notin committed(k)$.

By the induction hypothesis, $k(I_{LOCK}).intentions(a)$ is defined in $k(I_{LOCK}).current$. Since $b\mathfrak{C}committed(k)$ and h is well-formed, $k(I_{LOCK}).intentions(b)$ is defined in $k(I_{LOCK}).current$. By Lemma 5-7, $k(I_{LOCK}).intentions(b)$ commutes with $k(I_{LOCK}).intentions(a)$. By the definition of commutativity, $k(I_{LOCK}).intentions(a)$ is defined in $N_{SERIAL}(k(I_{LOCK}).current, k(I_{LOCK}).intentions(b))$. which is $h(I_{LOCK}).current$.

An obvious corollary of the above two lemmas is that $h(I_{LOCK}).current$ is never $\perp$, implying that $h|committed(h)$ is serializable in any order consistent with $commit\text{-}order(h)$.

The penultimate lemma shows that the active activities can be serialized in any order starting in the current serial state, and the resulting final serial state does not depend on the order.

Lemma 5-9: Let h be a history in the language of the machine LOCK, and let $s = h(I_{LOCK})$. Suppose $A \subseteq ACT\text{-}aborted(h)\text{-}committed(h)$, and let T be any total order of activities. Then $N_{SERIAL}(s.current, opseq(serial(h|A,T)))$ is defined and does not depend on T.

Proof: Let $a1,...,an$ be the elements of A. By Lemmas 5-7 and 5-8, $opseq(h|ai)$ is defined in $s.current$ for all i. By Lemma 5-7, $opseq(h|ai)$ commutes with $opseq(h|aj)$ for all i and j, $1 \leq i < j \leq n$. The result follows from Lemma 5-4.

The final lemma proves that every history in L(LOCK) is on-line dynamic atomic. This completes our proof of Theorem 5-1, since by Lemma 5-6, every history (and hence every complete history) in L(LOCK) is therefore dynamic atomic.

Lemma 5-10: Suppose h is a history in L(LOCK). Then h is on-line dynamic atomic.

Proof: Since $h = h|x$, it suffices to show that h is on-line dynamic atomic at x. We repeat the conditions for on-line dynamic atomicity here: h is on-line dynamic atomic at x if, for every set C of activities such that $committed(h|x) \subseteq C \subseteq ACT\text{-}aborted(h|x)$, the following conditions are satisfied:

- $h|x|C$ is serializable in every total order consistent with $precedes(h|x)$.

- if T1 and T2 are total orders consistent with $precedes(h|x)$, then $N_M(I_M, opseq(serial(h|x|C, T1))) = N_M(I_M, opseq(serial(h|x|C, T2)))$.

The proof proceeds by induction on the length of h. The basis case, when $h = \Lambda$, is trivial.

For the induction step, suppose $h \neq \Lambda$, and assume that the lemma holds for all histories in L(LOCK) that are shorter than h. Then $h = k \cdot e$ for some history k in L(LOCK) and some event e. Since k is shorter than h, the lemma holds for k.

Let C be such that $committed(h) \subseteq C \subseteq ACT \cdot aborted(h)$, and let $T1$ and $T2$ be total orders consistent with $precedes(h)$.

First, note that $precedes(k) \subseteq precedes(h)$, $committed(k) \subseteq committed(h)$, and $aborted(k) \subseteq aborted(h)$. Thus, C, $T1$ and $T2$ satisfy the conditions of the definition of on-line dynamic atomicity for k. There are now two cases, depending on the type of e.

If e is a commit, abort, or invocation event, note that $opseq$ throws away pending invocation events (those without corresponding termination events) and completion events. Thus, $opseq(serial(h|C,Ti)) = opseq(serial(k|C,Ti))$. Since the lemma holds for k, it also holds for h.

Now suppose that e is a termination event $\langle r,x,a \rangle$. This is the difficult case. Note that if $a \notin C$ then $h|C = k|C$, and the result follows from the induction hypothesis. Assume that $a \in C$.

Let T be a total order of the activities in C such that the committed activities in h appear in $commit\text{-}order(h)$, a appears next, and the remaining activities appear in arbitrary order. Note that T is consistent with $precedes(h)$. We will first show that $h|C$ is serializable in the order T, and then show that for any order U consistent with $precedes(h)$, $N_{SERIAL}(I_{SERIAL}, opseq(serial(h|C, T))) = N_{SERIAL}(I_{SERIAL}, opseq(serial(h|C, U)))$. This suffices to prove the lemma.

Let $A = C\text{-}committed(h)$. The sequence $opseq(serial(h|C,T))$ can be written as $opseq(serial(h|committed(h),commit\text{-}order(h))) \cdot opseq(serial(h|A,T))$. By Lemma 5-7, $h(I_{LOCK}).current = N_{SERIAL}(I_{SERIAL}, opseq(serial(h|committed(h),commit\text{-}order(h))))$. By Lemma 5-9, $N_{SERIAL}(h(I_{LOCK}).current, opseq(serial(h|A,T))$ is defined. Thus, $h|C$ is serializable in the order T.

Now suppose U is consistent with $precedes(h)$. The sequence $opseq(serial(h|C,U))$ can be written as $S1 \cdot opseq(serial(h|a)) \cdot S2$, where $S1 = opseq(serial(h|C1,U))$, $S2 = opseq(serial(h|C2,U))$, and $committed(h) \subseteq C1$. Note that $Si = opseq(serial(k|Ci,U))$, since $a \notin Ci$.

Let V be a total order on $C1$ in which the elements of $committed(h)$ occur first in $commit\text{-}order(h)$, following by the remaining elements of $C1$ in arbitrary order. Let $NS1 = opseq(serial(k|C1,V))$. By the inductive hypothesis, $N_{SERIAL}(I_{SERIAL}, S1) = N_{SERIAL}(I_{SERIAL}, NS1)$.

Now let NS2 = $opseq(serial(k|C1\text{-}committed(h),V))$. The sequence NS1 can be written as $opseq(serial(h|committed(h),commit\text{-}order(h)))$ • NS2. The following equalities show the desired result:

$$N_{SERIAL}(I_{SERIAL} \cdot opseq(serial(h|C,U)))$$

$$= N_{SERIAL}(I_{SERIAL} \cdot S1 \cdot opseq(h|a) \cdot S2)$$

$$= N_{SERIAL}(I_{SERIAL} \cdot NS1 \cdot opseq(h|a) \cdot S2)$$

$$= N_{SERIAL}(I_{SERIAL} \cdot$$
$$opseq(serial(h|committed(h),commit\text{-}order(h))) \cdot NS2 \cdot opseq(h|a) \cdot S2)$$

$$= N_{SERIAL}(h(I_{LOCK}) \cdot current, NS2 \cdot opseq(h|a) \cdot S2)$$

$$= N_{SERIAL}(I_{SERIAL} \cdot opseq(serial(h|C,T))).$$

The last line follows from Lemma 5-9; the others follow from equalities argued above.

5.4 Remarks

This section consists of two parts. In the first we discuss the protocols in [Bernstein *et al.* 81, Korth 81a]. In the second we illustrate limitations of locking protocols, showing that there is potentially useful concurrency permitted by on-line dynamic atomicity that cannot be achieved by locking.

5.4.1 Existing Protocols Revisited

In this section we discuss the relationship between our general protocol and the locking protocols in [Bernstein *et al.* 81, Korth 81a]. We argue that these other protocols are special cases of our protocol.

The protocols in [Bernstein *et al.* 81, Korth 81a] are modeled using the scheduler model (see Figure 3-1). The scheduler determines whether an invocation can be executed; if so, the invocation is passed to the storage module. The storage module executes invocations in the order in which they are received, changing state and determining the results to be returned. (We note that the presentations in [Bernstein *et al.* 81, Korth 81a] do not consider recovery. The protocols are presented as characterizing the behavior of committed activities, and the implementation of recovery is not discussed.)

As discussed above, the protocols in [Bernstein *et al.* 81, Korth 81a] require invocations to be total and deterministic. This means that we can treat invocations as functions: If the serial specification of an object is described by a machine M (with state domain S_M), and TERM is the set of termination events, we can define a function $perform_i: S_M \rightarrow S_M \times$ TERM for each invocation event i. This function is defined by $perform_i(s) = \langle s',r \rangle$ such that $N_M(s,\langle i,r \rangle) = s'$.

The storage module executes an invocation i using $perform_i$. If the current state of the storage module is s, and $perform_i(s)$ = $\langle s',r \rangle$, then the new state of the storage module after executing i is s' and the result returned is the event r.

Commutativity is defined for *invocations*, rather than *operations*, in [Bernstein *et al.* 81, Korth 81a]: Two invocations i and j commute if $perform_i \circ perform_j$ = $perform_j \circ perform_i$, where "$\circ$" denotes composition of functions. (If invocations are restricted to be total and deterministic, this is equivalent to saying that $\langle i,q \rangle$ commutes with $\langle j,r \rangle$ for all termination events q and r.) The scheduler schedules an invocation for an activity if it commutes with all other invocations already executed by concurrent activities.

Recovery is not covered in [Bernstein *et al.* 81, Korth 81a]; rather, they rely on unstated assumptions about the operation of the storage module in handling abort events for activities. The description in the papers, and our explanation above, only cover committed activities. It is not clear how one would implement recovery using the scheduler model: the intent in [Bernstein *et al.* 81, Korth 81a] is clearly to use some sort of undo log, but this approach is complicated by the fact that not all operations have natural inverses (consider, for example, the insert operation on a set object). One could use intentions lists, but this requires a more complex notion of what it means for the storage module to "execute" an invocation once it has been scheduled. (Indeed, one would then have a description much closer to our machine LOCK, and much less like the scheduler model.)

From the above definition of the protocols in [Bernstein *et al.* 81, Korth 81a], it is easy to show that every history involving committed activities that is permitted by these protocols is also permitted by our protocol. The converse, however, is not true. One reason is that we permit objects, such as semiqueues, with partial non-deterministic invocations. The protocols in [Bernstein *et al.* 81, Korth 81a] do not.

The other reason is that our locking protocol, unlike the protocols in [Bernstein *et al.* 81, Korth 81a], permits information about the results of executing an invocation (i.e., the termination event) to be used in scheduling invocations. For example, consider a bank account object y, with a serial specification described by the machine in Figure 5-2. A bank account provides three operations: *deposit*, *withdraw*, and *balance*. *Deposit* adds a specified amount to the bank account. *Withdraw* withdraws a specified amount from the bank account if the current balance is adequate (in which case its result is "ok"); otherwise it leaves the account unchanged (with a result of "no"). *Balance* determines the current balance of the bank account. Note that the invocations on y are total and deterministic: For each invocation i and every state s of the machine M in Figure 5-2, there exists *exactly one* termination event r such that $N_M(s,\langle i,r \rangle) \neq \perp$. Further note that the invocation *withdraw*(3) does not commute with the invocation *balance* (by the definition of "commutativity" for invocations given above).

States: integers initially 0

Transitions: {y:<deposit(i),ok>, y:<withdraw(i),ok>,
                      y:<withdraw(i),no>, y:<balance,i>: i is an integer}

   N(s,y:<deposit(i),ok>):
         changes s to s + i   ·

   N(s,y:<withdraw(i),ok>):
         when s≥i
         changes s to s-i

   N(s,y:<withdraw(i),no>):
         when s<i

   N(s,y:<balance,i>):
         when s = i

Figure 5-2:Serial specification of a bank account object y.

Now consider the following history:

<withdraw(3),y,a>
<balance,y,b>
<no,y,a>
<0,y,b>
<commit,y,b>
<commit,y,a>

Since the invocation withdraw(3) does not commute with balance, the protocols in [Bernstein et al. 81, Korth 81a] cannot produce this history. However, this history can be produced by our locking protocol, since the operation y:<withdraw(3),no> commutes with the operation y:<balance,0>.

5.4.2 Limitations of Commutativity-based Protocols

All known two-phase locking protocols are based on some notion of commutativity: activities are allowed to execute operations concurrently only if the operations "commute." As discussed in the previous section, previously existing locking protocols are special cases of our protocol. In this section we illustrate how on-line dynamic atomicity permits more concurrency than can be achieved by any known locking protocol.

Consider is a bank account object y, with a serial specification as described in Figure 5-2. Two operations of the form y:<deposit(i),ok> and y:<deposit(j),ok> commute, since addition is commutative. However, two operations of the form y:<withdraw(i),ok> and y:<withdraw(j),ok> do not commute: If the current balance is greater than i and j but less than their sum, then neither sequence of both operations is defined. Similarly, y:<deposit(i),ok> does not commute

with y:⟨withdraw(j),no⟩: The latter operation is defined only in states s less than j; if s + i is greater than j, then y:⟨withdraw(j),no⟩ is not defined in the state resulting from executing y:⟨deposit(i),ok⟩ in state s. Also, y:⟨deposit(i),ok⟩ does not commute with y:⟨withdraw(j),ok⟩.

Consider the following history:

$$
\begin{array}{c}
\langle deposit(10),y,a\rangle \\
\langle ok,y,a\rangle \\
\langle commit,y,a\rangle \\
\langle withdraw(4),y,b\rangle \\
\langle withdraw(3),y,c\rangle \\
\langle ok,y,c\rangle \\
\langle ok,y,b\rangle \\
\langle commit,y,c\rangle \\
\langle commit,y,b\rangle
\end{array}
$$

This history is on-line dynamic atomic: It is serializable in the orders a-b-c and a-c-b, and the "final serial state" does not depend on the serialization order. However, since y:⟨withdraw(4),ok⟩ does not commute with y:⟨withdraw(3),ok⟩, and b and c execute these operations concurrently, this history is not permitted by locking protocols.

Similarly, the following sequence is on-line dynamic atomic but is not permitted by locking protocols:

$$
\begin{array}{c}
\langle deposit(1),y,a\rangle \\
\langle ok,y,a\rangle \\
\langle commit,y,a\rangle \\
\langle deposit(1),y,b\rangle \\
\langle ok,y,b\rangle \\
\langle withdraw(1),y,c\rangle \\
\langle ok,y,c\rangle \\
\langle commit,y,b\rangle \\
\langle commit,y,c\rangle
\end{array}
$$

On-line dynamic atomicity allows withdraw operations to be executed concurrently with deposit operations as long as the deposits are not needed to cover the withdrawals, or the withdrawals are too large to be affected by the deposits.

Locking implementations of on-line dynamic atomicity achieve less than maximal concurrency because they do not use two kinds of information available to them. First, locking protocols are conflict-based: synchronization is based on a pair-wise comparison of operations executed by concurrent activities. In contrast, on-line dynamic atomicity depends on the *sequences* of operations executed by activities. Second, locking protocols are history-independent: synchronization is independent of past history, in particular the operations executed by committed activities. In contrast, on-line dynamic atomicity depends on the entire history; witness the concurrent execution of withdraw operations when enough money has been deposited by committed activities to cover all of the withdrawals.

An example of a flight reservation list for an airline reservation database is presented in [Reuter 82]. This example is similar to our bank account, and also illustrates the limitations of locking.

A similar situation arises with semiqueue objects (see the serial specification in Figure 2-3). *Deq* operations do not always commute with other *deq* operations or with *enq* operations, but on-line dynamic atomicity permits *deq*'s to be executed with *enq*'s and other *deq*'s as long as there are enough enqueued items to cover all the *deq*'s.

Locking protocols are clearly useful for many applications, and can be implemented relatively easily. The examples above illustrate, however, that there may be applications for which locking protocols are inadequate. Later in this dissertation we will present implementations of the semiqueue and the bank account that achieve the kind of concurrency illustrated above. It remains to be seen whether the increased concurrency is worth the added complexity of the implementations.

