MIT/LCS/TR-327

DEBUGGING DISTRIBUTED COMPUTATIONS IN A NESTED ATOMIC ACTION SYSTEM

Sheng Yang Chiu

This blank page was inserted to preserve pagination.

Debugging Distributed Computations in a Nested Atomic Action System

by

Sheng Yang Chiu

December, 1984

This research was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661/N00014-83-K-0125.

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

Debugging Distributed Computations in a Nested Atomic Action System

by

Sheng Yang Chiu

Submitted to the
Department of Electrical Engineering and Computer Science
on December 24, 1984 in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Abstract

Concurrent and distributed programs are hard to debug. In this thesis, we argue that structuring activities as nested atomic actions can make debugging such programs much like debugging traditional sequential programs. To support the argument, we present a method for debugging computations in the Argus language and system. Our method is applicable to other action systems since it depends only on the atomicity properties of actions.

To debug a computation in our method, the user inspects a serial execution that is equivalent to the original computation. The debugging process involves two phases. In the first phase, the user examines pre- and post- states of actions to isolate the action that exposes the bug. In the second phase, the debugging system re-executes code to reproduce the details of the culprit action. The user can repeat this re-execution and can use standard "break-and-examine" tools on it to isolate the bug.

Our debugging system supports the method by saving a partial history when an action runs. This history consists mainly of recovery versions of objects. The system also timestamps the termination of actions so it can determine from the saved versions the values of objects in an action's pre- and post- states. The debugging system itself uses pre-states to repeat actions. Our work presents the first detailed design that uses recovery versions and timestamps for debugging.

Thesis supervisor: Barbara H. Liskov
Title: Professor of Computer Science and Engineering

Keywords: debugging, nested atomic actions, distributed programs, recovery versions, timestamps.

Acknowledgments

I thank my thesis advisor, Barbara Liskov, for the countless hours she spent teaching me how to do research and to write. I am grateful for her kindness and her example.

I thank my readers, Dave Reed, Bert Halstead, and Bill Weihl, for pointing out and then helping to smooth the rough edges in the draft I submitted for my defense. Bill's copious comments were much appreciated.

I thank Maurice Herlihy, Gary Leavens, Bob Scheifler, and Paul Johnson for serving as willing sounding boards for some of my ideas.

I thank Juliet Sutherland, Jim Restivo, Brian Oki, Kathy Yelick, Jennifer Lundelius, and Sharon Perl for listening when I need to talk.

I thank my parents for their constant understanding and love.

I thank my daughter, Christine, for the joy that she is to me.

Finally, I thank my wife, Marti, for her constant understanding, companionship, and love, and for shouldering more than her share of family responsibilities at the expense of her own thesis research and writing.

Table of Contents

Table of Figures

Chapter One

Introduction

The asynchrony of interactions among concurrent activities and the non-determinism of node and network failures make a concurrent and distributed computation hard to understand and to repeat. Concurrent and distributed programs, therefore, are hard to debug.

The main conclusion of this thesis is that structuring activities as nested atomic actions makes debugging concurrent and distributed programs easier. To support this conclusion, we present a method for debugging computations in the Argus language and system [Liskov & Scheifler 83, Liskov 84]. Using the method, a person debugs a concurrent and distributed computation much like he or she would debug executions of traditional sequential programs. The method and implementation approach presented in this thesis are applicable to other action[1] systems, even though details may differ.

Actions (sometimes called *transactions* in the literature) [Davies 73, Eswaren et al. 76, Lomet 77, Reed 78, Gray 81, Lampson 81, Moss 81] are activities that are characterized by the following two properties: serializability and recoverability. Actions are *serializable* in that if a group of them should run concurrently, their effects on the system state will be as if they had run one after another in some serial order. (The system state may be distributed among multiple nodes in a network.) An action is *recoverable* in that if it should for some reason fail to complete successfully, all of its effects on the system state will be undone and will not be visible to other actions. An atomic action, therefore, groups together operations on the system state into a unit that is indivisible to other actions, in spite of concurrency and in spite of node and network failures.

[1] For succinctness, we sometimes drop the adjective "atomic".

In a nested action system [Davies 73, Reed 78, Moss 81], an action may itself contain other actions, some of which may run sequentially, some concurrently. The hierarchy of actions that are nested within some particular action can be arbitrarily deep. We will often use terminology that pertains to a tree data structure to describe relationships among actions in this hierarchy, e.g., *child*, *parent*, *sibling*.

We call actions that are nested within other actions *subactions*; we call those that are not nested within other actions *topactions*. Subactions have the following properties:

1. A subaction is serializable with respect to other sibling subactions.
2. A subaction is recoverable.
3. When a subaction aborts, its parent is not aborted.
4. When a subaction commits, it commits only to its parent action: its effects will be visible to its parent and to other sibling subactions but will not be visible to the world that is outside of its parent unless and until its parent also commits.
5. The effects of a committed subaction are undone if its parent aborts.

Subactions, therefore, provide controlled concurrency within an action, and act as fine-grained firewalls for failures.

If all activities are actions, understanding the effects of the concurrent execution of a group of activities becomes no harder than understanding some serial execution of the activities. The interleaving of asynchronous interactions among concurrent activities can be safely ignored. In addition, the effects of failures are also easier to understand. Failures due to node crashes and network outages are converted to action aborts. This cuts down on the number of states that failures can leave the system and also reduces the complexity of user code for dealing with failures.

Our debugging method makes use of the atomicity properties of actions. A user examines an equivalent serial execution when debugging a computation of

concurrent activities. An *equivalent serial execution* is a serial execution of the activities that has the same effects on the system state as the original computation. The debugging system provides an equivalent serial execution by re-executing code using the limited amount of history it has saved when the computation ran. The amount of information that the debugging system saves to support the reproducibility of an equivalent serial execution is fairly small: the information consists mainly of data that the Argus run-time system creates to support the recoverability of actions.

1.1 Overview of Thesis

In this thesis, we propose a method for debugging atomic actions. The method uses action trees, together with a serialization order, to summarize a computation to the user. A *computation* is a group of topactions; an *action's tree* is the hierarchy of subactions that are contained within the action; a *serialization order* is the ordering of sibling subactions and topactions in some equivalent serial execution.

In the method, a node in an action tree is viewed as a map from a *pre-* state to a *post-* state. (A state is a map from object identifiers to object values.) The value of an object in the pre-state of an action A is the net effect at the object of all modifications made by actions serialized before A and all modifications made by ancestors before A ran. The value of an object in the post-state of A is the pre-A value of the object updated by the modifications of A, if any. A person who is debugging an action will be interested primarily in objects that are accessible from the action's environment, i.e., variables that are global to the action, as well as the arguments and input to the action and the results and output from the action.

In the method, a user follows the progress of an action by doing a *serial walk* of the action's tree. In a serial walk, committed subactions of a node are visited in serialization order. Where appropriate, the user can choose to ignore the details of any subaction in the tree.

Debugging a faulty action in our method involves three phases. In *Phase Zero*,

some subset of an action's history is collected as the action runs. The history that is collected is used for supporting the next two phases of the method, and is the subject of the next subsection of this overview. Phase Zero history is saved for all actions that run in a program of interest. Older saved history, however, is discarded as space is needed. Phase Zero does not require any user intervention.

Phase One is the first of the two interactive phases in the method. In Phase One, the user uses the computation's action trees, the serialization order of siblings, and other saved history to narrow a bug to as "small" an action in a tree as possible: this is the youngest action that maps a "correct" pre-state to an "incorrect" post-state. (Action A is younger than action B if A is contained within B.) The user, not the debugging system, decides whether an action's pre- or post- state is correct. The debugging system helps by displaying the value of an object in an action's pre- or post- state on request from the user.

Sometimes the bug becomes obvious once the user narrows it to an action; at other times, it is not. If the bug is still not obvious after the fault has been narrowed to an action, the user moves on to *Phase Two*. In this phase, the faulty action's code is re-executed, using the data collected during the original computation to recreate the action's history. A single thread of control is used when an action is retraced; concurrent siblings are retraced in their serialization order. The user uses the usual break-and-examine tools of sequential debugging (e.g., breakpointing and single-stepping) on the single thread of control in the re-execution to isolate the bug.