Chapter Six

Linguistic Support in Argus

In the next two chapters, and in the appendix, we consider how atomic types can be implemented. Our purpose is two-fold: First, we will provide several detailed examples of implementations of highly concurrent atomic types. Second, we will evaluate alternative programming language constructs for implementing atomic types.

Our approach in these chapters is informal. We do not provide formal specifications of the types used in examples, nor do we formally verify the correctness of example implementations. Taking a formal approach to these issues would require developing a formal specification language and deductive system, and a formal semantics for the language in which implementations are expressed. Such work is well beyond the scope of this dissertation.

We limit our scope in these chapters in two ways. First, we focus on implementing dynamic atomic types. Second, we restrict our attention to pessimistic (as opposed to optimistic [Kung & Robinson 81]) implementations. In our remarks at the end of Chapter 7 we will discuss what can be concluded about other kinds of atomic types, and about optimistic implementations.

We use Argus [Liskov & Scheifler 82, Liskov et al. 83] as a vehicle for describing implementations. The mechanisms in Argus support a program structure in which no user code is executed when activities commit or abort. In this chapter we present the Argus approach. In Chapter 7 we will introduce extensions to Argus to support an alternative program structure, and discuss the relative merits of the two approaches.

In our formal analysis of atomicity we restricted our attention to single-level activities. In our study of implementations, we will permit activities to be nested, and will explore how implementations can support atomicity for nested activities. We will discuss the extensions needed to cope with nesting in Section 6.2.

Each of our example implementations ensures on-line dynamic atomicity (see Section 5.3.2). In addition, all of our examples are implemented using clusters [Liskov et al. 83], a data abstraction mechanism originally developed for CLU [Liskov et al. 81]. The view of types supported by clusters is slightly different from the class-like view (as in Simula [Dahl et al. 70] and Smalltalk [Robson 81]) taken in our formal model. These differences, however, are unimportant for our purposes; we will discuss them briefly later in this chapter.

The remainder of this chapter is organized as follows: In Section 6.1, we discuss the issues

involved in implementing an atomic type. Then, in Section 6.2, we discuss atomicity of nested activities, and illustrate how the definition of dynamic atomicity applies to nested activities. Next, in Section 6.3, we discuss the differences between the Argus view of types and the view taken in our formal model. Finally, in Section 6.4, we discuss the linguistic support provided by Argus, illustrating its use with a detailed example.

6.1 Issues

Like an implementation of a data type in a sequential language, an implementation of an atomic type must define a representation for objects of the type, and must provide implementations for each operation of the type in terms of that representation. However, the implementation of an atomic type must also ensure appropriate synchronization and recovery for activities using objects of the type. The necessary synchronization and recovery are defined by the type's specification, and depend on the local atomicity property satisfied by the type.

To provide synchronization and recovery for activities using objects of an atomic type, it is necessary to update the representation of objects as activities commit and abort. In Argus, the programmer relies on the system to update the representation. We call this an *implicit* approach. An alternative is for the programmer to supply code that is run when activities complete to update the representations of objects. We call this an *explicit* approach. In this chapter we focus on the implicit approach as supported by Argus. We will discuss the explicit approach and compare the two alternatives in Chapter 7.

In addition to providing appropriate synchronization and recovery for activities using objects of the type, an implementation of an atomic type must cope with internal concurrency and failures. An operation invoked by an activity is not executed instantaneously: It may fail after completing only some of the steps described by its implementation. Operations invoked by concurrent activities may also run concurrently. Steps must be taken by the implementation of the type to manage concurrency and failures of operations. (We avoided this issue in our analysis of LOCK in Chapter 5 by assuming that the transitions of LOCK were instantaneous.)

6.2 Nested Activities

Nested activities, or *subactivities*, are used for composing activities into larger activities. They are also a mechanism for limiting the scope of failures, and for introducing concurrency within an activity.

An activity may contain any number of subactivities, some of which may be performed sequentially, some concurrently. Atomicity for nested activities means that the internal

structure of an activity cannot be observed outside the activity; in other words, the activity as a whole, including all its nested activities, is serializable with respect to other activities at the same level. If the order in which nested activities appear to be executed does not matter, nested activities can be executed concurrently without any additional synchronization.

Nested activities can commit or abort independently, and a nested activity can abort without forcing its parent activity to abort. However, the commit of a nested activity is relative to its parent: Even if a nested activity commits, it will have no effect if its parent later aborts. A top-level activity has no parent; its effects cannot be undone once it has committed. (See [Reed 78, Moss 81, Liskov & Scheifler 82] for more detailed discussions of nested activities.)

As discussed in Chapter 1, nested activities form a natural tree structure, with each activity appearing as the parent of its subactivities. We can define the notions of child, sibling, ancestor, proper ancestor, descendant, and proper descendant in the usual way. As a technical device, we assume the existence of a single "root" activity which is the parent of all top-level activities.

The notion of the *least common ancestor* of two activities, and the related notion of *visibility*, are keys in extending dynamic atomicity to cover nested activities: If *a* and *b* are activities, then the *least common ancestor of a and b* is the ancestor of both *a* and *b* which is a descendant of all other ancestors common to *a* and *b*. For example, consider the following tree of activities:
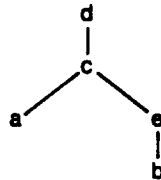
The ancestors common to *a* and *b* are *c* and *d*; *c* is the least common ancestor of *a* and *b*.

Informally, we say that an activity *b* is *visible to* an activity *a* if *b* has committed up to the level of the least common ancestor of *a* and *b*. For example, suppose that *a* and *b* are subactivities of the same activity. Then *b* is visible to *a* if and only if *b* has committed. As another example, consider the following tree of activities:
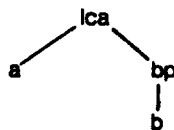
b is visible to *a* if and only if *b* and its parent, *bp*, have committed. If *b* is not visible to *a*, then either *b* (or one of its ancestors) has aborted, or *b* (or one of its ancestors) is still active. In

either case, a should not be permitted to "depend on" b's effects, since if b aborts a must then be aborted as well. (Note that the least common ancestor of two top-level activities is the "root" activity; thus, one top-level activity is visible to another if and only if it has committed.)

We extend our notion of the *precedes* order as follows: We say that b *precedes* a in an execution if b is visible to a when a, or a descendant of a, executes an operation. If b precedes a, then dynamic atomicity only requires that b be serializable before a. If, however, a and b are unrelated by the precedes order, then a and b must be serializable in either order. (On-line dynamic atomicity also requires that the final "serial state" not depend on the serialization order.)

For example, consider the tree of activities described above, and suppose activities execute the following steps on a semiqueue object y:

$$\langle enq(1),y,b \rangle$$
$$\langle ok,y,b \rangle$$
$$\langle commit,y,b \rangle$$
$$\langle deq,y,a \rangle$$
$$\langle 1,y,a \rangle$$
$$\langle commit,y,bp \rangle$$
$$\langle commit,y,a \rangle$$
$$\langle commit,y,lca \rangle$$

In this history b does not precede a, yet a is serializable only after b. Thus, this history is not dynamic atomic. On the other hand, the following history is dynamic atomic:

$$\langle enq(1),y,b \rangle$$
$$\langle ok,y,b \rangle$$
$$\langle commit,y,b \rangle$$
$$\langle deq,y,a \rangle$$
$$\langle commit,y,bp \rangle$$
$$\langle 1,y,a \rangle$$
$$\langle commit,y,a \rangle$$
$$\langle commit,y,lca \rangle$$

If an activity a is executing an operation on an object, then a's "view" of the object includes the effects of all activities that are visible to a. The results returned by the operation are allowed to "depend on" the activity's view, but must be independent of operations executed by concurrent (not visible) activities.

As discussed earlier, nested activities must be serializable at every level: each subactivity must appear indivisible to its siblings. The following example illustrates this requirement. Consider the following tree of activities:
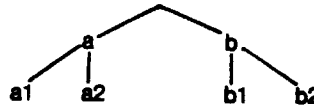
Suppose that x and y are both semiqueue objects, and consider the following history:

$$\langle enq(1),x,a1 \rangle$$
$$\langle ok,x,a1 \rangle$$
$$\langle enq(2),y,b1 \rangle$$
$$\langle ok,y,b1 \rangle$$
$$\langle commit,y,b1 \rangle$$
$$\langle commit,x,a1 \rangle$$
$$\langle deq,y,a2 \rangle$$
$$\langle 2,y,a2 \rangle$$
$$\langle deq,x,b2 \rangle$$
$$\langle 1,x,b2 \rangle$$
$$\langle commit,y,a2 \rangle$$
$$\langle commit,x,b2 \rangle$$
$$\langle commit,y,a \rangle$$
$$\langle commit,y,b \rangle$$
$$\langle commit,x,a \rangle$$
$$\langle commit,x,b \rangle$$

a1 and b1 enqueue items concurrently at x and y, respectively, and then a2 and b2 dequeue items at y and x, respectively. However, a2 is serializable only after b1, while b2 is serializable only after a1. This implies that a and b are not serializable, since in any equivalent serial execution, the activity that executes first will not see the item enqueued by the other activity. Indeed, the deq operation invoked by a2 should not return the item enqueued by b1 until b1 is visible to a2; i.e., until b1 and b have committed. At the point that a2's deq operation terminates in the above history, a2's view of the semiqueue contains no items. Similarly, the deq operation invoked by b2 should not return the item enqueued by a1 until a1 is visible to b2.

6.3 Types versus Objects

In our formal model we take a "class-like" [Dahl et al. 70, Robson 81] view of objects: We treat objects as independent entities, each with an associated collection of operations. Argus takes a different view, treating each data type as a "type manager," and associating operations with the type rather than with the objects. This view is called "cluster-like," after the name of the module used to implement such types.

The class-like view is more appropriate to distributed systems, in which the objects may be physically distributed. (For example, guardians in Argus, used to implement distributed

objects, are class-like. We use clusters for our implementations to avoid introducing issues such as the distinction between local and remote data.) The cluster-like view is useful for local data; for example, operations that involve more than one object of the type are more conveniently expressed. The class-like view, on the other hand, provides natural support for hierarchies of types. None of these distinctions, however, is important for our examples.

Operations that create new objects are not naturally associated with the objects themselves; rather, they are more naturally a part of the data type. A cluster-like view easily accommodates creation operations, since all operations are associated with a type. A class-like view can be made to accommodate creation operations by assuming that each type is itself an object, and by associating the creation operations with the type object.

We took the class-like approach in our formal model: We assumed that each type is itself an object, with operations to create new objects of the type. We model the serial behavior of creation operations in the following way: We assume that all objects exist for all time, and that when a system starts executing there is a sufficient supply in the system of each kind of object, in each possible initial state, to satisfy all creation operations that will ever be executed. (Formally, we can model this by assuming an infinite number of each kind of object.) A creation operation provided by a type simply selects any object of the type (in the appropriate initial state) that has not been previously selected.

The behavioral specification of a type must then describe how the type copes with concurrency and failures of activities creating objects. We require that an object never be returned by a creation operation more than once in an execution; since our sequential specification of creation operations is non-deterministic, this suffices to ensure that the type satisfies all three local atomicity properties discussed earlier.

6.4 Implementing Atomic Types in Argus

In this section we discuss how atomic types can be implemented in Argus. The section is divided into three parts. In the first, we discuss the linguistic support in Argus for implementing atomic types. In the second, we present an example implementation. Finally, in the third, we discuss the strengths and weaknesses of the approach taken in Argus.

6.4.1 Linguistic Support

The mechanisms in Argus have two important characteristics: The names of activities are not accessible to user code, and no user code runs when activities complete. The programmer must rely on the system to update the representations of objects when activities complete. The linguistic support in Argus consists of several built-in atomic types, statements that use

those types, and a mutual exclusion primitive.

The only processing done by the Argus system when an activity completes is to update locks and versions in the representation of each object of a built-in atomic type. Because of this, the programmer has to include some lower-level atomic objects (and ultimately at the lowest level some built-in atomic objects) in the representation of a user-defined atomic type. However, to implement types that permit highly concurrent use, the programmer must include non-atomic objects in addition to atomic objects. Some kind of synchronization and recovery is needed for these objects to cope with internal concurrency and failures; this is the purpose of the mutual exclusion primitive in Argus. The details of the Argus mechanisms are described below; a more complete description can be found in [Liskov *et al.* 83].

6.4.1.1 The Type Generator Atomic_variant

Argus provides several built-in atomic types and type generators. Of particular interest to us in our examples is the built-in type generator **atomic_variant**. The serial specification of atomic variants is essentially that of variants in CLU [Liskov *et al.* 81]: A variant type specification consists of a list of tags and associated types. The state of a variant object consists of a tag and a value; if the current tag of a variant is t, then the type of the current value of the variant is the type associated with t. For each tag t in the type specification, there are four operations: *make_t*, *change_t*, *is_t*, and *value_t*. *Make_t* takes one argument of the type associated with the tag t, and returns a new variant object whose tag is t and whose value is the argument of the operation. *Change_t* takes two arguments, a variant and an object of the type associated with t, and changes the state of the variant so that its tag is t and its value is the second argument of the operation. *Is_t* takes one variant argument, and returns **true** if and only if the tag of the variant is t. Finally, *value_t* takes one variant argument; if the tag of the variant is t then it returns the current value, and otherwise it signals *wrong_tag*. When only these operations are used, atomic variants are dynamic atomic.

Atomic variants are used in two ways: in conjunction with other atomic objects, to make activities atomic; and in conjunction with non-atomic objects, as part of the representation of a user-defined atomic object. In the latter case, it may be possible for an activity to gain access to an atomic variant created by another activity that has aborted or is still active. For example, one activity might create an atomic variant and insert in a (non-atomic) array; a concurrent activity with access to the array could then access the newly created atomic variant. Thus, the *make_t* operations, which create new atomic variant objects, require special consideration: We must define what happens to an atomic variant when its creator aborts, and whether concurrent activities can use an atomic variant before the object's creator has completed.

Argus defines the *make_t* operation to create a new atomic variant object whose tag is t and

whose value is the argument to the operation; this state is the object's "base" state, and the object will continue to exist in this state even if the creating activity aborts. (An alternative is to have the object "disappear" when its creator aborts; our experience indicates that this leads to awkward and complex programs.)

Concurrent use of a newly created atomic variant is limited as follows: The operations on atomic variants are classified as readers and writers. Synchronization of activities using an atomic variant object is done with read and write locks. The usual locking rules apply: Any number of activities can hold read locks simultaneously, but if one activity holds a write lock then no other activity can hold a read lock or a write lock. *Make_t*, *is_t*, and *value_t* are all readers, and *change_t* is a writer; readers acquire read locks when executed, and writers acquire write locks. By having the activity that creates an atomic variant retain a read lock, we ensure that the activity will not observe concurrent use of the object by other activities.

6.4.1.2 The Tagtest Statement

In an implementation of a user-defined atomic type, it is convenient for an activity to be able to test whether it would have to wait if it were to invoke a particular operation on an atomic variant. Argus provides the **tagtest** statement as structured support for testing and setting locks on atomic variants. The use of the **tagtest** statement can violate atomicity, since it permits an activity to observe the presence of concurrent activities. However, it appears to be necessary for implementing user-defined atomic types using an implicit program structure.

A **tagtest** statement has the following form:[1]

> **tagtest** expression
> > atag_arm { atag_arm }
> > [**others**: body]
> > **end**

where

> atag_arm :: = tag_kind name, ... [(idn: type_spec)] : body

> tag_kind :: = **tag**
> > **| wtag**

The expression must evaluate to an atomic variant object. If a read lock could be obtained on the atomic variant object by the activity executing the statement, then the tag of the object is matched against the names on the atag_arms; if a matching name is found, then the tag_kind on the arm is considered.