Our debugging method has the following characteristics:

1. The method helps a user in debugging a computation only after the computation has already run. The generation of a computation with particular characteristics, perhaps through the use of appropriate test data and run-time control of the timing of interactions among concurrent activities, is beyond the scope of this thesis.
2. The method helps a user debug only functional bugs. A computation, when viewed as a map from a pre-state to a post-state, manifests a functional bug if its pre-state is transformed into a post-state that is not

expected. The method is not designed for isolating bugs that have to do with performance, e.g., starvation.

3. The computations that the method helps a user in debugging are those produced from executing Argus programs in a run-time system that is implemented correctly. A study of how the Argus system itself may be debugged is interesting in its own right, but is beyond the scope of this thesis.

We will not present a user interface for our debugging system. Instead, we simply assume that there is a reasonable way for a person to use the system.

1.1.1 Implementing the Method

In Argus, actions are guaranteed to be atomic only if they share *atomic objects*. Atomic objects are objects that provide synchronization and recovery for actions that access them. Atomic objects that are built into Argus use two-phase locking [Eswaren et al. 76] for concurrency control and back-up versions for recovery [Gray et al. 81]. Locks are automatically acquired and versions automatically created when actions invoke operations on atomic objects. The run-time system dispenses of an action's locks and versions appropriately when the action terminates. The rules for lock and recovery management in Argus are given by Moss [Moss 82]. We will describe these rules in detail in the next chapter.

An atomic object in Argus can also be user-defined. A programmer implements new atomic abstract data types in Argus by judiciously combining non-atomic and atomic objects. *Non-atomic* objects are objects that do not provide the synchronization and recovery that are found in atomic objects.

We introduce timestamps and multiple versions into Argus so the debugging system can provide values of atomic objects in an action's pre- and post- state, and can retrace action histories. We stress that the versions and timestamps are used by the debugging system only and are not used by the Argus run-time system or by Argus programs.

As it turns out, the recovery versions that are created for built-in atomic objects to ensure action recovery on aborts are mainly what the debugging system needs. The timestamps are generated with Lamport clocks [Lamport 78] and are assigned to actions when the actions terminate. Since Argus releases locks only when an action terminates, these timestamps order the lock points of actions and give a valid serialization order [Eswaran et al. 76, Bernstein & Goodman 81]. We show how these timestamps, together with versions, can be used to give the pre- and post- values of atomic objects for topactions as well as subactions, regardless of whether the actions committed or aborted.

We also show how versions and timestamps can be used to retrace an action's history in serial-walk order. In the retrace, the appropriate portion of the program is re-executed. Even though the same objects are referenced in the re-execution, "old" recovery versions are used so that a retrace of an action reads the same values as the original computation. A retrace accesses only saved history in an object and, therefore, does not disrupt other actions. Recovery versions and timestamps have been used in concurrency control; our work is the first detailed design that uses them for debugging nested actions.

A programming system that supports actions should permit users to define their own atomic types. The usual arguments forwarded in support of abstract data types apply here [Liskov & Zilles 74]. Increased concurrency is also an oft-cited reason. However, there is as yet no consensus on how implementations of user-defined atomic types should be supported. Weihl [Weihl 84], Schwarz and Spector [Schwarz & Spector 84], Allchin and McKendry [Allchin & McKendry 83], and Herlihy [Herlihy 84] have proposed mechanisms that differ from Argus's [Weihl & Liskov 82]. Ease of debugging, both of actions that use the objects and of implementations of the types, is one of the factors that should be considered when evaluating a mechanism. This thesis contributes in a modest way by studying the "debuggability" of Argus's user-defined atomic objects. In Argus, non-serializable interleaving of activities are allowed at objects within the internal representation

(**rep**) of a user-defined atomic object, as long as the interleaving does not cause the behavior of the user-defined atomic object as a whole to be non-serializable. The mechanism provided by Argus constrains the user to synchronize actions through the use of built-in atomic objects in the **rep**. Recovery also depends on the proper use of built-in atomic objects. We show how to derive pre- and post- action values of user-defined atomic objects that are correctly implemented, and how to retrace operations at these user-defined atomic objects using the history saved for pre- and post- action values. We also demonstrate the use of this same history to isolate bugs in the implementation of a user-defined atomic type.

1.2 Related Work

Work related to that of this thesis can be categorized under two major areas: concurrency control in the database field, and debugging systems.

1.2.1 Concurrency Control

[Bayer et al. 80] and [Stearns & Rosenkrantz 81] are generally cited as the first works to use recovery versions (called *before values*) for increasing the concurrency of locking schemes. Whereas Bayer et al. [Bayer et al. 80] and Stearns and Rosenkrantz [Stearns & Rosenkrantz 81] used a single recovery version in each object to reduce read-write conflicts, DuBourdieu [DuBourdieu 82] and Chan et al. [Chan et al. 82] used all the recovery versions created at an object to ensure that read-only actions never delay or abort update actions, and vice versa. (Update actions are actions that modify at least one object.) Update actions use standard two-phase locking to synchronize among themselves. Read-only actions, on the other hand, do not use locking to ensure consistent views of the database. Instead, when a read-only action begins execution it takes note of the set of update actions that committed before it. When a read-only action accesses an object, it reads the latest version written by an action recorded in its list of committed update actions.

Bernstein and Goodman [Bernstein & Goodman 83] modified the scheme in

[DuBourdieu 82] and [Chan et al. 82] to use Lamport clocks to timestamp the termination of update actions. A read-only action can now be given an arbitrary timestamp that is equal to or less than the current value of its local Lamport clock, to read a consistent view of the system state. It would read, at each object, the version created by the action that has the largest timestamp less than its own.

Timestamps and multiple versions were used earlier by Reed [Reed 78] for concurrency control without locks.

The support of pre- and post- action values of objects in our work is an adaptation of [DuBourdieu 82], [Chan et al. 82], and [Bernstein & Goodman 83] to debugging. Translated into their framework, the actions of Argus would be update actions, regardless of whether they actually modify the system state; debugging requests for pre- and post- action values would be read-only actions. However, our work differs in three important respects. First and most important, the versions and timestamps that we maintain are for debugging, not concurrency control. Second, whereas their work assumes a system of single-level actions, ours take into account nesting of actions. Third, we are interested in not only the views of committed actions, as is the case for DuBourdieu et al., but also views of actions that aborted. This is because we may sometimes want to find out why an action aborted.

1.2.2 Debugging Systems

Schiffenbauer [Schiffenbauer 81] described a system that supports debugging a concurrent and distributed computation in "real-time", i.e., while the computation runs. The system does not record any history of the computation and, therefore, cannot guarantee that the computation can be repeated. The person who is debugging has exactly one pass through the computation to catch a bug, should it surface. Also, inherent in such "real-time" systems is the problem of ensuring that debugging activities, e.g., breakpointing, do not interfere with the sequencing and timing of the computation's events so that an improbable computation results, i.e., a computation that is so unlikely to occur in the absence of the debugging system as to

make the debugging session effectively useless. Ensuring this non-interference constitutes a major part of Schiffenbauer's work. Schiffenbauer proposes a solution for a system, where processes communicate only through messages, that uses virtual time and requires all messages to be directed through a central node. In the scheme, suspending a process P requires all processes that are resident in the same node as P to be suspended, as well. The need to direct all messages through a central node and to suspend all processes in a node when one of them is suspended makes Schiffenbauer's system somewhat impractical.

Our debugging system is more akin to systems that monitor computations. Such systems fall into two classes: those that do not support reproducibility of computations and those that do. Systems in the first category are designed mainly for performance evaluation and general understanding of a program's dynamic characteristics. Those in the second category are mainly for debugging the functional behavior of computations. Our work quite obviously belongs to the second category.

We begin with a review of some systems in the first category. Some examples of such systems are reported by Model [Model 79], McDaniel [McDaniel 77], and Gertner [Gertner 80]. The system in [Model 79] works for uniprocess artificial intelligence representation systems. Predefined probes continuously log occurrences of "basic operations" on "fundamental system structures." The log processing system then reports high-level information about the monitored computation interactively.