---

[1]We use an extended BNF for syntactic descriptions, with the following conventions: **|** is used to separate alternatives; " [*a*] " denotes an optional *a*; "{ *a* }" denotes a sequence of zero or more *a*'s; and "*a*, ..." denotes a list of one or more *a*'s separated by commas.

If the tag_kind is **tag**, a read lock is obtained on the object and the match is complete. If the tag_kind is **wtag** and the activity can obtain a write lock on the object, then a write lock is obtained and the match is complete. In all other cases the match is incomplete.

If a complete match is not found, or the activity could not obtain a read lock, then the body in the **others** arm, if present, is executed; if there is no **others** arm, the **tagtest** statement terminates.

When a complete match is found, if a declaration (*idn*: *type_spec*) appears on the matching arm, the value component of the object is assigned to the local variable *idn*. The body on the matching arm is then executed; *idn*, if declared, is defined only in that body. The entire matching process, including testing and acquisition of locks, is indivisible.

6.4.1.3 Mutual Exclusion

Argus provides the built-in type generator **mutex** and the **seize** statement to enable implementations to cope with concurrency among executions of operations. **Mutex** and **seize** can be used to ensure mutual exclusion among regions of code executed by concurrent operations; thus, for example, implementations can prevent interference among concurrently executing operations by forcing them to run serially.

Mutex objects are mutable containers for information. The type generator **mutex** has a single parameter, which is the type of the contained object. Mutex types provide operations to create and decompose mutex objects. The *create* operation takes a single argument of the parameter type and creates a new mutex object containing the argument object. The *get_value* operation extracts and returns the contained object from its mutex argument; following the conventions of Argus, the expression "mutex[t]$get_value(m)" is usually written "m.value".

Mutexes are used primarily to provide mutual exclusion on non-atomic shared data. Argus provides the **seize** statement, which allows a sequence of statements to be executed by an activity while the activity is in exclusive possession of a mutex object. The **seize** statement has the following form:

> **seize** expression **do** body **end**

The expression must evaluate to a mutex object. After evaluating the expression, the executing activity attempts to gain *possession* of the resulting mutex object. Only one activity may have possession of a mutex object at one time; thus, an activity may be forced to wait when it attempts to gain possession. Once the activity gains possession, the body of the **seize** statement is executed. Termination of the body causes possession of the mutex object to be released. If several processes are waiting for possession of the same mutex object, possession will be awarded *fairly*, in the sense that as long as no process retains possession

forever, every waiting process will eventually gain possession.

An implementation of an operation often has a precondition that must be true before the operation can be executed. Sometimes it is necessary to gain possession of mutex objects simply to test the precondition; if the precondition is false, the operation must wait. It is important to release possession of mutex objects while waiting, particularly if some other operation that requires possession of the mutex objects must be executed for the precondition to become true. Argus provides the **pause** statement for this purpose. It may be executed only inside the body of a **seize** statement. When a process executes **pause**, the mutex object seized by the closest enclosing **seize** statement is released, and the process is blocked for a system-determined time. When the process is unblocked, it regains possession of the mutex object released by the **pause** statement, waiting if necessary, and then continues execution with the statement following the **pause**.

Activities in Argus never fail while in possession of a mutex object unless the containing guardian crashes. When such a crash occurs, the states of the objects in the guardian are restored from stable storage. A discussion of the interactions between implementations of atomic types and stable storage in Argus can be found in [Weihl & Liskov 82]. In this chapter and the next we will assume that an activity in possession of a mutex object executes the body of the **seize** statement correctly, and does not abort until after it has released possession of the mutex.

6.4.2 Implementation of the Semiqueue Type

Our first example is an implementation of the *semiqueue* type. The serial specification of a semiqueue object was presented in Figure 2-3; the corresponding informal specification of the semiqueue type is presented in Figure 6-1. Semiqueues are similar to queues, except that enqueued items are not necessarily dequeued in first-in-first-out order. Instead, the *deq* operation makes a non-deterministic choice of an item to remove and return.

As noted in Sections 4.4.1 and 5.4.2, semiqueues place few constraints on concurrency. Two *enq* operations commute with each other, as do an *enq* and a *deq* operation or two *deq* operations as long as they involve different items. Thus, many different activities can *enq* concurrently, or *deq* concurrently. Furthermore, one activity can *enq* while another *deq*'s, provided only that the *deq* not return the newly *enq*'d item. Dynamic atomicity permits some additional concurrency as well: one activity can *enq* an item while another *deq*'s the same item as long as at least one committed activity has enqueued the same item.

An implementation of the semiqueue data type appears in Figure 6-3. The plan of this implementation is to keep the enqueued items in a regular (non-atomic) array. This array can be used by concurrent activities, but it is enclosed in a mutex object to control internal

data type semiqueue[item: type] is create, enq, deq

% A semiqueue is like a bag (or multiset) of items; *semiqueue[item]* is dynamic atomic if
% *item* is dynamic atomic.

create = **proc** () **returns** (semiqueue)
% Returns a new, empty semiqueue.

enq = **proc** (q: semiqueue, i: item)
% Adds *i* to *q*.

deq = **proc** (q: semiqueue) **returns** (item)
% If *q* is non-empty, chooses some element of *q*, removes it from *q*, and returns it.

Figure 6-1:Informal specification of the data type semiqueue.

concurrency. All modification and reading of the array occurs inside a **seize** statement on this containing mutex object. An informal specification of arrays appears in Figure 6-2.

To determine the status of each item in the array, we associate with each item an atomic object that tells the status of activities that inserted or deleted that item. For this purpose we use the built-in atomic type **atomic_variant** (described in the previous section).

The semiqueue operations are implemented as follows: The *create* operation simply creates a new empty array and places it inside a new mutex object. The *enq* operation associates a new atomic variant object with the argument item; this atomic variant will have tag "enqueued" if the calling activity commits later, and tag "dequeued" if it aborts. Then *enq* seizes the mutex and adds the new atomic variant to the contained array.

The *deq* operation seizes the mutex and then searches the array for an item it can dequeue: If an atomic variant has tag "enqueued" and the activity that called *deq* can get a write lock on it, the contained item is selected and returned after changing the variant's tag to "dequeued." If no suitable atomic variant is found, **pause** is executed (releasing the mutex) and later the search is retried.

Proper synchronization of activities using a semiqueue is achieved by using the *qitems* in the buffer. An *enq* operation need not wait for any other activity to complete. It simply creates a new *qitem* and adds it to the array. Of course, it may have to wait for another operation to release the mutex object before adding the *qitem* to the array, but this delay should be relatively short. A *deq* must wait until some activity that executed an *enq* operation commits relative to the activity that invoked *deq*; thus it searches for a *qitem* with tag "enqueued" that it can write.

The *qitems* are also used to achieve proper recovery for activities using a semiqueue. Since

data type array[t: type] is new, size, empty, fetch, store, addh, addl, remh, reml, elements

% Arrays are extensible: they can grow and shrink. They may be viewed as partial mappings
% from integers to objects of type t, with the restriction that an array is always defined on a
% connected interval of integers. The state of an array a can be modeled with two compo-
% nents: an integer $a.low$, called the low bound; and a sequence $a.elts$ of objects of type t,
% called the elements. An array a is empty if $a.elts$ is the empty sequence, and is otherwise
% defined on integers from $a.low$ to $a.low + |a.elts|-1$, where $|a.elts|$ is the length of $a.elts$. We
% say that an integer i is *in bounds* for an array a if a is defined on i. If s is a sequence, we
% use $s(i)$ to denote the ith element of s (where $s(1)$ is the first element of s).

new = **proc** () **returns** (array[t])
% Returns a new empty array with low bound 1.

size = **proc** (a: array[t]) **returns** (int)
% Returns $|a.elts|$.

empty = **proc** (a: array[t]) **returns** (bool)
% Returns **true** if and only if $|a.elts| = 0$.

fetch = **proc** (a: array[t], i: int) **returns** (t) **signals** (bounds)
% If a is defined on i, then returns $a.elts(i-a.low + 1)$; otherwise signals *bounds*.

store = **proc** (a: array[t], i: int, x: t) **signals** (bounds)
% If a is defined on i, then changes $a.elts(i-a.low + 1)$ to x; otherwise signals *bounds*.

addh = **proc** (a: array[t], x: t)
% Changes $a.elts$ to $a.elts\|x$.

addl = **proc** (a: array[t], x: t)
% Changes $a.elts$ to $x\|a.elts$, and changes $a.low$ to $a.low$-1. Decrementing
% $a.low$ keeps the indexes of the previously existing elements the same.

remh = **proc** (a: array[t]) **returns** (t) **signals** (bounds)
% Changes $a.elts$ by removing the last element in the sequence; signals *bounds* if $|a.elts| = 0$.

reml = **proc** (a: array[t]) **returns** (t) **signals** (bounds)
% Changes $a.elts$ by removing the last element in the sequence, and increments $a.low$; signals
% *bounds* if $|a.elts| = 0$. Incrementing $a.low$ keeps the indexes of the previously existing
% elements the same.

elements = **iter** (a: array[t]) **yields** (t)
% Yields the elements of $a.elts$ in order from the first to the last.

Figure 6-2: Informal specification of the data type **array**.

Figure 6-3:Implicit implementation of the data type semiqueue.

```
semiqueue = cluster[item: type] is create, enq, deq

qitem = atomic_variant[enqueued: item,
                         dequeued: null]
buffer = array[qitem]
rep = mutex[buffer]

create = proc () returns (cvt)
    return(rep$create(buffer$new()))
    end create

enq = proc (q: cvt, i: item)
    qi: qitem : = qitem$make_dequeued(nil) % dequeued if activity aborts
    qitem$change_enqueued(qi, i)           % enqueued if activity commits
    seize q do
        b: buffer : = q.value
        buffer$addh(b, qi)                 % add new qitem to buffer
        end
    end enq

deq = proc (q: cvt) returns (item)
    cleanup(q)        % cleanup should be called less frequently
    seize q do
        b: buffer : = q.value
        while true do
            for qi: qitem in buffer$elements(b) do
                tagtest qi % see if item can be dequeued by this activity
                    wtag enqueued (i: item):  qitem$change_dequeued(qi,nil)
                                                return(i)
                    end
                end
            pause
            end
        end
    end deq
```

94

Figure 6-3: (continued)

```
cleanup = proc (q: rep)
    enter topaction   % start an independent activity
        seize q do
            b: buffer : = q.value
            for qi: qitem in buffer$elements(b) do
                % remove only qitems in the dequeued state
                tagtest qi
                    tag dequeued: buffer$reml(b)
                    others: return
                    end
                end
            end
        end
    end cleanup

end semiqueue
```

the array in the mutex is not atomic, changes to the array made by activities that abort later are not undone. This means that a *deq* operation cannot simply remove a *qitem* from the array, since this change could not be undone if the calling activity later aborted. Instead, a *deq* operation changes the state of a *qitem*; the atomicity of *qitems* ensures proper recovery for this modification. If the calling activity later commits to the top level, the *qitem* will have tag "dequeued" permanently. Such *qitems*, which are also generated by *enq* operations called by activities that later abort, have no effect on later operations. Leaving them in the array wastes storage, so the internal procedure *cleanup*, called by *deq*, removes them from the low end of the array. (Of course, a more realistic implementation would call *cleanup* only occasionally.)

Note that *cleanup* cannot run in the calling activity: If the calling activity had previously executed a *deq* operation, that *deq* is visible to a later operation executed by the same activity. Instead, *cleanup* runs as an independent activity. This activity will only be able to lock *qitems* that are not being used by any active activities; thus it will not remove any *qitems* that could affect later operations.

6.4.3 Remarks

The implementation of the semiqueue type in the previous section illustrates the general strategy used to implement a user-defined atomic type in Argus. The representation of a user-defined atomic type typically consists of a mutex object containing a non-atomic collection (e.g., an array) of atomic objects (typically atomic variants). Greater concurrency among activities using the type is achieved by introducing atomic objects only at the lowest level of the representation.

The implementation in the previous section also illustrates a number of limitations of the expressive power of the implicit approach supported by Argus.

First, the implementation of *deq* is relatively inefficient, since in the worst case it takes time proportional to the size of the representation of the semiqueue. There is no obvious way to improve the efficiency of this implementation: The activity that executes an operation is implicit, so there is no way to structure the representation of an object based on the activities that enqueued or dequeued an item.

Second, scheduling of *deq* operations is accomplished using busy-waiting. The system has very little information on which to base scheduling decisions, implying that an activity is likely to be awakened when it is unable to complete the operation, and also that an activity may be unlikely to be awakened very soon after the precondition for the operation becomes true.

Finally, the programmer has no control over when the representation of an object gets

96

updated by the system as activities commit and abort. In Argus, the system updates built-in atomic objects automatically when activities complete, and does so at arbitrary times. The following example illustrates the problems that can result: Suppose that we want to implement the semiqueue type with the following additional constraint: If there is only one dequeuing activity at a time, and dequeuing activities do not abort, then items enqueued by a single activity should be dequeued in the order in which they were enqueued. This constraint is not satisfied by the implementation presented above. Suppose that activity A has enqueued two items, X and Y, in that order, and that activity B starts to execute a *deq* operation. If A commits after B has examined the first *qitem* (containing X) in the representation of the semiqueue and before B has examined the second, the *deq* operation will return Y.

It seems impossible to modify the implementation of semiqueue presented above to satisfy this additional constraint, given the semantics of Argus: Suppose *deq* has found an item that can be dequeued. There is no way to tell whether some other item was enqueued by the same activity. If we impose the additional restriction on the system that commits and aborts appear to be instantaneous (so if an activity has committed at one **atomic_variant** then it has committed at any others that it touched), then we can modify *deq* to search backwards through the representation and to return the last available item that it finds. (Or to search forwards until it finds one available item, and then to search backwards from there.) The resulting implementation satisfies the additional constraint on semiqueues.

This example illustrates that the programmer does not have complete control over all events that affect the representation of an object. Commit and abort events involving lower-level objects are controlled by the system, and can occur asynchronously. This asynchrony is visible to the programmer, and can affect the correctness of an implementation.

In the next chapter we will present an alternative approach that avoids the problems discussed above.

Chapter Seven

Support for an Explicit Approach

In this chapter we discuss an explicit approach for implementing atomic types, in which the programmer supplies code that is run when activities complete to update the representations of objects. We begin in Section 7.1 by describing extensions to Argus to support the explicit approach. Then, in Section 7.2, we present an example implementation to illustrate the approach. Finally, in Section 7.3, we compare the implicit and explicit approaches.

7.1 Linguistic Support

We present the language constructs as additions to Argus. We do not intend this to be a complete language proposal; rather, it is a vehicle for presenting examples using an explicit approach. The examples will serve both to illustrate how implementations of atomic types can be constructed using an explicit approach, and as a basis for comparing the explicit and implicit approaches.

We extend Argus in three ways. First, we add a new built-in data type, **aid**, to represent names of activities. Second, we extend the existing module for implementing abstract data types (the **cluster**) to provide the implementation of an operation with access to the name of the activity that invoked it, and to permit easy identification to the system of the code to be run when activities commit and abort. Third, we add a queuing/signalling mechanism designed to support efficient synchronization of activities.

An informal specification of the type **aid** appears in Figure 7-1. Note that no operations are provided to create new aids. While we will allow an implementation of an atomic type explicit access to the aid of an activity that invokes one of the type's operations, we follow Argus in implicitly associating aids with processes. Thus, the system automatically creates new aids whenever an existing activity executes the **enter** statement to create subactivities or nested top-level activities, and associates the new aids with the corresponding processes. Also note that the set of operations provided by the type **aid** is not complete; we have included only those that we need for our examples, and expect that others would be needed for general use.

We extend clusters in two ways. First, a routine in a cluster that implements an operation of the defined type can have two interface specifications. The *external* specification corresponds to the interface specification of the operation in the type's specification. The *internal* specification differs from the external specification in that it has an additional implicit

data type aid is parent, ancestors, proper_ancestors, top, equal

parent = **proc** (a: **aid**) **returns** (**aid**) **signals** (**top**)
% Returns a's parent; signals *top* if a has no parent.

ancestors = **iter** (a: **aid**) **yields** (**aid**)
% Yields the ancestors of a, including a itself, in root-to-leaf order.

proper_ancestors = **iter** (a: **aid**) **yields** (**aid**)
% Yields the proper ancestors of a (i.e., not including a itself) in root-to-leaf order.

top = **proc** (a: **aid**) **returns** (**bool**)
% Returns **true** if a is a top-level activity; otherwise returns false.

equal = **proc** (a1, a2: **aid**) **returns** (**true**)
% Returns **true** if a1 and a2 name the same activity; otherwise returns false.

Figure 7-1:Informal specification of the data type **aid**.

argument. This implicit argument must appear as the first argument in the argument list of the routine, and has type **aid**. Thus, for example, an operation with external specification

op = **proc** (x1: t1, ...) ...

might have internal specification

op = **proc** (a: **aid**, x1: t1, ...) ...

The identifier used to declare the implicit argument may be chosen at the convenience of the programmer. When a routine with distinct internal and external specifications is invoked, the implicit argument is assigned the value of the **aid** of the invoking activity, and the other arguments are assigned the values of the corresponding actuals.

Second, a cluster may supply two additional operations, called **commit** and **abort**. We call these special operations *completion operations*. Their interfaces are as follows:

commit = **proc** (a: **aid**, x: **rep**) **signals** (failure(**string**))

abort = **proc** (a: **aid**, x: **rep**) **signals** (failure(**string**))

These operations are intended to be called by the system (say, with arguments a and x) when an activity a that used the object represented by x commits or aborts. To let the system know that an activity has used an object, the routines inside a cluster may call the special procedure **register**, which has the following interface:

register = **proc** (a: **aid**, x: **rep**) **signals** (completed)

An invocation of **register** will signal *completed* if the activity named by the first argument has already committed or aborted. Otherwise the invocation will return. Sometime after the activity a completes, the system will invoke the appropriate completion operation defined in the cluster (**commit** if the activity commits, **abort** if the activity aborts) with arguments a and x. If the completion operation signals *failure*, the system will try again at some future time. If a committing activity is a subactivity of another activity, and the invocation of the **commit**

operation terminates normally, the system will also register the activity's parent on the same object; thus, when the parent completes, the appropriate completion operation will again be invoked. (Similarly, if the parent commits and its **commit** operation terminates normally, the system will register its parent on the same object, and so on until a top-level activity is reached.)

We will use the **mutex** type and the **seize** statement in Argus to cope with internal concurrency. As in the previous chapter, we assume that activities do not fail when in possession of a mutex object.

Finally, we add the built-in data type **action_queue** to allow operations to wait for necessary preconditions. An informal specification of the operations provided by **action_queue** appears in Figure 7-2. We also provide the **block** statement to allow an activity to wait on an **action_queue**. The **block** statement has the following form:

> **block** expr1 **on** expr2

The first expression must evaluate to an **aid**, and the second to an **action_queue**. The **block** statement can appear only within a **seize** statement. When executed, it blocks the executing process on the specified **action_queue** on behalf of the specified **aid**, and releases the mutex object seized by the closest enclosing **seize** statement.

A process blocked on an **action_queue** is in one of two states: *asleep* or *waiting*. When a process executes a **block** statement, it is initially *asleep*. An *asleep* process on an **action_queue** changes to the *waiting* state when some other process executes the *notify* or *wake* operation on the **action_queue**. A process in the *waiting* state attempts to regain possession of the mutex object that was released when the process blocked, and is unblocked as soon as it regains possession.

7.2 Implementation of the Semiqueue Type

In this section we present an implementation of the semiqueue type using the linguistic constructs described in the previous section. An informal specification of semiqueues appeared in Figure 6-1. The implementation appears in Figure 7-4. It uses the type generator *log*; an informal specification of logs appears in Figure 7-3.

The representation of a semiqueue consists of three components enclosed in a mutex object. The components are: *committed*, which represents the items known to be in the semiqueue (they have been enqueued by activities that have committed to the top-level, and they have not been dequeued); *logs*, which is a collection of summary information about the operations executed by active activities; and *pending*, which is an activity queue used for blocking *deq* operations that cannot find an item to dequeue. The mutex object is used to prevent

data type action_queue is create, notify, wake, empty

% A process can add itself to an **action_queue** by executing the **block** statement, specifying
% an aid on whose behalf it wishes to wait. A process on an **action_queue** is in one of two
% states: *asleep* or *waiting*. A *waiting* process will be unblocked as soon as it can regain
% possession of the mutex object released when it blocked. An **action_queue** is *empty* if
% and only if no processes, asleep or waiting, are blocked on it.

create = **proc** () **returns** (action_queue)
% Returns a new, empty action queue.

notify = **proc** (q: action_queue, a: aid)
% Changes all *asleep* processes on *q* waiting on behalf of siblings of *a* or their descendants to
% *waiting*; all top-level activities are considered to be siblings of a top-level activity.

wake = **proc** (q: action_queue)
% Changes all *asleep* processes on *q* to *waiting*.

empty = **proc** (q: action_queue) **returns** (bool)
% Returns **false** if any process, *asleep* or *waiting*, is blocked on *q*; otherwise returns **true**.

Figure 7-2: Informal specification of the data type **action_queue**.

interference among concurrently executing operations on the same semiqueue by forcing them to run serially.

The summary for an activity consists of two parts: *enq*, which represents the items enqueued by the activity (or its committed descendants) and not subsequently dequeued; and *deq*, which represents the items dequeued by the activity (or its committed descendants), and contains sufficient information to be able to "undo" the *deq* operations if the activity aborts.

The implementation of *enq* works as follows: It finds the summary record for the invoking activity by calling the internal procedure *find_log*. It then adds the item to be enqueued to the list of items enqueued by the activity, registers the invoking activity and the semiqueue object (so the appropriate completion operation will be invoked by the system when the activity completes), and returns. The mechanism used for *enq* operations is like an intentions list: a record of the operation is kept, but the operation only becomes visible to the activity's siblings when the activity commits. If the activity aborts, the record of the operation is discarded.

The implementation of *deq* is more complex: It first looks for an item to dequeue by calling the internal procedure *find_elist*. *Find_elist* searches the committed items and the intentions lists for the calling activity and its ancestors, looking for a non-empty list. The lists searched by *find_elist* contain the enqueued items that are visible to the calling activity and that have not yet been dequeued. If no list is found by *find_elist*, *deq* blocks on the queue in the representation of the semiqueue, and tries again when some activity that used the semiqueue becomes visible to the calling activity. If a non-empty list is found by *find_elist*, *deq* removes

data type log[t: **type**] **is** create, fetch, store, delete, ancestors

% A *log*[*t*] object maps aids to *t* objects.

create = **proc** () **returns** (log[t])
% Returns a new, empty log.

fetch = **proc** (l: log[t], a: **aid**) **returns** (t) **signals** (not_found)
% Returns the *t* object associated with *a* in *l*, signalling *not_found* if *a* is not bound in *l*.

store = **proc** (l: log[t], a: **aid**, x: t)
% Binds *a* to *x* in *l*.

delete = **proc** (l: log[t], a: **aid**)
% Unbinds *a* in *l*.

root2leaf = **iter** (l: log[t], a: **aid**) **yields** (aid, t)
% Yields each ancestor of *a* (including *a* itself) with its associated binding, if it is bound in *l*;
% items are yielded in root-to-leaf order.

leaf2root = **iter** (l: log[t], a: **aid**) **yields** (aid, t)
% Yields each ancestor of *a* (including *a* itself) with its associated binding, if it is bound in *l*;
% items are yielded in leaf-to-root order.

Figure 7-3: Informal specification of the data type log.

the first item from the list and creates an undo record containing that item and the list. Next, if the item dequeued was not enqueued by the invoking activity (or one of its committed descendants), the undo record is added to the summary information for the activity. (If the same activity enqueued and then dequeued an item, there is no need to remember either operation; the net effect on the semiqueue will be the same regardless of whether the activity commits or aborts.) Finally, the invoking activity and the semiqueue are registered, and the item to be dequeued is returned.

The mechanism used for *deq* operations is like an undo log: If the invoking activity aborts, the information in the undo records is used to put the item back in the list from which it was removed, effectively "undoing" the operation.

The **abort** routine is simple: It undoes the *deq* operations executed by the aborting activity and its committed descendants by putting the dequeued items back in the lists from which they were removed, and then discards the summary (including the intentions list of enqueued items) for the aborting activity. Finally, if any *deq* operations were undone, the **abort** routine unblocks any pending *deq* operations, since the items that were returned from the aborting activity's undo list might now be visible to the pending operations.

The **commit** routine merges the summary for the committing activity with that for its parent. The activity's list of enqueued items is simply appended to its parent's list. The undo

Figure 7-4:Explicit implementation of the data type semiqueue.

semiqueue = **cluster** [item: **type**] **is** create, enq, deq

undo = **struct**[i: item,   % item returned by deq op.
              deleted_from: elist]   % elist it was removed from.

elist = **array**[item]   % intentions list for enq's, and list
                           % of fully committed enq'd items.

dlist = **array**[undo]   % undo log for deq's.

summary = **struct**[enq: elist,   % summary of operations executed by
                deq: dlist]   % a single activity.

aq = **action_queue**

components = **struct**[committed: elist,   % committed items in semiqueue.
                logs: log[summary], % summaries for all activities.
                pending: aq]   % pending deq ops.

rep = **mutex**[components]

% rep invariant:
%   for each activity a, and for each undo record u in rep.value.logs[a].deq, either
%   u.deleted_from = rep.value.committed, or there exists a proper ancestor a' of a such
%   that u.deleted_from = rep.value.logs[a'].enq

create = **proc** () **returns (cvt)**
     return(rep$create(components${committed: elist$new(),
                                    logs: log[summary]$create(),
                                    pending: aq$create()})))

     end create

% external spec: enq = **proc** (q: **cvt**, i: item)
enq = **proc**(a: **aid**, q: **cvt**, i: item)
     seize q do
          s: summary : = find_log(q.value.logs, a)
          elist$addh(s.enq, i)
          register(a, q)
          end
     end enq

Figure 7-4: (continued)

```
% external spec:  deq = proc (q: cvt) returns (item)
deq = proc (a: aid, q: cvt) returns (item)
    seize q do
        while true do
            visible: elist : = find_elist(q.value,a)
                except when none: block a on q.value.pending    % pause
                                        continue                % and retry
                    end
            u: undo : = undo${i: elist$reml(visible), deleted_from: visible}
            s: summary : = find_log(q.value.logs, a)
            if u.deleted_from ~ = s.enq
                then dlist$addh(s.deq, u) end % else operations cancel each other
            register(a, q)
            return(u.i)
            end
        end
    end deq


% gets the summary for a from logs.
find_log = proc (logs: log[summary], a: aid) returns (summary)
    return(logs[a])
        except when not_found: s: summary : = summary${enq: elist$new(),
                                                        deq: dlist$new()}
                    logs[a] : = s
                    return(s)
            end
    end find_log


% finds a non-empty elist visible to a (i.e., either committed, or belonging to an ancestor
% of a). If q.committed is non-empty it is returned.  Signals if no non-empty elist is found.
find_elist = proc (c: components, a: aid) returns (elist) signals (none)
    if elist$size(c.committed) ~ = 0 then return(c.committed) end
    for anc: aid, s: summary in log[summary]$root2leaf(c.logs, a) do
        if elist$size(s.enq) ~ = 0 then return(s.enq) end
        end
    signal none
    end find_elist
```

Figure 7-4: (continued)

```
commit = proc (a: aid, q: rep) signals (failure(string))
    seize q do
        qv: components : = q.value
        l: log[summary] : = qv.logs
        as: summary : = l[a]
            except when not_found: return end
        log[summary]$delete(l, a)
        ps: summary : = find_log(l, aid$parent(a))
            except when top: merge_enq(qv.committed, as.enq)
                            if elist$size(as.enq) ~ = 0 then aq$wake(qv.pending) end
                            return
                end
        merge_enq(ps.enq, as.enq)
        merge_deq(ps, as.deq)
        if elist$size(as.enq) ~ = 0 then aq$notify(qv.pending, a) end
        end
    end commit

abort = proc (a: aid, q: rep) signals (failure(string))
    seize q do
        as: summary : = q.value.logs[a]
            except when not_found: return end
        log[summary]$delete(q.value.logs, a)
        for u: undo in dlist$elements(as.deq) do
            elist$addh(u.deleted_from, u.i)
            end
        if dlist$size(as.deq) ~ = 0 then aq$wake(q.value.pending) end
        end
    end abort

% appends (in order) items in from onto to.
merge_enq = proc (to: elist, from: elist)
    for i: item in elist$elements(from) do
        elist$addh(to, i)
        end
    end merge_enq
```

Figure 7·4: (continued)

```
% appends (in order) undos in from onto to.deq, ignoring those undos u
% with u.deleted_from = to.enq (such deq's have committed to the
% level of the corresponding enq, so both operations can be forgotten).
merge_deq = proc (to: summary, from: dlist)
    for u: undo in dlist$elements(from) do
        if u.deleted_from ~ = to.enq
            then dlist$addh(to.deq, u)
            end
        end
    end merge_deq

end semiqueue
```

information for *deq* operations is similarly appended, except that records for items enqueued by the parent are discarded from the undo log. (The operations effectively cancel each other in this case.) In the case of a committing top-level activity, the tentatively enqueued items are appended to the list of committed items, and the undo log is discarded. Finally, if any items enqueued by the committing activity were added to its parent's intentions list. the **commit** routine unblocks pending *deq* operations invoked by activities to which the committing activity is now visible.

When a blocked *deq* operation is unblocked, there is no guarantee that it will be able to execute. For example, there may be several pending *deq* operations unblocked by the same completing activity, but it is possible that only one will actually be able to dequeue an item. Thus, the *deq* operation loops after blocking, and may block again if it still finds no items available for it to dequeue.

7.3 Remarks

In this section we discuss the relative merits of the implicit and explicit approaches. We begin in Section 7.3.1 by summarizing the conclusions to be drawn from the examples, including those presented in the appendix. Next, in Section 7.3.2, we compare the two approaches. Finally, in Section 7.3.3, we discuss related work.

7.3.1 Summary of Examples

First, in Section 7.3.1.1 we discuss the two implementations of the semiqueue type. Next, we summarize the conclusions to be drawn from the three examples in the appendix. Two of those examples are implementations of the *map* type, one using an implicit approach and the other an explicit approach. The third is an implementation of the *bank_account* type. We discuss the implementations of map in Section 7.3.1.2, and the implementation of bank_account in Section 7.3.1.3.