METRIC [McDaniel 77] is an extremely flexible monitoring system for a distributed environment. A user determines the information to be logged through calls on special routines. The user may ask that this information be filtered through user-defined processes before it is eventually logged. Because of flexibility of content in the log, which may be distributed, METRIC leaves it up to the user to write programs to process and analyze the log.

The system in [Gertner 80] is also a monitoring system for distributed programs. Processes are assumed to communicate only through messages. Events that are logged consist of inter-process messages as well as process state changes that are registered with pseudo-messages. Each logged message carries three timestamps: the time the sender queued the message, the time the receiver received the message, and the time the receiver completed processing the message. The system supports a command language that allows the user to ask for various interesting time intervals that are associated with a user-defined (sub)computation. The user defines a (sub)computation with a finite state machine.

The class of monitoring systems that collect information to support the debugging of a computation's functional behavior is exemplified by EXDAMS [Balzer 69] and the system proposed by Smith [Smith 81]. EXDAMS is designed for uniprocess programs. The source code has to be run through a pre-processor for the automatic insertion of debugging statements before compilation. Execution of the compiled code will generate an audit trail of its execution sequence into a "history tape." This tape is used as a database, together with the source code and symbol table, for a sophisticated interactive query system that works by simulating the execution sequence. Some of the query system's capabilities include forward and backward execution and "flowback" analysis in which the user can request a display of how information flowed through the program to produce a specified value.

The system in [Smith 81] is designed for multiprocess but not distributed programs. As in [Gertner 80], processes communicate via messages. In [Smith 81], interprocess communication and system calls translate into events that are placed on a FIFO event queue for execution by the run-time system (called the *kernel*). The kernel executes events on the list one at a time. Execution of an event may cause some other event to be appended to the list. The execution of a process may be transcribed for later replay. This consists of logging all events that were executed on behalf of the process. The system reported in [Smith 81] may also be used for real-time control of a process. This is done by controlling the execution of the event

queue. Processes may be suspended by delaying the events that represent their system calls. Messages to processes may likewise be delayed or altered through event manipulation.

Our debugging system is more like that in [Smith 81] than EXDAMS in that it saves just enough information so it can create an equivalent serial execution of an action through re-executing code. Our system does not save a complete audit trail. In addition, unlike most works in debugging concurrent and/or distributed computations, our system allows activities to communicate through shared data. In fact, even though message-passing is also taken into account, our work is much more oriented toward shared data as the means of communication among activities.

1.3 Plan of Thesis

Chapter Two introduces Argus. The chapter gives an overview of the structure of programs and computations in Argus, but does not describe details of the language constructs. The chapter explains the implementation of synchronization and recovery in the atomic types that are built into Argus.

Chapter Three presents the essential ideas of the proposed debugging method. The rest of the thesis concentrates on modifications that need to be incorporated into Argus to support the method.

Chapter Four is the heart of this thesis. It demonstrates how versions, timestamps, and other history can be put together to provide pre- and post- action values of built-in atomic objects.

Chapter Five shows how an equivalent serial execution of an action can be created from the history that is collected for pre- and post- action values. The proposed retracing technique does not interfere with the use of the system's objects by other actions.

Chapter Six extends the method to include user-defined atomic objects. In particular, it provides details for determining the pre- and post- action values of these objects and for retracing operations at these objects. Chapter Six assumes that user-defined atomic types are correctly implemented.

Chapter Seven is about debugging implementations of user-defined atomic types. It discusses the isolation of bugs in the implementation of atomicity. History that is already collected to provide pre- and post- action values is sufficient for determining and isolating these atomicity bugs.

Chapter Eight sums up the contributions of this thesis and suggests further work.

Chapter Two

Argus

The Argus language and system [Liskov & Scheifler 83, Liskov 84] provides the context within which we study how to debug distributed computations that are structured as trees of nested atomic actions. In this chapter, we introduce Argus. We concentrate on the model of computation (Section 2.1) and the implementation of synchronization and recovery (Section 2.2), rather than on details of the language constructs.

In Argus, programs may generate computations that are not atomic, in that if the computations were to run concurrently the net effect on the system state will not be equivalent to any serial execution of the computations involved. Most programs written in Argus, however, are expected to be structured to produce atomic computations. In this thesis, we focus on the debugging of this "atomic" subset of Argus programs. Programs in this subset conform to properties that constrain how objects are used to convey information from one action to another. We discuss these properties in Section 2.3.

2.1 Overview

Argus is an integrated programming language and system that is designed to make the implementation of concurrent and distributed programs easier. Argus is derived from the object-oriented, sequential programming language CLU [Liskov et al. 81].

The underlying hardware base that Argus assumes is comprised of *nodes* connected (only) via a communications network. A node consists of one or more processors, one or more levels of memory, and any number of devices. A node can

communicate with any other node when the network is functioning properly. Argus makes no other assumptions about the speed or connectivity of the network. The system that supports Argus, however, is assumed to exhibit only fail-stop [Schlichting & Schneider 83] failures. For example, when there is something wrong with a node, the node crashes; when a disk fails, it becomes inaccessible; when the network malfunctions, messages are lost — Argus is never given a duplicate message or a bad message that looks like a good one.

Argus is designed for applications that concern the manipulation and preservation of long-lived, on-line data. Airline reservation systems and banking systems are examples of such applications.

An application implemented in Argus will consist of one or more communicating *guardians*. A guardian is a virtual node that provides a service or controls access to one or more resources, e.g., databases or devices. It consists of an address space, entirely contained in a single physical node, in which *data objects* and *processes* reside. Data objects, as well as guardians, belong to abstract data types with defined sets of *states* and *operations*. A guardian's operations are called *handlers*. When a handler is called, a process is created at the called guardian to run the handler's code. The process runs as an action.

Actions in the same guardian communicate with each other through the guardian's data objects. The data objects that provide the synchronization and recovery for actions are called *atomic objects*. Atomic objects in Argus may either be built-in or user-defined. (We describe built-in atomic objects in Section 2.2 and user-defined atomic objects in Chapter Six.) Since synchronization and recovery are likely to be expensive to implement, some objects in Argus do not implement them. These objects are called *non-atomic objects*.

Actions in Argus are guaranteed to be atomic only if they share atomic objects. However, it is possible for actions to share non-atomic objects and still be atomic if some conditions are satisfied. We list these conditions in Section 2.3.

An action calls a handler of another guardian, in much the same way it would invoke an operation of a data object in its own address space. Handler calls are implemented as remote procedure calls. Arguments and results are passed by value in messages to and from the target guardian; it is impossible to pass a reference to a local data object to another guardian. This rule ensures that a guardian retains control of its own objects. The method supported by Argus for the transmission of abstract data objects is described in [Herlihy & Liskov 82]. It requires the user to provide two procedures with an object's type. The first procedure, called *encode*, translates the type's objects (with their current values) into objects that the Argus system knows how to transmit, e.g., arrays of integers. The second procedure, called *decode*, does the inverse translation at the receiving end. The objects created by *decode* are initialized to the values that were transmitted.

When an action C makes a handler call, Argus creates a subaction, referred to as the *call action*, to make the call on C's behalf. The *handler action* that runs the handler activation is itself a subaction of the call action (and a grandchild of C). Should there be problems with the handler call, the system can simply abort the call action, without aborting C, and be assured that the handler action will have no effect on the system state, even if it commits. This use of nested actions guarantees that remote procedure calls have *at-most-once* semantics [Liskov 82] in spite of network or node failures.

An action in a guardian may spawn possibly concurrent *in-line subactions*, i.e., subactions that run in the same guardian as the parent, for increased concurrency and failure control. It may also create *nested topactions*. A nested topaction is not a subaction of its parent; once created, a nested topaction is just like any other topaction with respect to its parent. Nested topactions are meant to execute benevolent side effects, i.e., changes that do not affect the abstract state of a subsystem. For example, in a naming system a name look-up may cause information to be copied from one location to another to speed up subsequent look-ups of that name. Copying the data within a nested topaction ensures that the changes remain in effect even if the parent action aborts.

When a parent action creates an in-line subaction, a nested topaction, or a handler action, the parent is suspended until the created child terminates. A parent and child action never run concurrently. This simplifies interactions between ancestors and descendants.