7.3.1.1 Implementations of the Semiqueue Type

The explicit implementation of the semiqueue type has a number of advantages over the implicit implementation. First, the implementation of the *deq* operation in the explicit implementation is more efficient than in the implicit implementation. In the implicit implementation the *deq* operation takes time proportional to the size of the representation of its semiqueue argument (in the worst case); in the explicit implementation the *deq* operation takes time proportional to the number of ancestors of the calling activity (again in the worst case). This difference arises because the explicit implementation has access to the names of invoking activities.

Second, the semiqueue type cannot be implemented in Argus to satisfy additional properties, like the restriction that items enqueued by a single activity be dequeued in the order in which they are enqueued; the explicit mechanism presented here does not suffer from this limitation. The problem with Argus is that the system does not update the states of all built-in atomic objects instantaneously, so it is possible for one activity to see that another activity has committed at one built-in object, and later to see that the activity still holds a lock on another built-in object. It is possible to change the semantics of Argus to avoid this problem. However, it might be expensive for the system to guarantee that commits and aborts appear instantaneous in a distributed system. In addition, even with this additional guarantee, it can be difficult to implement the "fifo"-like restriction on semiqueues.

Third, the implicit implementation of the semiqueue type uses busy-waiting to schedule *deq* operations, while the explicit implementation uses a signalling mechanism. The signalling mechanism can be significantly more efficient than busy-waiting: In the implicit implementation, the system has very little information on which to base scheduling decisions, and is quite likely to awaken a pending operation when the operation cannot proceed, and not to awaken a pending operation when in fact it can proceed. In the explicit implementation, a pending *deq* operation is awakened only if an activity that enqueued some items becomes visible to the activity waiting to *deq*, or if an activity that dequeued some items aborts, making those items available for other activities to dequeue.

Finally, the implicit implementation requires a "cleanup" routine to keep the size of the representation of a semiqueue from increasing forever; the explicit implementation accomplishes the same effect with the user-supplied completion operations, and does it more efficiently since it is possible to tell from the arguments to the completion operation exactly what information needs to be deleted from the representation and when it needs to be deleted.

7.3.1.2 Implementations of the Map Type

Maps are like associative memories, binding uids to other objects. A map provides three operations: *insert*, which adds a new binding to the map; *delete*, which deletes a binding from the map; and *lookup*, which retrieves the binding for a specified uid from the map.

Both implementations of the map type in the appendix use a two-phase locking protocol, based on the state machine LOCK described in Chapter 5. They illustrate how a locking protocol that chooses locks based on the results of operations as well as their arguments can be implemented.

The two implementations provide the same level of concurrency. In addition, they use similar representations for maps: both represent a map using a non-atomic table protected by a

mutex; tables are like maps, except that they are not atomic. The implementations differ in several ways, however.

First, the implicit implementation requires an internal "cleanup" routine (like the cleanup routine in the implicit implementation of semiqueues) to keep the representation from growing too large. The explicit implementation uses the completion operations to update the representation of a map as activities complete, avoiding this kind of periodic "garbage collection" of the representation.

Second, the implicit implementation uses busy-waiting for scheduling, while the explicit implementation uses action_queues. Separate queues are used for each uid, with the result that a pending operation on a uid will be awakened only if an activity that used the same uid aborts or becomes visible to the activity that invoked the operation (i.e., commits to their least common ancestor). A queue is stored for a uid only if an operation on the uid that was invoked by an active activity was forced to wait. If conflicts causing operations to wait are relatively rare, then the number of queues stored should be small.

Third, the explicit implementation keeps track explicitly of the uids used by an activity; this effect is achieved in the implicit implementation by the fact that the system keeps track of the built-in atomic objects used by each activity.

The explicit implementation appears significantly more complex than the implicit implementation: It contains 156 lines of code, while the implicit implementation contains 64 lines of code. Some of the extra code (31 lines) is needed for managing queues. Much of the rest (65 lines) is needed to manage information and perform tasks that are handled automatically by the system in the implicit approach, for example, the completion operations themselves, and keeping track of the uids used by each activity.

7.3.1.3 Implementation of the Bank_account Type

A bank account provide three operations: *deposit*, which adds a specified amount to the account; *withdraw*, which removes a specified amount from the account, signalling *insufficient_funds* if the balance is too low; and *balance*, which returns the current balance in the account. The explicit implementation in the appendix illustrates how an implementation of an atomic type can provide more concurrency than can be achieved with two-phase locking. It also serves to illustrate further the greater expressive power and, in some cases, ease of use, of the explicit approach: The lack of access to the names of invoking activities in the implicit approach makes it difficult, if not impossible, to construct an implicit implementation that permits comparable levels of concurrency.

7.3.2 Comparison

We base our comparison of the implicit and explicit approaches on the two implementations of the semiqueue type presented above, and on the implementations of the map and bank_account types presented in the appendix.

The explicit approach described earlier in this chapter is strictly more powerful than the implicit approach in Argus. It extends the Argus approach in three ways: implementations of atomic types have explicit access to the names of invoking activities; the programmer can supply code that is run when activities complete to update the representations of objects; and implementations can explicitly awaken a pending operation when it might be able to proceed, rather than using busy-waiting to perform scheduling.

By permitting implementations to access the name of the activity that invokes an operation, we achieve greater flexibility in structuring the representations of objects. and avoid some of the limitations of the implicit approach. For example, the *deq* operation in the implicit implementation of the semiqueue type takes time proportional to the size of the representation of its semiqueue argument (in the worst case). In the explicit implementation, the *deq* operation takes time proportional to the number of ancestors of the calling activity (again in the worst case).

Similarly, it is impossible using the implicit approach to satisfy the constraint on semiqueues that, as long as at most one activity dequeues items at a time and that activity does not abort, items enqueued by a single activity should be dequeued in the order in which they were enqueued. The reader can easily verify that the explicit implementation of the semiqueue type satisfies this constraint.

Achieving this flexibility in structuring representations appears to require access to the names of invoking activities, but does not require the use of user-supplied completion operations. We could provide a "query" operation on aid objects, permitting an implementation to find out from the system whether a given activity is still active, and if not, whether it has committed or aborted. An implementation could periodically check the status of active activities, and update the representation of an object appropriately when told that an activity has completed. In this way we could achieve much of the effect of user-supplied completion operations, without having them executed automatically by the system.

While user-supplied completion operations are not needed for flexibility in structuring representations, they may be necessary for achieving efficient scheduling of operations. The implicit approach in Argus uses busy-waiting to schedule operations; as illustrated by the implementations of the semiqueue and map types, the queuing mechanism in the explicit approach presented here permits much more control over scheduling of operations. The mechanism relies on user-supplied completion operations to awaken blocked operations

explicitly. It is not clear whether comparable control over scheduling can be achieved without user-supplied completion operations.

The explicit implementations that we have presented are obviously more complex than the implicit ones. Much of this complexity arises from the need to represent explicitly certain kinds of information that are handled automatically by the system in the implicit approach. For example, in the explicit implementation of the map type in the appendix, we explicitly represent the uids used by an activity; in the implicit implementation, this is not necessary because the system keeps track of the built-in atomic objects used by activities.

Some of the complexity, however, arises out of the desire for more efficient and more concurrent implementations. Some of the information that is handled automatically by the system in the implicit approach may be more efficiently used when structured differently (e.g., compare the two semiqueue implementations): As discussed above, the explicit approach, by permitting implementations to access the names of invoking activities, provides control over this structure, while the implicit approach does not. In addition, the management of queues introduces complexity, but also permits much more efficient scheduling.

More work may be required of the programmer in the explicit approach, since it may be necessary to supply completion operations. However, in the implicit approach, it is frequently necessary to provide an internal operation that compacts the representations of objects (e.g., the *cleanup* routine in the implicit implementation of the semiqueue type); this kind of internal garbage collection is not needed in explicit implementations.

In addition, the explicit approach does not require linguistic support that is specific to a particular local atomicity property. The **tagtest** statement in Argus is designed to support implementations of dynamic atomic types, and provides little if any help in building implementations of static atomic types. In contrast, the explicit structure presented above, by permitting explicit access to **aids**, provides the programmer with tools that can be used for implementing static atomic types as well as dynamic atomic types. Hybrid atomic types can also be implemented using the explicit structure, although for efficient management of old versions it might be useful to add explicit "initiate" operations for read-only activities.

Neither the linguistic support in Argus, nor the extensions we presented earlier in this chapter, supports optimistic implementations. It appears relatively easy to do so using an explicit approach, simply by allowing programmers to supply an explicit "pre-commit" operation for objects to vote on whether to allow an activity to commit. It appears very difficult to extend an implicit approach to permit optimistic implementations.

The use of mutual exclusion to control internal concurrency is just a simple way of making the implementations of operations "atomic," by forcing them to run serially. A more general

approach is to use atomic actions. making each execution of an operation a top-level activity. This approach appears viable, however, only if implementations have access to the names of invoking activities. In an implicit structure, certain steps in the implementation of an operation (e.g., operations on lower-level atomic objects) must be executed on behalf of the invoking activity. yet there is no way for a process to act on behalf of two activities at the same time. Even in an explicit structure, this approach may have difficulties. It is not clear whether a queuing mechanism comparable to our **action_queues** can be designed to work with top-level activities instead of with seize statements.

In summary, the expressive power of the implicit approach is limited in several ways. However, implementations using an explicit approach appear more complex. It is not clear whether an intermediate approach can be found that provides the efficiency of the explicit approach but avoids some of the complexity.

7.3.3 Related Work

Some recent work at CMU by Spector and Schwarz [Schwarz & Spector 82] has considered how to build "atomic objects." Spector and Schwarz ignore recovery, however, and focus only on locking implementations. They appear to suggest that the system should manage "lock tables" automatically, but do not describe in detail how the programmer can describe the set of lock modes and their conflict relationships to the system. In addition, it is sometimes difficult for an operation to tell in advance what kind of lock it will need; for example, in implementations of the map type, an *insert* operation will need a different kind of lock depending on whether it terminates normally or signals. This indicates that an automatic locking mechanism might be difficult to use.

Work at Newcastle on recovery blocks [Anderson *et al.* 78, Anderson & Lee 79, Verhofstad 76] investigated recovery techniques for building user-defined data types. While concurrency was not considered, alternative program structures were explored. The *inclusive recovery* scheme in [Anderson *et al.* 78] is similar to our implicit approach, while their *disjoint recovery* scheme is similar to our explicit approach. The authors of [Anderson *et al.* 78, Anderson & Lee 79] note that the inclusive scheme provides limited control over recovery and can be less efficient than the disjoint scheme, but that implementations in the disjoint scheme can be more complex than those in the inclusive scheme. These conclusions are similar to our conclusions about the implicit and explicit approaches for implementing atomic types.

Allchin [Allchin & McKendry 83, Allchin 83] has also investigated "atomic objects." He has focused on the implicit approach, attempting to make the system do as much of the work as possible. As we have discussed above, this approach provides limited expressive power. One interesting mechanism that he proposes is a queuing mechanism for an implicit scheme: He associates one queue with each object. An activity can wait on a queue, and will be

awakened when another activity that used the same object becomes visible to it. However, he does not consider examples like the map type (illustrated in the appendix of this dissertation, and discussed in Section 7.3.1), for which it may be important to have finer control over scheduling. In particular, it is important in the implementation of map in Section A.2 to be able to discard queues that are no longer needed for pending operations. This appears impossible in the implicit queuing mechanism proposed by Allchin.

Chapter Eight

Summary and Conclusions

8.1 Summary

Atomicity is a useful organizational concept for reducing the complexity of a concurrent system. If activities are atomic, concurrency can be ignored when checking that the state of the system remains consistent. In this dissertation we have explored how to specify and implement "atomic types," which support atomicity of activities.

One of the most important contributions of this dissertation is a specification framework that permits the behavioral specification of an object to be derived systematically from a specification of its serial behavior. In addition, our framework permits the programmer of an individual activity to ignore how atomicity is achieved, and to focus on the serial behavior of each object.

In studying what it means for an object to be atomic, we generalized existing work on concurrency control in three important ways:

- Our definition of atomicity is based on an explicit specification of the desired behavior for the objects shared by activities.

- Our definition of atomicity encompasses both serializability and recoverability.

- We identify *local* properties of individual objects that ensure *global* atomicity of activities.

Our focus on local properties appears to be unique. A few papers (e.g., [Bernstein *et al.* 81, Korth 81a, Beeri *et al.* 83]) have considered using specifications to increase concurrency, and at least one paper [Lynch 83] has considered recovery in addition to synchronization. However, we know of no other work focusing on local properties of objects, or dealing with more than one of the above three issues.

We also presented a novel two-phase locking protocol, and verified its correctness. Our presentation and analysis of the protocol, unlike published descriptions of existing protocols, cover recovery as well as synchronization. The protocol extends existing protocols in two ways: It permits operations to be partial and non-deterministic, and it permits the results of operations, as well as their arguments, to be used in synchronizing activities.

Finally, we presented several examples of implementations of atomic types, illustrating how existing techniques for synchronizing and recovering activities can be extended to achieve

greater concurrency. In addition, we presented two different linguistic mechanisms. The mechanism in Argus is clearly limited in expressive power; the alternative that we explore, while more powerful, may also be harder to use.

8.2 Conclusions and Further Work

Each of the three local atomicity properties defined in this dissertation characterizes the operation of a large class of protocols. Each defines limits on the concurrency among activities that can be permitted by types. In addition, each is *optimal*, in the sense that no more concurrency can be permitted without violating atomicity. The three properties do not characterize all protocols, however. It might be worth investigating how to extend other protocols (see, e.g., [Silberschatz & Kedem 80, Kung & Robinson 81]) to cope with user-defined operations.

Our optimality results indicate that, without further constraints, objects satisfying different local atomicity properties (e.g., dynamic atomicity and static atomicity) cannot be used in the same system. As systems grow and existing systems are interconnected, it will become necessary to cope with connecting systems that use different protocols. Existing protocols will need to be extended so that the interactions among different protocols can be handled gracefully.

At the moment it does not appear that any one local atomicity property is clearly "best," in the sense of providing better performance than any other. This means that a system designer must choose which local atomicity property to use in a system. More experimentation is needed to determine the kinds of applications for which each local property provides good performance.

It is clear from the example implementations that we have presented that implementing an atomic type is a difficult task. One of the reasons for the complexity of these implementations is the interaction between synchronization and recovery. The traditional approach to analyzing atomicity in database systems is to assume that synchronization and recovery are provided by separate modules in the implementation. However, our presentation and verification of the locking protocol in Chapter 5 should make it clear that synchronization and recovery can interact in subtle ways, particularly when we consider operations other than reads and writes. Implementations of atomic types need to be explored further in an attempt to develop a better understanding of how they should be structured.

Of the two program structures that we have studied, the explicit structure is the more general, but requires more work on the part of the programmer. It seems unlikely that general support could be designed that would significantly reduce the complexity of implementations and at

the same time provide the expressive power of the explicit structure studied here. On the other hand, it may be that specialized support can be developed for particular implementation strategies, such as locking. Experimental work is also needed to evaluate the need for certain kinds of expressive power; it may be that the expressive power limitations of the mechanisms in Argus are not important for many applications.

Perhaps the biggest problem with the Argus mechanism is its lack of flexibility: It appears difficult to use for implementing types other than dynamic atomic types. A more flexible mechanism would be preferable, if only because it would permit experimentation with different local atomicity properties.

Atomic types are clearly useful in many applications for supporting atomicity. However, it is not clear how often the programmer will need to take on the job of implementing synchronization and recovery. It may be that the performance demands of most applications can be satisfied without basing synchronization on the specifications of objects, or that only a few types in a system need to provide this extra concurrency. It is clear that this extra concurrency will be provided rarely as long as it is so difficult to construct an implementation that provides it.