A guardian may also contain processes that are not actions. These processes run in the **background code** of the guardian and serve mainly to initiate (non-nested) topactions. For example, a guardian that functions as a command interpreter may use a non-action process to interact with the user and to initiate the topactions that run the commands. Debugging non-action activities is beyond the scope of this thesis. Furthermore, we assume that these activities do not interfere with the actions that we are debugging.

Guardians are resilient to node crashes in that the Argus system will automatically reinstate a guardian after its node crashes and recovers. Data objects that are part of the guardian's *stable* state will be recovered with the guardian. If atomic, a stable object will be recovered to the value that was last written by a committed topaction; otherwise, it is given the value it had when the guardian was first created[2].

In Argus, node failures are reflected as aborts of actions. All active actions in a node at the time of a crash will be aborted. The effects of all committed subactions that ran in the node but whose topaction ancestors are still active at the time of the crash will also be undone; the topaction ancestors of these committed subactions will themselves eventually abort.

We note that all actions in Argus, including those that abort, see consistent views because of the locking rules and an orphan detection mechanism [Liskov 84]. (An *orphan* is any active action whose results are no longer wanted (see also [Nelson 81]).)

[2]except for a special kind of non-atomic objects called *mutexes*. We will describe mutexes in Chapters 6 and 7.

In conclusion, Argus provides an abstract machine where the effects of concurrency and hardware failures on distributed data are much more easily understood and managed than in the physical network of nodes. This is due to the support of activities as actions and the resilience of guardians and atomic data objects to crashes.

2.2 Atomic Objects

All immutable types in Argus are obviously atomic: since objects of these types are immutable, there is no need to synchronize accesses and there are no changes to undo when an action aborts. Examples of immutable types and immutable type generators include **integer**, **sequence** and **struct**.

There are only three kinds of mutable atomic types in Argus, namely **atomic_record**, **atomic_array** and **atomic_variant**. These types use two-phase locking for synchronization and versions for recovery.

A built-in mutable atomic object may be pictured conceptually as shown in Figure 2-1. (The picture is meant to suggest the information that is kept and used. We are not implying that the implementation of built-in atomic objects is, or should be, done this particular way.) The data structure labeled *lockholders* in the figure contains the lock-holding actions and the kind of locks they have on the object. Locks are distinguished as either read (R) or write (W) locks. Before an action uses an object, it must first acquire a proper lock on the object. For unrelated actions, i.e., actions that are not contained within the same topaction, the usual locking rules apply: multiple readers are allowed, but readers exclude writers and a writer excludes readers and other writers. The locking rules are generalized to include nested actions in Figure 2-2 [Moss 82]. Note that locks are acquired automatically by the operations called by an action and are held until the completion of the action. This use of two-phase locking avoids the problem of *cascading aborts* [Randell 75, Gray 78]: if a lock on an object were released early, and the action later aborted, any

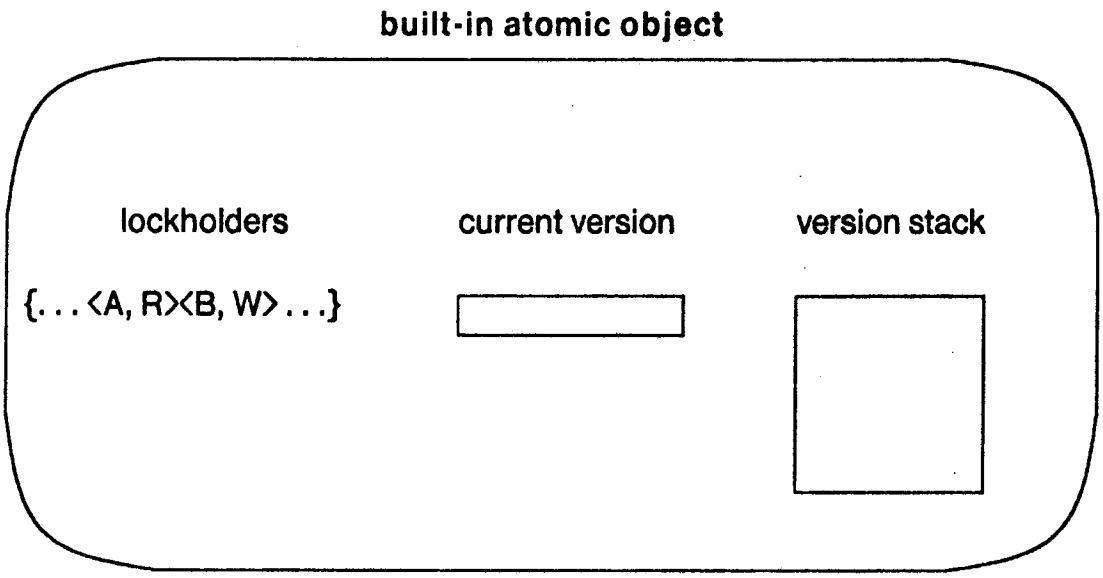


Figure 2-1: A Stylized Picture of a Built-in Atomic Object

action that had observed the new state of that object would also have to be aborted. This rule has the additional implication that for two unrelated actions, A and B, A can see the effects of B at an object only if B terminated before A. We will be making use of this implication in our selection of a serialization order for actions later in the thesis.

The *current version* holds the current value of the object. An action is allowed to access this current value only after it has acquired the proper lock on the object, per the locking rules given in Figure 2-2. The *version stack* component of the built-in atomic object is for recovery purposes. The *current version* and *version stack* are used and managed as explained in Figure 2-3. Note that when a copy of the *current version* is made for safekeeping in the *recovery stack*, we only copy down to contained atomic objects. Contained atomic objects need not be copied because they support their own recovery. Also, note that a version is removed from the

For an action A to acquire a read lock on object O:
All holders of write locks on O must be ancestors of A.

For an action A to acquire a write lock on object O:
All holders of read and write locks on O must be ancestors of A.

When an action A that has a lock on O commits:
If A is a subaction, then A's parent inherits A's lock on O.
Otherwise, A's lock on O is released.

When an action A that has a lock on O aborts:
A's lock is released.

Figure 2-2: Conditions for Lock Acquisition and Inheritance

version stack just as soon as it is not needed, i.e., when the computation has progressed far enough that it cannot revert to a state where the object has to be recovered to the value saved in the version.

In addition to serializability and recoverability, an atomic object that is in the stable state of a guardian is also *resilient* to crashes: it is recovered after a crash and is restored to the value that it held after the committed topaction that last modified it. Argus uses stable storage to support object resilience. When a topaction commits, the current values of all atomic objects on which the topaction holds write locks are written into stable storage with a two-phase commit protocol [Gray 78].

2.3 Atomic Subset of Argus

In this section, we characterize the subset of Argus programs that generate atomic computations. Our debugging method is meant to debug programs from this "atomic" subset. (We expect this atomic subset to constitute the majority of implemented Argus programs.)

When an action A reads object O:

A reads the *current version* of O.

When an action A modifies object O:

If A acquired a write lock on O for the first time because of this write, push a copy of O's *current version* onto the top of the *version stack*. The copy pushed onto the *version stack* is called A's backup version. A's modification of O is then made to the *current version*.

When an action A that has a write lock on O commits:

If A is a topaction or if A's parent already has a write lock on O, discard A's backup version, which should be at the top of the *version stack*. (A's parent must already have its own backup version if it holds a write lock.) Otherwise, A's backup version becomes A's parent's backup version.

When an action A that has a write lock on O aborts:

Pop A's backup version from the top of the *version stack* into the *current version*. This will revert the current value of O to that in the popped backup version.

When an action A that has a read lock on O terminates, i.e., commits or aborts:

Do nothing to the *current version* and *version stack*.