A hard problem that we have not considered is how to test and debug a concurrent system. Activities using atomic types, as well as implementations of atomic types, will need to be tested and debugged. It may be that the needs of testing and debugging will also influence the choice of mechanisms for implementing an atomic type.

We have barely touched on the problem of ensuring "reliable" operation of a system. Among the possible requirements that an application might make are that systems remain available, that information not be lost, and that activities make progress. Existing protocols for meeting these requirements need to be extended to cope with user-defined operations, and the interactions of these extensions with implementations of atomic types need to be explored.

References

[Allchin 83]
>   Allchin, J. E.
>   *An architecture for reliable decentralized systems.*
>   PhD thesis, Georgia Institute of Technology, September, 1983.
>   Available as Technical Report GIT-ICS-83/23.

[Allchin & McKendry 83]
>   Allchin, J. E., and McKendry, M. S.
>   Synchronization and recovery of actions.
>   In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 31-44. ACM, Montreal, Canada, August, 1983.

[Anderson & Lee 79]
>   Anderson, T., and Lee, P.
>   *The provision of recoverable interfaces.*
>   Technical Report 137, University of Newcastle upon Tyne Computing Laboratory, March, 1979.

[Anderson et al. 78]
>   Anderson, T., Lee, P., and Shrivastava, S.
>   A model of recoverability in multilevel systems.
>   *IEEE Transactions on Software Engineering* SE-4(6):486-494, November, 1978.

[Atkinson & Hewitt 77]
>   Atkinson, R. R., and Hewitt, C.
>   Synchronization in Actor Systems.
>   In *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages*, pages 267-280. ACM, January, 1977.

[Beeri et al. 83]
>   Beeri, C., et al.
>   A concurrency control theory for nested transactions.
>   In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 45-62. ACM, Montreal, Canada, August, 1983.

[Bernstein & Goodman 81]
>   Bernstein, P. A., and Goodman, N.
>   Concurrency control in distributed database systems.
>   *ACM Computing Surveys* 13(2):185-221, June, 1981.

[Bernstein & Goodman 83]
>   Bernstein, P. A., and Goodman, N.
>   Multiversion concurrency control -- theory and algorithms.
>   *ACM Transactions on Database Systems* 8(4):465-483, December, 1983.

[Bernstein et al. 81]
Bernstein, P., Goodman, N., and Lai, M.-Y.
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. February, 1981.

[Best 82]
Best, E.
Relational semantics of concurrent programs.
In Dines Bjorner, editor, Preliminary Proceedings of the IFIP TC-2 Working
Conference, pages 301-322. North-Holland Publishing Company, June, 1982.

[Best & Randell 81]
Best, E., and Randell, B.
A formal model of atomicity in asynchronous systems.
Acta Informatica 16:93-124, 1981.

[Birrell et al. 82]
Birrell, A. D., et al.
Grapevine: an exercise in distributed computing.
CACM 25(4):260-274, April, 1982.

[Campbell & Habermann 74]
Campbell, R. H. and Habermann, A. N.
The specification of process synchronization by path expressions.
· Springer-Verlag, 1974, .

[Chan et al. 82]
Chan, A., et al.
The implementation of an integrated concurrency control and recovery scheme.
Technical Report CCA-82-01, Computer Corporation of America, March, 1982.

[Dahl et al. 70]
Dahl, O.-J., et al.
The Simula 67 common base language.
Publication No. S-22, Norwegian Computing Center, Oslo, 1970.

[Davies 73]
Davies, C. T.
Recovery Semantics for a DB/DC System.
In Proceedings of the ACM Annual Conference, pages 136-141. ACM, Atlanta, GA,
1973.

[Davies 78]
Davies, C. T.
Data processing spheres of control.
IBM Systems Journal 17(2):179-198, 1978.

[DuBourdieu 82]
DuBourdieu, D.J.
Implementation of distributed transactions.
In Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and
Computer Networks, pages 81-94. 1982.

[Eswaren *et al.* 76]
Eswaren, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L.
The notions of consistency and predicate locks in a database system.
Communications of the ACM 19(11):624-633, November, 1976.

[Fischer & Michael 82]
Fischer, M. J., and Michael, A.
Sacrificing serializability to attain high availability of data in an unreliable network.
In *Proceedings of the Symposium on Principles of Database Systems.* ACM, March,
1982.

[Gifford 79]
Gifford, D. K.
Weighted Voting for Replicated Data.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages
150-162. ACM SIGOPS, Pacific Grove, CA, December, 1979.

[Ginsburg 75]
Ginsburg, S.
Algebraic and automata-theoretic properties of formal languages.
North Holland/American Elsevier, New York, 1975.

[Goree 83]
Goree, J. A.
Internal consistency of a distributed transaction system with orphan detection.
Master's thesis, MIT, January, 1983.
Available as MIT/LCS/TR-286.

[Gray 78]
Gray, J.
Notes on Database Operating Systems.
In *Lecture Notes in Computer Science,* Volume 60: *Operating Systems -- An Advanced
Course.* Springer-Verlag, 1978.

[Gray *et al.* 81]
Gray, J.N., et al.
The recovery manager of the System R database manager.
ACM Computing Surveys 13(2):223-242, June, 1981.

[Herlihy 84]
Herlihy, M. P.
Replication methods for abstract data types.
PhD thesis, MIT, 1984.
Forthcoming.

[Hoare 74]
Hoare, C. A. R.
Monitors: an operating system structuring concept.
CACM 17(10):549-557, October, 1974.

[Korth 81a]
Korth, H. F.
Locking protocols: general lock classes and deadlock freedom.
PhD thesis, Princeton University, 1981.

[Korth 81b]
Korth, H. F.
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.

[Kung & Papadimitriou 79]
Kung, H.T., and Papadimitriou, C.H.
An optimality theory of concurrency control for databases.
In *Proceedings of the 1979 SIGMOD Conference.* Boston, MA, May, 1979.
More recent version available as MIT/LCS/TM-185.

[Kung & Robinson 81]
Kung, H.T., and Robinson, J.T.
On optimistic methods for concurrency control.
ACM Transactions on Database Systems 6(2):213-226, June, 1981.

[Lamport 76]
Lamport, L.
Towards a theory of correctness for multi-user data base systems.
Technical Report CA-7610-0712, Massachusetts Computer Associates, October, 1976.

[Lamport 78]
Lamport, L.
Time, clocks, and the ordering of events in a distributed system.
CACM 21(7):558-565, July, 1978.

[Lampson 81]
Lampson, B.
Atomic transactions.
In Goos and Hartmanis, editors, *Lecture Notes in Computer Science*, Volume 105: *Distributed Systems: Architecture and Implementation*, pages 246-265. Springer-Verlag, Berlin, 1981.

[Liskov 82]
Liskov, B.
On linguistic support for distributed programs.
IEEE Transactions on Software Engineering SE-6(3):203-210, May, 1982.

[Liskov & Scheifler 82]
Liskov, B., and Scheifler, R.
Guardians and actions: linguistic support for robust, distributed programs.
In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 7-19. ACM, January, 1982.
Revised version to appear in TOPLAS.

[Liskov & Zilles 74]
> Liskov, B., and Zilles, S. N.
> Programming with abstract data types.
> In *Sigplan Notices*, Volume 9: *Proceedings of the ACM SIGPLAN Conference on Very High Level Languages*, pages 50-59. ACM, 1974.

[Liskov *et al.* 81]
> Liskov, B., et al.
> CLU reference manual.
> In Goos and Hartmanis, editors, *Lecture Notes in Computer Science*, Volume 114. Springer-Verlag, Berlin, 1981.

[Liskov *et al.* 83]
> Liskov, B., et al.
> *Preliminary Argus reference manual.*
> Programming Methodology Group Memo 39, MIT Laboratory for Computer Science, October, 1983.

[Lynch 83]
> Lynch, N.A.
> Concurrency control for resilient nested transactions.
> In *Proceedings of the Second ACM Symposium on Principles of Database Systems.* March, 1983.

[Moss 81]
> Moss, J.E.B.
> *Nested transactions: an approach to reliable distributed computing.*
> PhD thesis, Massachusetts Institute of Technology, 1981.
> Available as Technical Report MIT/LCS/TR-260.

[Nelson 81]
> Nelson, B. J.
> *Remote procedure call.*
> PhD thesis, Carnegie-Mellon University Department of Computer Science, May, 1981.
> Available as CMU-CS-81-119.

[Owicki & Gries 76]
> Owicki, S., and Gries, D.
> Verifying properties of parallel programs: an axiomatic approach.
> *Communications of the ACM* 19(5):279-285, May, 1976.

[Owicki & Lamport 82]
> Owicki, S., and Lamport, L.
> Proving liveness properties of concurrent programs.
> *ACM Transactions on Programming Languages and Systems* 4(3):455-495, July, 1982.

[Papadimitriou 79]
> Papadimitriou, C.H.
> The serializability of concurrent database updates.
> *Journal of the ACM* 26(4):631-653, October, 1979.

122

[Pnueli 77]
Pnueli, A.
The temporal logic of programs.
In *Proceedings of the 18th Annual Symposium on Foundation of Computer Science*,
pages 46-57. IEEE, Providence, RI, October, 1977.

[Randell 75]
Randell, B.
System structure for software fault tolerance.
IEEE Transactions on Software Engineering SE-1(2):220-232, June, 1975.

[Reed 78]
Reed, D.P.
Naming and synchronization in a decentralized computer system.
PhD thesis, Massachusetts Institute of Technology, 1978.
Available as Technical Report MIT/LCS/TR-205.

[Reuter 82]
Reuter, A.
Concurrency on high-traffic data elements.
In *Proceedings of the Symposium on Principles of Database Systems*, pages 83-92.
ACM, Los Angeles, CA, March, 1982.

[Robson 81]
Robson, D.
Object-oriented software systems.
BYTE 6(8):74-86, August, 1981.

[Schroeder, *et al.* 84]
Schroeder, M. D., Birrell, A. D., and Needham, R. M.
Experience with Grapevine: the growth of a distributed system.
ACM Transactions on Computer Systems 2(1):3-23, February, 1984.

[Schwarz & Spector 82]
Schwarz, P., and Spector, A.
Synchronizing shared abstract types.
Technical Report CMU-CS-82-128, Carnegie-Mellon University, September, 1982.

[Shrivastava & Banatre 78]
Shrivastava, S.K., and Banatre, J.-P.
Reliable resource allocation between unreliable processes.
IEEE Transactions on Software Engineering 4(3):230-241, May, 1978.

[Silberschatz & Kedem 80]
Silberschatz, A., and Kedem, Z.
Consistency in hierarchical database systems.
JACM 27(1):72-80, January, 1980.

[Skeen 82]
Skeen, M. D.
Crash recovery in a distributed database system.
PhD thesis, University of California at Berkeley, May, 1982.
Available as UCB/ERL M82/45.

[Skeen & Birman 83]
Skeen, D., and Birman, K.
Resilient objects.
Technical Report TR 83-553, Cornell University Computer Science Department, 1983.

[Stark 84]
Stark, E. W.
Foundations of a theory of specifications for distributed systems.
PhD thesis, MIT, May, 1984.
Forthcoming.

[Verhofstad 76]
Verhofstad, J.S.M.
Recovery for multi-level data structures.
Technical Report 96, University of Newcastle upon Tyne, December, 1976.

[Verhofstad 78]
Verhofstad, J. S. M.
Recovery techniques for database systems.
ACM Computing Surveys 10(2):167-195, June, 1978.

[Weihl & Liskov 82]
Weihl, W., and Liskov, B.
Specification and implementation of resilient, atomic data types.
Computation Structures Group Memo 223, MIT, December, 1982.

[Wood 80]
Wood, W.G.
Recovery control of communicating processes in a distributed system.
Technical Report 158, University of Newcastle upon Tyne, 1980.

.

Appendix A

Example Implementations

In this appendix we present three example implementations of dynamic atomic types. The first, in Section A.1, is an implementation of a map data type using an implicit structure. The second, in Section A.2, is an implementation of the same data type using an explicit structure. The third, in Section A.3, is an implementation of a bank account data type using an explicit structure. This implementation illustrates how a data type that permits more concurrency than allowed by locking can be implemented. Finally, in Section A.4, we discuss the conclusions to be drawn from the examples.

A.1 Implicit Implementation of the Map Type

In this section we present an implementation of the *map* type using an implicit structure. An informal specification of the map type appears in Figure A-1. Maps are like associative memories, binding uids to other objects. In different maps, the uids can be bound to different types of objects. The type of the bound object for a particular map type is given by the parameter type *vtype*.

Maps permit substantial concurrency. *Insert*, *delete*, and *lookup* operations involving different uids commute, and so can be used by concurrent activities. Not much concurrency, however, is possible among operations involving the same uid. If we classify *insert* and *delete* operations that terminate normally (rather than signalling) as writers, and all other operations as readers, we see that readers commute with each other, and that a writer does not commute with any other operation involving the same uid. For example, if an activity inserts a binding for a uid, concurrent activities cannot execute any operations involving that uid, although the *lookup* operation could be performed by a concurrent activity if the insert had signalled *duplicate*. Similarly, if a *lookup* executed by an activity terminates normally, a concurrent activity cannot delete the same uid, while if a *lookup* executed by an activity signals *not_found*, a concurrent activity cannot insert the same uid.

The concurrency analysis above is the basis for the implementation of map in Figure A-2. The strategy employed in this implementation is similar to that used in the implementation of the semiqueue type presented earlier. A map is represented using a regular (non-atomic) table. Tables provide the same operations as maps, with the same serial specification, and in addition an iterator *pairs*, with the following specification:

    pairs = iter (t: table[vtype]) yields (uid, vtype)

data type map[vtype: type] is create, insert, delete, lookup

% Maps are like associative memories, binding uids to vtype objects; *map[vtype]* is dynamic
% atomic if *vtype* is dynamic atomic.

create = **proc** () **returns** (map)
% Returns a new, empty map (one containing no bindings).

insert = **proc** (m: map, u: uid, v: vtype) **signals** (duplicate)
% If *u* is not bound in *m*, binds *u* to *v* in *m*; otherwise signals *duplicate*.

delete = **proc** (m: map, u: uid) **signals** (not_found)
% If *u* is bound in *m*, unbinds *u* in *m*; otherwise signals *not_found*.

lookup = **proc** (m: map, u: uid) **returns** (vtype) **signals** (not_found)
% If *u* is bound in *m*, returns the associated *vtype* object; otherwise signals *not_found*.

Figure A-1: Informal specification of the data type map.

Pairs yields all pairs (u,v) such that u is bound to v in t. Unlike maps, however, tables are not atomic. Since the table is not atomic, changes to it made by activities that later abort are not undone; thus, the table maps uids to atomic variant objects rather than directly to the corresponding vtype objects. In addition, the table is enclosed in a mutex object to handle internal concurrency.

The implementation of map also uses the Argus built-in type **null**. **Null** is generally used as a kind of "place filler" in a variant type when the tag contains all necessary information. **Null** has exactly one immutable object, represented by the literal **nil**.

The implementations of *insert*, *delete*, and *lookup* are all similar. Each seizes the mutex, and then calls the internal procedure *find_status* to obtain the atomic variant associated with a given uid. Next, if the status object can be locked appropriately, its tag is checked, and then either information is returned about the binding or the status object is modified to reflect a change in the binding. (Of course, if the invoking activity later aborts, this modification to the status object will be undone.) If the status object cannot be locked appropriately, the operation **pauses**, releasing possession of the mutex. When possession is regained, the operation tries again.