Figure 2-3: Use and Management of Versions

The characteristics of the atomic subset of Argus are as follows:

1. *Objects shared by actions.*

Actions must share only atomic objects, with three exceptions. First, a shared atomic object may contain non-atomic objects in its **rep** if the atomic object is user-defined. Second, an ancestor may share a local non-atomic object with its descendants as long as the descendants do not modify the non-atomic object. A parent, for example, can write information into a local non-atomic object that its children subsequently read. A subaction is not allowed to modify an ancestor's local non-atomic objects

because such modifications are not recoverable should the subaction abort. Third, a guardian state may contain non-atomic objects as long as the objects are not modified by any of the handlers or the guardian's **background code**. However, without loss of generality, we will assume that all objects referenced by guardian variables are atomic.

2. *Variables*.

Variables do not provide the synchronization and recovery that are found in atomic objects. The above rule on sharing non-atomic objects, therefore, applies to variable assignment. In particular, a guardian's handlers and background code must not modify their guardian's variables and an action must not modify variables that are global to it.

3. *Nested topactions*.

Information that is passed from an ancestor to a nested topaction through non-atomic objects must not be derived from atomic objects in the guardian state. Otherwise, the nested topaction might not be serializable with the topaction of its parent. For example, a nested topaction T might modify an atomic object in the guardian state before its parent accesses the object, thus requiring T to be serialized before its parent's topaction. If T also reads information from a non-atomic object that reveals its parent's modification of some other atomic object in the guardian state, T then becomes non-serializable with its parent's topaction.

4. *Aborted actions*

No information derived from atomic objects may be returned to the caller when a handler activation aborts. Otherwise, serializability may be violated. For example, consider the following sequence of events:

1. Topaction A makes a handler call H
2. H reads an atomic object X
3. H aborts returning the value of X to A
4. Topaction T modifies atomic object X
5. T modifies atomic object Y
6. T commits
7. A reads Y

A and T will not be serializable since A saw the value of X before T's modification and the value of Y after T's modification.

2.4 Discussion

A guardian constitutes a logical node that has complete control over the data objects in its address space; objects (including variables) in one guardian can never be referenced directly from the handlers of other guardians. A handler activation, therefore, can be debugged without referring to its caller's local environment. Our debugging method takes advantage of this independence of a handler activation.

Chapter Three

The Method

There are two interactive phases in our method for debugging a computation. (Recall that a computation is a group of topactions.) In both phases, an action is regarded as a map from a *pre-* state to a *post-* state. The value of an object in the pre-state of an action A is the value that results from all modifications by actions serialized before A and all modifications by ancestors of A before A ran. The value of an object in the post-state of A is the value that results from updating the pre-A value with A's modifications, if any. A person who is debugging an action will be interested primarily in objects that are accessible from variables that are global to the action, as well as, the action's arguments, input, results and output.

An action has a *fault* if it maps a correct pre-state to an incorrect post-state. A *bug*, as used in this thesis, is a coding error in a program. Bugs are revealed by faults in actions. In our debugging system, the user, not the system, decides whether a pre- or post- state is correct and whether a piece of code is erroneous.

This chapter makes explicit our model of the debugging process (Section 3.1), gives a more precise definition of an *action tree* (Section 3.2), and explains the two phases of our debugging method in detail (Sections 3.3 and 3.4). We discuss only what the user sees of the method. Support for the method is covered in the rest of the thesis.

3.1 The Model

In our model, there are a number of entities involved in the debugging of a computation (see Figure 3-1). The *user* is the person who provides the intelligence required in debugging — the person who knows the expected behavior of

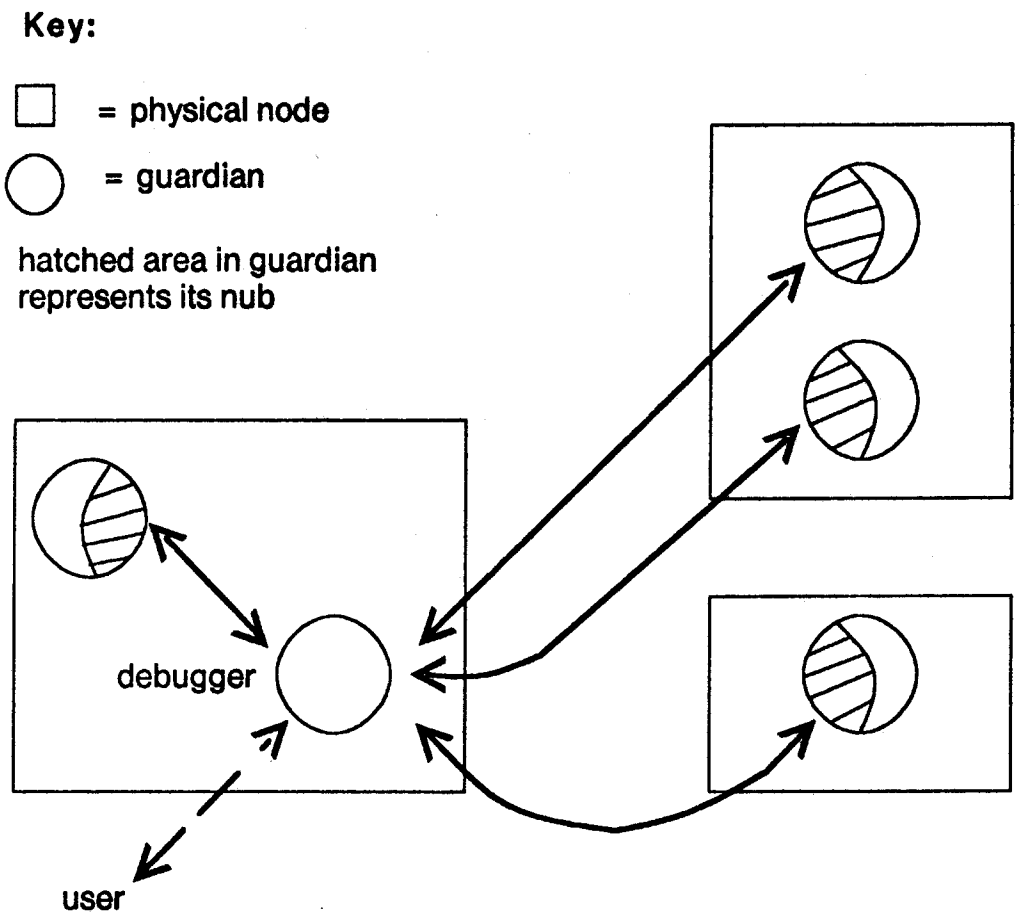


Figure 3-1: Participants of a Debugging Session

computations and who decides when a computation is misbehaving. The *debugging system* helps the user debug a program by providing, on request, information about computations.

The user's interface to the debugging system is a guardian called the *debugger*, which serves as the command interpreter with which the user interacts via a terminal. In addition to the debugger, the debugging system consists of *nubs*. There is one nub for each guardian in the program; the nub runs as a system process and accesses data and code of its guardian on behalf of the debugging system.

An action to be debugged could be either one that was generated by the user from the debugger, or one that was started from elsewhere but over which the user, through the debugger, has taken control. In either case, the action must have either terminated or been stopped by the user through the debugging system.

3.2 Action Trees

We define *action trees* in this section. In both phases of our debugging method, the user uses a computation's action trees and the serialization order of siblings as a guide to the internal working of the computation.

The action tree of an action A consists of the hierarchy of subactions that originated from A. The root of the tree is labelled by A's action identifier (*aid*); the interior nodes are labelled by the *aids* of A's descendant subactions. Each node of the tree contains information about the state of its action, i.e., whether it is *active*, *committed*, or *aborted*. Furthermore, if a node's action is a handler action, as opposed to an in-line subaction or topaction, the node will contain the name of the action's handler, as well as the identifier of the action's guardian.

We use the standard terms, *ancestor*, *sibling* and *descendant*, to refer to the relationships among actions in the tree. In this thesis, an action is its own ancestor and descendant. We will use *proper ancestor* (*proper descendant*) for those ancestors (descendants) that are not the action itself.

Figure 3-2 shows an example action tree. We use a circle to represent a committed action; a circle with an X over it for an action that aborted; and a box for an action that is still active. In addition, we double border the circle or box if the action is a topaction. In the figure, for example, A.2.1 aborted, A is active, and the rest of the actions committed. Furthermore, A.1 is a topaction; A, A.1.1, A.2.1, and A.2.2 are handler actions and the rest are in-line actions.

Much of the information about an action tree is encoded in action identifiers (*aids*) already maintained by Argus. *Aids* have the following properties:

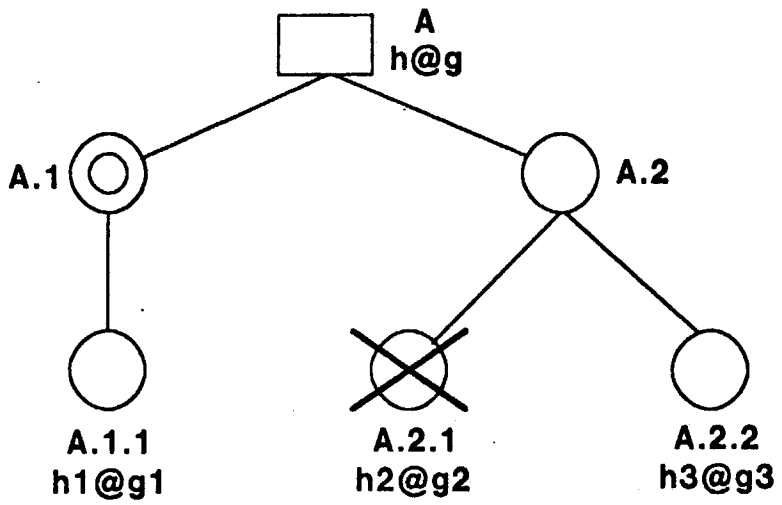


Figure 3-2: An Action Tree

1. An *aid* is globally unique.
2. An *aid* contains the identifier of the guardian where the action resides.
3. An *aid* contains the *aids* of all ancestors of its action.
4. We can tell from an *aid* whether or not it belongs to a topaction.

The other information needed for an action tree is maintained by the debugging system itself, and will be discussed in the next chapter.

Walking an action tree so that siblings are visited in their serialization order abstracts away internal concurrency and allows the user to follow and understand the working of an action in a sequential manner. Our debugging method exploits this "serial" walk quite heavily when narrowing a bug to a subaction, and when recreating a subaction's complete history.

3.3 Finding the Culprit Action

The objective of the first phase (called *Phase One*) is for the user to narrow the source of a fault in an action to as young a subaction in the action's tree as possible. (An action A is *younger* than another action B if A is a proper descendant of B.) We call this subaction the *culprit* action.

Assuming that the fault was manifested in action A, i.e., A noticed an inconsistency or did something unexpected because of an inconsistency, the first thing that the user has to do is determine whether the problem is within A. The user, through knowledge that is outside of the debugging system, decides which objects in A's pre-state might have been involved with the fault. He or she then examines the values of these objects as they were in the pre-state of A. If these *pre-A values* are correct, i.e., they do not explain why A misbehaved, then A's behavior must have violated its specification — the fault lies within A. Otherwise, the fault is either within some action serialized before A or with some input to A.

Once the fault is determined to be within a particular action A, the user next attempts to narrow the problem to one of A's children. He or she examines the child actions, looking for the first (in serialization order) of these to have an incorrect pre-state or to map a correct pre-state to an incorrect post-state. If the user is not able to find a child that has an incorrect pre-state or that implements an incorrect map, A then would be the youngest subaction to which the user can attribute the fault. If an action that has an incorrect pre-state is found, A must have erred in giving this pre-state to the child, assuming of course that it is not an inappropriate input that made the pre-state incorrect. If a child is found to map a correct pre-state to an incorrect post-state, the user has successfully narrowed the fault to that child. He or she then focuses on this child and tries to pin the fault to a grandchild, and so on. Details of the algorithm are provided in Figure 3-3. The algorithm is heavily dependent on the user:

1. to examine the "right" objects in the pre- and post- states of each of

Figure 3-3: Algorithm for "Top-down" Isolation of the Culprit Action

1. Let A be the action that has been determined to be faulty.
2. Get the action tree of A. Assume that A has children A.1, A.2, ..., A.n and that A.i is serialized before A.j for i < j.
3. For i := 1 to n do
    a. Decide which objects in the pre-state of A.i might shed light on the noticed misbehavior.
    b. Examine the pre-A.i values of these selected objects. If the pre-A.i values are inconsistent with what was expected, then the fault is either with A or with some input provided to A.i — return A or the erroneous input, identified with A.i's *aid*, as the result of this algorithm.
    c. If the pre-A.i values suggest a correct pre-A.i state, choose and examine some appropriate subset of the post-A.i state. As with the pre-state, the objects chosen from the post-state should have something to do with the fault being isolated.
    d. Decide whether A.i is correct in mapping the examined subset of pre-A.i state to the examined subset of post-A.i state, as far as the fault is concerned. If A.i is found to implement an incorrect map, repeat the algorithm on A.i.
4. If none of A's children is found to be faulty, then A is the youngest action to which the user can attribute the fault.

the children actions. (The "right" objects are those that are somehow involved with the fault currently being isolated.)

2. to decide whether an action is given a correct pre-state, as suggested by the examined objects, and

3. if an action's pre-state is correct, to decide whether the map from pre-state to post-state is consistent with behavior that is expected of

the action. As with the pre-state, the post-state is inferred from the objects that the user samples.

What our debugging system provides are the action's tree, the serialization order of siblings in the tree, the recreation of an action's environment, and values of objects in the pre- and post- states of actions.

The "top-down" algorithm described above is used when the user has ascertained that the fault is within an action and he or she would like to pin it down to the youngest descendant possible. When an action, for instance B, exposes an error that is not due to it — this implies that the action B was given an incorrect pre-state — the "bottom-up" algorithm, given in Figure 3-4, would be more applicable. In the "bottom-up" algorithm, the user searches "upward" from the noticing action B for the youngest ancestor that was given a correct pre-state. Because this correct pre-state led to the fault noticed in B, the ancestor must be the youngest within which the noticed fault can be explained. However, this ancestor, call it A_B, is not necessarily the youngest action to which the fault can be attributed. The fault could perhaps be due to some descendant of A_B that is serialized before B. The user has to apply the "top-down" algorithm on A_B to find this culprit subaction.

It is possible that there is no ancestor of B to which the fault can be pinned. The problem then must be with the most recently committed topaction that was serialized before B and that had a correct pre-state. Figure 3-4 gives details of a strategy for finding this faulty topaction.

3.3.1 Example

This subsection sketches the Phase One debugging of a computation in a simple mail system. Let us assume that Jim suspects that there is something wrong with the mail system: he asked for his mail and was told that there was none when he knows (through a source other than the mail system) that John has sent him a piece of mail.

Figure 3-4: Algorithm for "Bottom-up" Isolation of the Culprit Action

Preamble: If an incorrect input is isolated, the user has found the "bug". In this algorithm, we assume that when a pre-state is incorrect, it is not because of an incorrect input.

1. Let B be the action whose pre-state is incorrect, i.e., holds some unexpected values.
2. Check to see whether the pre-state of B's parent is correct. The user does this by examining the pre-action values of judiciously selected objects.
3. If B's parent has a correct pre-state, then the noticed fault must be caused by either B's parent or one of B's siblings serialized before B. Use the "top-down" algorithm (Figure 3-3) on B's parent to find the culprit subaction.
4. If B's parent was given an incorrect pre-state, check the next older ancestor, and so on, until one that has a correct pre-state is found. This ancestor is the youngest that encompasses the noticed fault. Apply the "top-down" algorithm on this faulty ancestor to find the culprit subaction.
5. It is possible that the fault is not with the topaction of which B is a part. The user, when out of ancestors to test, should switch to testing committed topactions that are serialized before B. But how are candidate topactions to be found? One strategy would be for the user to examine "interesting" objects and, from history saved there for support of pre- and post- action views (to be explained in the next chapter), deduce the topactions that both modified the objects and are serialized before B. "Interesting" objects are those that are in the state of the program's guardians and that might have something to do with the fault at hand. The topaction that caused the fault should be among the topactions that modified these interesting objects. So, go through these topactions in reverse serialization order, sampling pre- and post-action states, to find one that violated expected behavior and thus contributed to the fault noticed by B. Once such a topaction is found, apply the "top-down" algorithm to it.