Find_status gives the illusion that a status object exists for all uids. If the uid is bound to a status object, that one is returned. Otherwise, the uid is not bound by the map, so *find_status* creates a new status object with base state "absent." binds the uid to it in the table, and returns it.

When an activity that deleted a uid from the map commits, the status object associated with the uid will have tag "absent." Such status objects waste space: The same information can be represented by the absence of a binding for the uid in the table. The internal procedure

Figure A-2: Implicit implementation of the data type map.

```
map = cluster[vtype: type] is create, insert, delete, lookup

status = atomic_variant[present: vtype,
                          absent: null]

log = table[status]    % maps uids to status objects.

rep = mutex[log]

create = proc () returns (cvt)
    return(rep$create(log$create()))
    end create

insert = proc (m: cvt, u: uid, v: vtype) signals (duplicate)
    seize m do
        while true do
            s: status := find_status(m.value, u)
            tagtest s
                tag present: signal duplicate
                wtag absent: status$change_present(s, v)
                            return
                end
            pause  % couldn't lock s; wait and try again.
            end
        end
    end insert

delete = proc (m: cvt, u: uid) signals (not_found)
    seize m do
        while true do
            s: status := find_status(m.value, u)
            tagtest s
                wtag present: status$change_absent(s, nil)
                            return
                tag absent: signal not_found
                end
            pause  % couldn't lock s; wait and try again.
            end
        end
    end delete
```

Figure A-2: (continued)

```
lookup = proc (m: cvt, u: uid) returns (vtype) signals (not_found)
    seize m do
        while true do
            tagtest find_status(m.value, u)
                tag present (v: vtype): return (v)
                tag absent: signal not_found
                end
            pause   % couldn't lock s; wait and try again.
            end
        end
    end lookup


find_status = proc (l: log, u: uid) returns (status)
    cleanup(l)
    return (log$lookup(l, u))
        except when not_found:
                s: status := status$make_absent(nil)
                log$insert(l, u, s)
                return (s)
                end
    end find_status


cleanup = proc (l: log)
    enter topaction
        for u: uid, s: status in log$pairs(l) do
            tagtest s
                wtag absent: log$delete(l, u)
                end
            end
        end
    end cleanup


end map
```

cleanup, called by *find_status*, finds and removes such bindings from the table.

Note that *cleanup* runs as an independent activity. and removes only those status objects that are not being used by any active activity. *Cleanup* cannot remove status objects with tag "absent" that are being read by active activities: these status objects must be retained to prevent the "phantom record" problem [Eswaren *et al.* 76], in which one activity observes the absence of a binding for a given uid, and another activity adds a binding for the uid before the first activity completes; the first activity may then observe the state of the map both before and after the second activity, thus violating serializability.

Note that *insert*, *delete*, and *lookup* all call *find_status* each time they test whether the appropriate locks can be acquired. They cannot use the same status object each time, since it might have been removed from the table by another activity's call of *cleanup*. The mutual exclusion enforced by the **seize** statement, however, ensures that a status object returned by find_status will remain in the table at least until the calling operation releases the mutex.

A.2 Explicit Implementation of the Map Type

In this section we present an implementation of the map type using an explicit structure. The implementation appears in Figure A-5. It uses two types, *versions* and *set*, whose specifications appear in Figures A-3 and A-4, respectively.

This implementation of map can be viewed as an optimized implementation of the state machine LOCK discussed in Chapter 5, suitably extended to allow activities to be nested. The implementation does not store the entire sequence of operations executed by each activity, however; instead, it keeps track of a summary that is sufficient to update the committed state when the activity commits and to synchronize with other activities.

Recall from the previous section that we can classify *insert* and *delete* operations that terminate normally as writers, and all other operations as readers. Operations involving different uids commute, as do readers involving the same uid; a writer does not commute with any other operation involving the same uid. This implementation synchronizes operations by using read/write locks on individual uids. An operation can acquire a read lock on behalf of an activity as long as no concurrent activity holds a write lock on the same uid; in other words, as long as only ancestors of the requesting activity hold write locks. An operation can acquire a write lock on behalf of an activity as long as no concurrent activity holds a read or write lock on the same uid; on other words, as long as no non-ancestors of the requesting activity hold locks of any kind. When an activity commits, its locks are transferred to its parent; when an activity aborts, its locks are discarded.

The representation of a map is enclosed in a mutex object, and consists of three pieces:

data type versions[t: **type**] is create, can_read, can_write. read_lock. write_lock,
read, write, busy, commit, abort

% A versions[t] object is a stack of versions. each with an associated set of readers and
% writers. The usual read/write locking discipline for nested activities is observed: any
% number of concurrent readers, and at most one writer (with no concurrent readers).

create = **proc** (x: t) **returns** (versions[t])
% Returns a new object with base state x.

can_read = **proc** (v: version[t], a: **aid**) **returns (bool)**
% Returns **true** if and only if every writer of v is an ancestor of a.

can_write = **proc** (v: version[t], a: **aid**) **returns (bool)**
% Returns **true** if and only if every reader and writer of v is an ancestor of a.

read = **proc** (v: version[t], a: **aid**) **returns** (t) **signals (conflict)**
% If a can read v, returns the most recent version of v; otherwise signals *conflict*. Does *not*
% make a a reader of v (the operation *read_lock* must be used explicitly to set a read lock).

write = **proc** (v: version[t], x: t, a: **aid**) **signals (conflict)**
% If a can write v, removes any version of v associated with a and pushes x on v on behalf
% of a; otherwise signals *conflict*. If no conflict, also makes a a writer of v.

read_lock = **proc** (v: version[t], a: **aid**) **signals (conflict)**
% If every writer of v is an ancestor of a then makes a a reader of v; otherwise signals *conflict*.

write_lock = **proc** (v: version[t], a: **aid**) **signals (conflict)**
% If every reader and writer of v is an ancestor of a then makes a
% a writer of v; otherwise signals *conflict*.

busy = **proc** (v: version[t]) **returns (bool)**
% Returns **false** if and only if there are no readers or writers of v.

commit = **proc** (v: version[t], a: **aid**)
% If a is a reader (resp. writer) of v, makes a's parent (if any) a reader (resp. writer) of v,
% and removes a as a reader (resp. writer) of v. If a version was pushed on v on behalf of
% a, replaces version associated with a's parent (if any) with that associated with a.

abort = **proc** (v: version[t], a: **aid**)
% Releases locks held by a on v and discards any versions of v written by a.

Figure A-3: Informal specification of the data type versions.

data type set[t: type] is create, insert, delete, member, empty, append

create = **proc** () **returns** (set[t])
% Returns a new, empty set.

insert = **proc** (s: set, x: t)
% Inserts x in s.

delete = **proc** (s: set, x: t)
% Removes x from s.

member = **proc** (s: set, x: t) **returns (bool)**
% Returns **true** if and only if x is in s.

empty = **proc** (s: set) **returns (bool)**
% Returns **true** if and only if s is empty.

append = **proc** (to, from: set)
% Inserts all elements in from into to.

Figure A-4: Informal specification of the data type set.

items, which contains versions and locks for uids; logs, which records for each active activity a summary consisting of the uids for which it has executed operations; and pending, which contains an activity queue for uids. Items contains versions and locks only for those uids that are bound in the map or that have been used by an active activity. Pending contains queues only for those uids with pending operations; queues are created when needed, and removed as soon as all activities that used them either commit to the top level or abort. Items and pending use the type table, also used in the implicit implementation of map presented in the previous section. The mutex object is used to prevent interference among concurrently executing operations.

The implementation of insert works as follows: First, it finds the summary for the invoking activity, adds the uid to the summary, and registers the activity and the map. Then, it finds the versions for the uid, and attempts to read the most recent version. If there is a conflict, the operation blocks on the queue associated with the uid, and tries again when a conflicting activity becomes visible or aborts. If there is no conflict, the operation tests the tag of the version. If the tag is "present," the operation acquires a read lock on the version and signals duplicate. If the tag is "absent," the operation writes a new version with tag "present" and value v, and returns; a write lock is acquired in writing the new version, unless there is a conflict, in which case the operation blocks and tries again later.

The implementations of delete and lookup are similar to that of insert. They differ largely in the conditions under which they acquire particular types of locks.

The **commit** routine begins by finding the summary for the committing activity. For each uid touched by the activity, it finds the versions for the uid and transfers the activity's locks and

Figure A-5: Explicit implementation of the data type map.

```
map = cluster[vtype: type] is create, insert, delete, lookup

aq = action_queue

status = oneof[present: vtype,
               absent: null]

tentative = versions[status]

data = table[tentative]    % versions for each uid.

queues = table[aq]         % queues for pending operations on each uid.

summary = set[uid]         % uids touched by an activity.

components = struct[items: data,
                    logs: log[summary],
                    pending: queues]

rep = mutex[components]

% rep invariant:
%      if there exists a such that u ∈ rep.value.logs[a], then u is mapped by rep.value.items;
%      if there is a blocked operation involving u, then u is mapped by rep.value.pending;
%      a is a reader or writer of rep.value.items[u] if and only if u ∈ rep.value.logs[a];
%      if u is mapped by rep.value.pending then u is mapped by rep.value.items.

create = proc () returns (cvt)
    return(rep$create(components${items: data$create(),
                                  logs: log[summary]$create(),
                                  pending: queues$create()})))

    end create
```

Figure A-5: (continued)

```
% external spec: insert = proc (m: cvt. u: uid, v: vtype) signals (duplicate)
insert = proc (a: aid, m: cvt, u: uid, v: vtype) signals (duplicate)
    seize m do
        s: summary : = find_log(m.value.logs, a)
        summary$insert(s, u)
        register(a, m)
        t: tentative : = find_item(m.value.items, u)
        while true do
            tagcase tentative$read(t, a)
                tag present: tentative$read_lock(t, a)
                            signal duplicate
                tag absent: tentative$write(t, status$make_present(v), a)
                            return
            end
            except when conflict: q: aq : = find_queue(m.value.pending, u)
                                block a on q
                                continue
                end
        end
    end
end insert


% external spec: delete = proc (m: cvt, u: uid) signals (not_found)
delete = b[proc] (a: aid, m: cvt, u: uid) signals (not_found)
    seize m do
        s: summary : = find_log(m.value.logs, a)
        summary$insert(s, u)
        register(a, m)
        t: tentative : = find_item(m.value.items, u)
        while true do
            tagcase tentative$read(t, a)
                tag present: tentative$write(t, status$make_absent(nil), a)
                            return
                tag absent: tentative$read_lock(t, a)
                            signal not_found
            end
            except when conflict: q: aq : = find_queue(m.value.pending, u)
                                block a on q
                                continue
                end
        end
    end
end delete
```

134

Figure A-5: (continued)

```
% external spec: lookup = proc (m: cvt, u: uid) returns (vtype) signals (not_found)
lookup = proc (a: aid, m: cvt, u: uid) returns (vtype) signals (not_found)
    seize m do
        s: summary := find_log(m.value.logs, a)
        summary$insert(s, u)
        register(a, m)
        t: tentative := find_item(m.value.items, u)
        while true do
            tentative$read_lock(t, a)
                except when conflict: q: aq := find_queue(m.value.pending, u)
                                      block a on q
                                      continue
                    end
            tagcase tentative$read(t, a)
                tag present (v: vtype): return(v)
                tag absent: signal not_found
                end
            end
        end
    end lookup

find_log = proc (logs: log[summary], a: aid) returns (summary)
    return(logs[a])
        except when not_found: s: summary := summary$create()
                               logs[a] := s
                               return(s)
            end
    end find_log

find_item = proc (items: data, u: uid) returns (tentative)
    t: tentative := data$lookup(items, u)
        except when not_found: s: status := status$make_absent(nil)
                               t := tentative$create(s)
                               data$insert(items, u, t)
            end
    return(t)
    end find_item
```

Figure A-5: (continued)

```
find_queue = proc (pending: queues, u: uid) returns (aq)
    q: aq := queues$lookup(pending, u)
        except when not_found: q := aq$create()
                                queues$insert(pending, u, q)
            end
    return(q)
    end find_queue


commit = proc (a: aid, m: rep) signals (failure(string)
    seize m do
        mv: components := m.value
        as: summary := mv.logs[a]
            except when not_found: return end
        log[summary]$delete(mv.logs, a)
        is_top: bool := aid$top(a)
        for u: uid in summary$elements(as) do
            t: tentative := find_item(mv.items, u)
            tentative$commit(t, a)
            q: aq := queues$lookup(mv.pending, u)
                except when not_found: if is_top cand ~tentative$busy(t)
                                            cand status$is_absent(tentative$read(t, a))
                                            then data$delete(mv.items, u)
                                            end
                                    continue
                    end
            aq$notify(q, a)
            if is_top cand aq$empty(q)
                then queues$delete(mv.pending, u)
                    if ~tentative$busy(t) cand status$is_absent(tentative$read(t, a))
                        then data$delete(mv.items, u)
                        end
                end
            end
        ps: summary := find_log(mv.logs, aid$parent(a))
            except when top: return
                end
        summary$append(ps, as)
        end
    end commit
```

Figure A-5: (continued)

```
abort = proc (a: aid, m: rep) signals (failure(string)
    seize m do
        mv: components := m.value
        as: summary := mv.logs[a]
            except when not_found: return end
        log[summary]$delete(mv.logs, a)
        for u: uid in summary$elements(as) do
            t: tentative := find_item(mv.items, u)
            tentative$abort(t, a)
            q: aq := queues$lookup(mv.pending, u)
                except when not_found: if ~tentative$busy(t)
                                        cand status$is_absent(tentative$read(t, a))
                                            then data$delete(mv.items, u)
                                            end
                                    continue
                    end
            aq$wake(q)
            if aq$empty(q)
                then queues$delete(mv.pending, u)
                    if ~tentative$busy(t) cand status$is_absent(tentative$read(t, a))
                        then data$delete(mv.items, u)
                        end
            end
        end
    end
end abort

end map
```

version (if any) to its parent. It then unblocks pending operations on the uid that were invoked by activities to which the committing activity is now visible. Finally, if the versions or the queue for a uid are no longer needed, they are deleted from the representation of the map. After processing each uid, the **commit** routine adds the summary information for the activity to that for its parent.

The **abort** routine is similar: For each uid touched by the activity, it releases the activity's locks and discards its versions, unblocks all pending operations for the same uid, and then deletes versions and queues that are no longer needed.

The process scheduling in this implementation is safe in the sense that all *asleep* pending operations that can proceed will be changed to *waiting*, and eventually unblocked, when an activity completes. However, some pending operations that cannot proceed might also be unblocked. For example, suppose that an activity invokes the insert operation for a uid, and the operation is forced to block because several other activities have read locks on the uid. When one of the readers becomes visible to the blocked activity, the pending operation will be unblocked. However, it can not yet proceed, since there are still conflicting readers. Similarly, if several activities are blocked waiting to get a write lock on a uid, they will all be unblocked at the same time, but only one of them will obtain the lock and proceed.

The **commit** and **abort** routines delete the versions for a uid only if no processes are blocked on the uid's queue. This permits the implementations of *insert*, *delete*, and *lookup* to find the versions for a uid only once, before entering the loop; if one of these operations blocks, the uid will remain bound to the same versions object at least until the operation unblocks.

A.3 Explicit Implementation of the Bank_account Type

In this section we present an explicit implementation of the bank_account type. The serial specification of a bank account object was given in Figure 5-2; the corresponding informal specification of the bank account type appears in Figure A-6. (The signal *neg_arg* appears in the informal specification and not in the original serial specification because the arguments to the *deposit* and *withdraw* operations in the informal specification are integers rather than natural numbers.)