To debug the program, Jim (or more probably, the person who is maintaining the mail system) might begin with the action that represents his call on the mail system. The first thing that Jim has to do is find the action's *aid*. We assume that there is some facility, perhaps like the process status queries available in most operating systems, that would help him do this. On finding the action's *aid*, Jim then uses it to get the action's tree, depicted in Figure 3-5, and a serialization order from the debugging system to guide him through the debugging process. Assuming that A.1 is serialized before A.2, the tree shows that Jim read his mail by calling the *read_mail* handler of the *mailer* guardian. *Read_mail* in turn called the *where_is* handler of the *mailbox locator* guardian, and then the *read* handler of the *MIT post office* guardian.

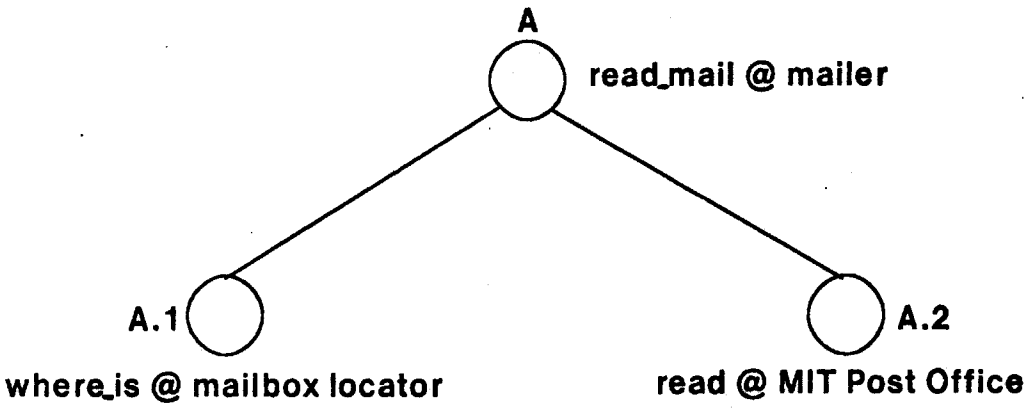


Figure 3-5: Action Tree of an Example Computation

We assume that read_mail works as follows. When provided with a user name, it first calls the *where_is* handler of the *mailbox locator* to find the <post office, box number> that is the user's mailbox. It then calls the *read* handler of the *post office* guardian where the user's mailbox is located to get the user's mail. The retrieved mail is then returned as response.

Following the "top-down" algorithm of Phase One, Jim takes the following steps to isolate the culprit action:

1. Check to make sure that the argument provided *read_mail* was indeed "Jim". Otherwise, the pre-state to the action was incorrect — the action was provided with a wrong name and had nothing to do with the perceived problem.

2. Check to make sure that "Jim" was the user name that was passed along as argument to the *mailbox locator*. Otherwise, the bug lies with the *read_mail* handler of the *mailer* — it forwarded to the *mailbox locator* a user name that was not what was given it.

3. Check the pre-A.1 values of appropriate objects in the state of the *mailbox locator* to make sure that the result returned by the *where_is* handler was correct. (A.1 is the *aid* of the *where_is* handler activation.) Otherwise, the bug is in the *where_is* handler of the *mailbox locator*.

4. Check to make sure that the results of the *where_is* handler call correspond to the *post office* guardian that was contacted by *read_mail* and the argument that was sent in the call to the *post office* guardian. If the results do not correspond, the bug is in the *read_mail* handler of the *mailer* — it either contacted the wrong *post office* guardian or gave the correct *post office* guardian the wrong box number to search for mail.

5. Check to make sure that no mail was returned by the *read* handler to *read_mail*. Otherwise the problem is with *read_mail* — it lost the mail that it received from the *MIT post office*.

6. Finally, find in the pre-A.2 state of the *MIT Post Office* the object that represented Jim's P.O. Box. (A.2 is the *aid* of the *read* handler activation.) Check to make sure that the pre-A.2 state of this object is

indeed empty of mail. If not empty, the bug is in the *read* handler of the *post office* guardian. If empty, the culprit action is not the *read_mail* handler call or any of its subactions.

If the fault is not in the *read_mail* handler call or any of its subactions, it may be in the action that corresponds to John's sending of mail to Jim. Jim should debug this action next, and so on.

3.4 Retracing an Action

The user moves into the second phase (called *Phase Two*) if details about the internal working of a faulty action are needed to pin-point the bug in the code. In Phase Two, our debugging system re-executes code to *retrace* the history of an action. Data that is saved for Phase One is used to simulate the original conditions. For example, when an action reads an atomic object in a retrace, it reads its pre-action value of the object from the original execution.

Our debugging system uses a single thread of control when re-executing an action. This single thread of control will retrace a serial execution that is equivalent to the action's original execution. Terminated siblings are retraced in their serialization order and before any sibling that might still be active, i.e., those that are stopped but not terminated. Active siblings, on the other hand, are retraced in some arbitrary order; the order does not matter because an active sibling cannot possibly have already seen the effects of another active sibling, assuming of course that user-defined atomic objects are implemented correctly. To isolate the bug, the user uses the usual break-and-examine tools on the retrace's single thread of control; the user breakpoints and single-steps the retrace and examines current views of objects at the points where a retrace is stopped. Phase Two debugging, therefore, is very much like sequential debugging.

We stress that our debugging system will create an execution that is equivalent to the action's original execution in a retrace. A handler activation, for example, will

not return results or call subactions that it did not return or call in the original computation. For an action that is still active, our debugging system will not retrace (noticeably) past the point where the action is stopped in the original computation. We also emphasize that retracing does not interfere with the use of objects by other actions. It does not, for example, modify the state of the system nor does it require that we lock objects from use while the user retraces an action.[3]

All terminal output that results from a retrace is directed to the user's controlling terminal. This helps the person who is debugging to have a better feel for the progress of the retrace. Input, needed in a retrace, is taken from history, so when an action reads from the terminal it will be given exactly the same input it originally received. When an action reads an input value from history, the input is also sent to the user's terminal to keep the user updated on the progress of the retrace.

When retracing an action, the user has the option of skipping the re-execution of topactions nested within the action, and the re-execution of subactions and handler calls that the action makes, without affecting the rest of the retrace. The user may want to skip a nested topaction, subaction, or handler call if he or she feels that the skipped action would shed no light on the bug that is being tracked. If a handler call is retraced, our debugging system will re-execute not only the handler action but also the *encode* of the arguments at the sending guardian and the corresponding *decode* at the receiving guardian. This enables the user to debug the *encode* and *decode* operations of a type.

3.5 Discussion

Handler activation bias.

As pointed out in the previous chapter, a handler activation does not have any

---

[3] If the action that the user is debugging is not terminated, it may have active locks on objects. Other actions may then be prevented from accessing these objects. This is not the same as locking objects specifically to permit retracing the action.

stack variables that need to be set up before it can run; its environment consists solely of its guardian's variables. The environment of an in-line action, on the other hand, consists not only of the guardian's variables but also the local variables of its local ancestors. In our debugging system, we choose not to save environments. Retracing, then, will always have to begin with a handler action. Also, when the local environment of an in-line subaction or a nested topaction is needed in Phase One, the user has to recreate it by retracing from the youngest ancestor that is a handler action.[4] Note that our debugging system can avoid saving local environments precisely because the intermediate states of an action can be recreated faithfully.

Our debugging system is biased toward handler activations because it is much simpler and cheaper than the alternative of saving local environments, and also because we expect the structure of computations in Argus to consist mainly of handler calls.

Display operations.

We assume that all built-in and user-defined types have a *display* operation that presents the abstract states of the type's objects in some relatively understandable form. A *display* operation, therefore, serves as an abstraction function [Guttag et al. 78] and shields the users of a type from having to know details about the type's implementation. Our debugging system uses a type's *display* operation to present to the user pre- and post- action values of the type's objects, as well as values in the intermediate states of a retrace. *Display* operations, therefore, must not modify any object. In addition, *display* results must be transmissible because the user is connected to a command interpreter guardian (the *debugger*) that is separate from the program's guardians. (An object is transmissible if its type provides *encode* and *decode* operations.)