The implementation appears in Figure A-8. It permits significantly more concurrency than would be permitted by a locking implementation; for example, it permits activities to withdraw money concurrently from an account as long as the account contains sufficient money to cover all the withdrawals. (*Cf.* Section 5.4.2.) The implementations of the *deposit* and *withdraw* operations are similar to the implementations of *enq* and *deq* in the semiqueue implementation presented in Section 7.2. Deposits are handled using intentions lists, while

data type bank_account is create. deposit, withdraw, balance

% A bank account always has a non-negative balance. The type is dynamic atomic.

create = **proc** () **returns** (bank_account)
% Returns a new bank account with a balance of 0.

deposit = **proc** (b: bank_account. amt: **int**) **signals** (neg_arg)
% Adds *amt* to *b* if *amt* ≥ 0; otherwise signals *neg_arg*.

withdraw = **proc** (b: bank_account. amt: **int**) **signals** (insufficient_funds, neg_arg)
% If *amt* < 0 then signals *neg_arg*; otherwise if *amt* > *b* then signals *insufficient_funds*;
% otherwise subtracts *amt* from *b*.

balance = **proc** (b: bank_account) **returns** (**int**)
% Returns the current balance in *b*.

Figure A-6: Informal specification of the data type bank_account.

withdrawals that terminate normally are handled using undo logs. Withdrawals that signal and balance operations are handled separately. Not all concurrency permitted by on-line dynamic atomicity is permitted by the implementation; for example, withdrawals that signal cannot be executed concurrently with withdrawals that succeed. This additional concurrency could be permitted at the expense of a more complicated and less efficient implementation.

The implementation uses the data type *crowd*; an informal specification of crowds appears in Figure A-7. The representation of a bank account is enclosed in a mutex object, and consists of five pieces: *committed*, which represents the money known to be in the account (it has been deposited by activities that have committed to the top level, and it has not yet been withdrawn); *changes*, which is a collection of summary information about the operations executed by active activities; *reads*, which records the active activities that have read the account (i.e., executed *withdraw* operations that signalled or executed *balance* operations); *writes*, which records the active activities that have written the account (i.e., executed *withdraw* operations that succeeded or executed *deposit* operations); and *pending*, which is used for blocking all pending operations on the account.

The summary for an activity consists of two parts: *credit*, which represents the money deposited by the activity (or its committed descendants) and not subsequently withdrawn; and *debit*, which represents the money withdrawn by the activity (or its committed descendants), and contains sufficient information to be able to "undo" the *withdraw* operations if the activity aborts.

Operations that read the account exclude operations that write it, and vice versa, but readers can run concurrently with each other, as can writers. The only restriction is that a withdrawal operation can be executed only when the account is guaranteed to contain sufficient funds, regardless of withdrawals executed by concurrent activities.

data type crowd is create, add, conflicts, empty, commit, abort

% A crowd is a set of activity ids.

create = **proc** () **returns** (crowd)
% Returns a new, empty crowd.

add = **proc** (c: crowd, a: **aid**)
% Adds *a* to *c*.

conflicts = **proc** (c: crowd, a: **aid**) **returns** (**bool**)
% Returns **true** if and only if a non-ancestor of *a* is a member of *c*.

empty = **proc** (c: crowd) **returns** (**bool**)
% Returns **true** if and only if *c* is empty.

commit = **proc** (c: crowd, a: **aid**)
% If *a* is a member of *c*, removes *a* from *c* and adds *a*'s parent (if any) to *c*.

abort = **proc** (c: crowd, a: **aid**)
% Removes *a* from *c*.

Figure A-7: Informal specification of the data type crowd.

The internal procedure *find_lower_bound* is used to obtain a lower bound on the amount of money available to an activity. It adds the balances in the intentions list for each ancestor of the activity to the committed balance; since money required to cover withdrawals executed by other activities has been removed from the intentions lists, the amount computed by *find_lower_bound* is available to cover a withdrawal by the specified activity. The value returned by *find_lower_bound* equals the activity's view when there are no concurrent writers.

The implementation of *deposit* works as follows: First it checks for conflicting read operations. If there is a conflict, it blocks and tries again when one of the conflicting activities has completed. Otherwise, the invoking activity is added to the set of writers, the deposited amount is added to the activity's intentions list (using the internal procedure *credit*), the activity and the object are registered, and the operation returns. The money deposited by an activity will only become visible to the activity's siblings when the activity commits; if the activity aborts the record of the deposit will be discarded.

The implementation of *withdraw* is more complex. First, it uses *find_lower_bound* to compute a lower bound on the money available to the invoking activity. There are then three cases. If the amount to be withdrawn is greater than the lower bound, and there are no concurrent writers, then the lower bound equals the activity's view and the account has insufficient funds. In this case the operations records the activity as a reader, registers the activity and the object, and signals *insufficient_funds*. If the amount to be withdrawn is less than the lower bound, then the account can cover the withdrawal: If there are no concurrent readers then

Figure A-8: Explicit implementation of the data type bank_account.

```
bank_account = cluster is create, deposit, withdraw, balance

source = oneof[committed: null,
               active: aid]

undo = struct[debit: int,
              taken_from: source]

udlist = array[undo]

summary = record[credit: int,
                 debit: udlist]

aq = action_queue

components = record[committed: int,
                    changes: log[summary],
                    reads, writes: crowd,
                    pending: aq]

rep = mutex[components]

% rep invariant:
%       rep.value.committed ≥ 0; for each activity a, and for each undo record u in
%       rep.value.logs[a].debit, either tag(u.taken_from) = committed, or value(u.taken_from)
%       is mapped by rep.value.changes; if a ∈ rep.value.writes, then a is mapped
%       by rep.value.changes

create = proc () returns (cvt)
    return(rep$create(components${committed: 0,
                                  changes: log[summary]$create(),
                                  reads: crowd$create(),
                                  writes: crowd$create(),
                                  pending: aq$create()}))
    end create
```

Figure A-8: (continued)

```
% external spec: deposit = proc (b: cvt, amt: int) signals (neg_arg)
deposit = proc (a: aid, b: cvt, amt: int) signals (neg_arg)
    if amt < 0 then signal neg_arg end
    seize b do
        while true do
            if crowd$conflicts(b.value.reads, a)
                then block a on b.value.pending
                    continue
                else crowd$add(b.value.writes, a)
                    credit(b.value.changes, a, amt)
                    register(a, b)
                    return
                end
            end
        end
    end deposit

% external spec: withdraw = proc (b: cvt, amt: int) signals (insufficient_funds, neg_arg)
withdraw = proc (a: aid, b: cvt, amt: int) signals (insufficient_funds, neg_arg)
    if amt < 0 then signal neg_arg end
    seize b do
        bv: components := b.value
        while true do
            lb: int := find_lower_bound(bv, a)
            if amt > lb cand ~crowd$conflicts(bv.writes, a)
                then crowd$add(bv.reads, a)
                    register(a, b)
                    signal insufficient_funds
            elseif amt ≤ lb cand ~crowd$conflicts(bv.reads, a)
                then crowd$add(bv.writes, a)
                    debit(bv, a, amt)
                    register(a, b)
                    return
                else block a on bv.pending
                    continue
                end
            end
        end
    end withdraw
```

Figure A-8: (continued)

```
% external spec: balance = proc (b: cvt) returns (int)
balance = proc (a: aid, b: cvt) returns (int)
    seize b do
        while true do
            if crowd$conflicts(b.value.writes, a)
                then block a on b.value.pending
                    continue
                else crowd$add(b.value.reads, a)
                    register(a, b)
                    return(find_lower_bound(b.value, a))
                end
            end
        end
    end balance

% Returns a lower bound on the amount of money available to a; the value returned equals
% a's view if no concurrent activity has deposited or withdrawn money.
find_lower_bound = proc (c: components, a: aid) returns (int)
    lb: int : = c.committed
    for anc: aid, s: summary in log[summary]$ancestors(c.changes, a) do
        lb : = lb + s.credit
        end
    return(lb)
    end find_lower_bound

find_log = proc (logs: log[summary], a: aid) returns (summary)
    return(logs[a])
        except when not_found: s: summary : = summary${credit: 0, debit: udlist$new()}
                            logs[a] : = s
                            return(s)
            end
    end find_log
```

Figure A-8: (continued)

```
commit = proc (a: aid, b: rep) signals (failure(string))
    seize b do
        bv: components : = b.value
        crowd$commit(bv.reads, a)
        crowd$commit(bv.writes, a)
        aq$notify(bv.pending, a)
        as: summary : = bv.changes[a]
            except when not_found: return end
        log[summary]$delete(bv.changes, a)
        ps: summary : = find_log(bv.changes, aid$parent(a))
            except when top: bv.committed : = bv.committed + as.credit
                        return
                end
        ps.credit : = ps.credit + as.credit
        merge_debits(aid$parent(a), ps.debit, as.debit)
        end
    end commit

abort = proc (a: aid, b: rep) signals (failure(string))
    seize b do
        bv: components : = b.value
        crowd$abort(bv.reads, a)
        crowd$abort(bv.writes, a)
        aq$wake(bv.pending)
        as: summary : = bv.changes[a]
            except when not_found: return end
        log[summary]$delete(bv.changes, a)
        for u: undo in udlist$elements(as.debit) do
            tagcase u.taken_from
                tag committed: bv.committed : = bv.committed + u.debit
                tag active(ancestor: aid): anc_s: summary : = find_log(bv.changes, ancestor)
                                anc_s.credit : = anc_s.credit + u.debit
            end
            end
        end
    end abort

credit = proc (effects: log[summary], a: aid, amt: int)
    s: summary : = find_log(effects, a)
    s.credit : = s.credit + amt
    end credit
```

Figure A-8: (continued)

```
debit = proc (c: components, a: aid, amt: int)
    s: summary := find_log(c.changes, a)
    for anc: aid. anc_s: summary in log[summary]$leaf2root(c.changes, a) do
        if anc_s.credit > 0 then
            if anc ~ = a then
                if amt > anc_s.credit
                    then
                        udlist$addh(s.debit,
                            undo${debit: anc_s.credit, taken_from: source$make_active(anc)})
                    else
                        udlist$addh(s.debit,
                            undo${debit: amt, taken_from: source$make_active(anc)})
                    end
                end
            if amt > anc_s.credit
                then amt := amt - anc_s.credit
                    anc_s.credit := 0
                else anc_s.credit := anc_s.credit - amt
                    amt := 0
                end
            end
        if amt = 0 then return end
        end
    if amt > 0 then
        c.committed := c.committed - amt
        udlist$addh(s.debit, undo${debit: amt, taken_from: source$make_committed: nil})
        end
    end debit


% appends undos in from onto to, ignoring undos with taken_from = parent.
merge_debits = proc (parent: aid, to: udlist, from: udlist)
    for u: undo in udlist$elements(from) do
        tagcase u.taken_from
            tag committed:
            tag active(ancestor: aid): if ancestor = parent then continue end
            end
        udlist$addh(to, u)
        end
    end merge_debits


end bank_account
```

the activity is recorded as a writer; the specified amount is removed from the committed balance and the intentions lists of the activity's ancestors, and recorded in the undo log (using the internal procedure *debit*), with sufficient information to be able to put the money back in the intentions lists from which it was removed; the activity and the object are registered; and the operation returns. In all other cases the operation blocks and tries again later.

Debit searches the intentions lists for the activity and its ancestors, removing money from the intentions lists and adding appropriate undo records to the activity's undo log so that the money can be returned to the proper list if the activity aborts. (It searches from the activity toward the root to try to minimize the impact on concurrent activities.) If the activity withdraws money that it (or one of its committed descendants) had deposited, the operations cancel (at least in part), and no record of the canceling part is kept. If the activity and its ancestors have not deposited sufficient money to cover the withdrawal, money is removed from the committed balance instead. *Debit* assumes that the lower bound on the funds available to the activity is greater than the amount to be withdrawn; otherwise the committed balance would become negative, violating the representation invariant. It is easy to see that the call to *debit* in the implementation of *withdraw* satisfies this precondition.

. The implementation of *balance* is simple: If there are no conflicting writers, the invoking activity is recorded as a reader and registered with the object, and the activity's view is returned. Otherwise the operation blocks and tries again later.

The implementation of **commit** follows the pattern of the semiqueue implementation presented earlier. First, it commits the activity in the crowds of readers and writers, and unblocks pending operations that might now be able to proceed. Then, it finds the summaries for the activity and its parent, deletes the summary for the activity, and merges its intentions lists and undo logs into its parent's. (Undos that cancel with the parent's deposits are discarded.) If the activity is a top-level activity, the deposits are added into the committed balance, and the entire summary is discarded.

The **abort** routine is also similar to the **abort** routine in the semiqueue implementation. It aborts the activity in the crowds, and then unblocks all pending invocations. Next, it retrieves the summary for the activity, deletes it, and uses the information in the undo list to return all withdrawn money to the intentions list from which it was taken.

Withdrawals that signal could be run concurrently with withdrawals that succeed and with deposits if sufficient information were maintained to determine an upper bound on the money available to an activity. Reuter [Reuter 82] has explored an implementation of a similar type (to be used for reserving seats on flights in an airline reservation system) that exploits this idea. However, the implementation is quite subtle, and probably inefficient.

A.4 Remarks

The implementation of the bank_account type is interesting for two reasons. First, it illustrates how a type that permits more concurrency than can be obtained using locking can be implemented. Second, we have been unable to construct an implicit implementation of the bank_account type that provides a comparable level of concurrency; while it is difficult to argue that no such implementation exists, it is clear that such an implementation would be quite complex. The problem is partly due to the inability to tell which changes to the representation were made by which activity (because the representation cannot be organized to group data by aids), and partly due to the asynchronous processing of commits and aborts by the system. In contrast to the implicit approach, the explicit approach permits a reasonably systematic approach to building implementations like the bank_account implementation in Figure A-8.

It is also instructive to compare the two implementations of the map type. First, the explicit implementation is much more efficient in scheduling operations. While the implicit implementation uses busy-waiting, the explicit implementation uses separate queues for each uid for which an operation is blocked, and only unblocks a pending operation when an activity that used the uid of interest to the operation becomes visible to the activity that invoked the operation. A queue is actually stored for a uid only if an activity that invoked an operation on the uid was forced to wait and is still active.

Second, the explicit implementation does not need a "cleanup" operation to keep the representation from growing too large. Instead, versions for a uid can be discarded when an activity completes, precisely at the time that they are no longer needed. The kind of periodic "garbage collection" performed by the cleanup operation in the implicit implementation of map seems characteristic of implicit implementations.

Third, in the explicit implementation the programmer keeps track of the uids used by an activity. Otherwise, when an activity completed, it would be necessary to commit or abort the activity on the versions for all uids; unless the map were quite small, this approach would be too inefficient. In the implicit implementation, the system keeps track automatically of the built-in atomic objects used by each activity, avoiding having to represent and manipulate this information explicitly.

Finally, it is instructive to compare the sizes of the two implementations. The implicit implementation contains 64 lines of code, while the explicit implementation contains 156 lines of code. This additional complexity arises from several sources. First, the **tagtest** statement in Argus provides an efficient encoding of the lock tests in the bodies of the loops in the *insert*, *delete*, and *member* routines; the explicit implementation requires a total of 12 extra lines of code for testing and setting locks. Second, 31 lines of the explicit implementation are

devoted to managing queues. and could be removed if busy-waiting were used instead. Third. 13 lines. not counting those in the completion operations, are devoted to keeping track of the uids used by each activity. Finally. the completion operations contain 37 lines. not counting those needed to manage queues.

Appendix B

Index of Definitions

This appendix contains an index of the terms and notations defined in Chapters 2 through 5.