Our debugging system assumes that *display* operations are implemented correctly. Debugging *display* operations is a task that more properly belongs to

---

[4] We will briefly discuss the debugging of topactions in a guardian's **background code** in Chapter 5.

sequential debugging of an abstract data type's implementation and, therefore, is beyond the scope of this thesis.

Pre-action values.

Our debugging system gives pre-action values of objects that an action did not access, in addition to pre-action values of objects that the action accessed. The former pre-values are those that the action would have read if it had accessed the objects, given the action's place in the serialization order. These pre-values are sometimes useful for debugging.

Aborted actions.

Aborted actions may be of interest to the person who is debugging. He or she, for example, may want to know why calls to a particular handler abort as often as they do. Our debugging system treats aborted actions much like committed ones. The user can ask for the values of atomic objects in the pre-state of an aborted action, and he or she can retrace an aborted action, as well.

Our debugging system also provides the values of atomic objects at the point just before an action aborted. These "pseudo-post" values may help the user deduce how far an aborted action got and why it aborted, without retracing the action.

Chapter Four

Support for Isolating the Culprit Action

In this chapter, we present the implementation of the support for Phase One. In this phase, the user depends on the debugging system for the action trees, serialization order, and information about pre- and post- action states that are needed to isolate the culprit action of a faulty computation. Pre-action values of objects are also used by the debugging system to support the retrace of an action (see Chapter Five).

All references to relationships between actions in this chapter are made with respect to a special system-wide *universal tree*. The root of this tree is a (fictitious) action **U** that never terminates. All topactions in the system, including nested topactions, are children of **U** in the universal tree, and therefore are siblings of each other. Subactions make up the other interior nodes of the universal tree. A subaction has the same parent in the universal tree as it does in an action tree. Note that in contrast to the universal tree, a nested topaction is placed as a child of its calling action in an action tree. This mirrors the calling structure of a computation and is what the user needs to figure out the context of a nested topaction. The universal tree, on the other hand, gives a truer picture of the serialization of actions and is easier to use for deducing action views of the system state.

We begin in Section 4.1 with a discussion of the main design principles by which the implementation of our debugging system is guided.

In Section 4.3, we give a method for computing a serialization order. The method uses an ordering of the termination of actions that is derived from Lamport clocks [Lamport 78]. We note that the serialization order is also used to support the retracing of actions and to deduce pre- and post- action values of atomic objects.

In Sections 4.4 and 4.5, we discuss pre- and post- action values of built-in atomic objects. (Action views of user-defined atomic objects are discussed in Chapter Six.) We introduce versions into Argus and show how they can contribute, in cooperation with the serialization order and the universal tree, to compute action views of built-in atomic objects.

In Section 4.6, we discuss the implementation and use of the universal tree. Action trees, needed by the user, are also derived from this universal tree.

Arguments, results, input, and output are the other objects that are of interest to the person who is debugging. In Section 4.7, we describe how these are saved and provided to the user.

Finally, we discuss two practical issues. All the sections we have mentioned thus far ignore crashes and assume that saved history is never discarded. In Section 4.9, we study the effects of crashes on the results of these sections. In Section 4.10, we discuss how saved history can be reclaimed with minimal impact on the user's ability to debug actions.

4.1 Design Principles

There are two main principles by which our design choices are governed.

1. Debugging-related activities should add as little as possible to a computation's run-time.

The idea is to do just enough work and nothing more while a program executes. In particular, as much as possible of the debugging-related activities should be deferred until debugging time or when there are spare processor cycles. The run-time of a computation should not be unnecessarily penalized just because of an anticipation, which may well turn out to be false, that it will need to be debugged. For example, we will not make copies of object versions that are either unnecessary or that could be constructed at debugging time.

Our aim is to have data collection for debugging purposes be transparent enough to make it feasible to monitor all activities in the system all the time. So, when a bug surfaces, the user will have the information to track it; there will be no opportunities that are lost just because "debugging was not turned on".

2. Normal cases should be favored.

Wherever possible, we will favor cases that occur frequently over those that are less frequent. For example, we expect that commits will be more frequent than aborts, and reads will be more frequent than writes. Therefore, we save versions of atomic objects on writes and aborts instead of reads and commits.

4.2 Terminology

We collect together in this section some terms and notations that will be used in the rest of this chapter and thesis. All of them have to do with how actions are related in the universal tree.

Definition:

> We say that an action A *committed up to* an ancestor P if all ancestors of A (including A) that are proper descendants of P committed.

Definition:

> We say that an action A *committed relative to* an action B if A committed up to the least common ancestor of A and B.

Definition:

> We say that A and B are *ancestor-related* if either A is an ancestor of B or B is an ancestor of A.

Notation:

We use *LCA(A,B)* to denote the action that is the least common ancestor of A and B, where A and B are not ancestor-related. Note that if A and B are not descendants of the same topaction, then their LCA is U, the root of the universal tree.

Notation:

For two actions, A and B, that are not ancestor-related, we use $GA_A(B)$ (read "greatest ancestor of A that is not an ancestor of B") to denote the child of LCA(A,B) that is A's ancestor.

4.3 Termination Numbers

Termination numbers are unique timestamps that totally order the termination of all actions in our system. We generate these numbers with counters that are maintained as Lamport clocks [Lamport 78]. In this section, we explain how termination numbers are assigned to actions and how they are used to give a serialization order. We assume that crashes do not happen; the modifications for node crashes are discussed in Section 4.9.

Using Lamport's algorithm, we maintain and assign termination numbers as follows:

1. Each guardian has a *termination counter*. A termination counter is composed of two halves: the high order half and the low order half. A *termination number* is obtained from a termination counter by prefixing the digits contained in the low order half with those of the high order half. So, if the high order half has N digits and the low order half M digits, the resulting termination number will have N + M digits, with the value in the counter's high order half occupying the more significant places.
2. When a guardian is created, its termination counter's high order half is assigned an arbitrary value. Zero, for instance, is a perfectly good number to use. The counter's low order half, however, is given the guardian's identifier as value.[5] The value in the low order half will

[5] Guardian identifiers are unique in Argus.

never be changed. It is used to ensure that termination numbers are globally unique, even among non-communicating guardians.

3. When an action terminates, it is given the termination number derived from its guardian's counter. The high order half of the counter is then incremented. The "read-and-increment" of the counter's high order half is executed indivisibly.

   A topaction is given its termination number at the start of phase one (the *preparing* phase) of the two-phase commit protocol.

4. Every message sent from a guardian carries the termination number contained in the guardian's counter.

5. A guardian on receiving a message will check the termination number in its local counter against that carried in the message; if the local value is smaller, the guardian will advance the high order half of its counter to the value that is one greater than that in the high order half of the message's termination number.

From now on, we shall use the notation A# to refer to the termination number of an action A.

Termination numbers have been shown to produce a valid serialization order for a non-nested action system that uses two-phase locking [Bernstein & Goodman 81, Bernstein & Goodman 83]. In this section, we extend the result to nested actions. We argue that for two committed actions A and B that are either topactions or sibling subactions, if A must be serialized before B, then A# < B#.

Theorem 1:
For two committed topactions A and B, if A is serialized before B, then A# < B#.

Proof:

There are three possible reasons why A is serialized before B:

1. A and B are serial because of program flow
2. A and B accessed some common atomic object X in conflicting modes, either directly or via descendants, with A using X before B
3. There is a chain of topactions T_1, ..., T_n such that A is serialized before T_1, T_1 before T_2, ..., T_n before B.

We prove the theorem for each of these cases.

Case 1: There are two subcases that are predicated on whether A and B ran in the same guardian.

(a) If A and B ran in the same guardian:

1. A must have terminated before B, since A ran serially before B.
2. Termination numbers given by a guardian are monotonically increasing.

$\therefore$ A# < B#.

(b) If A and B did not run in the same guardian:

1. B must be a nested topaction, since the program flow that created A also created B in a different guardian.
2. A must have terminated before the first handler call in the chain of handler calls that eventually created B.
3. The termination number of B's guardian must be greater than A# when B is created, since call messages carry termination numbers.

$\therefore$ A# < B#.

Case 2: There are two subcases that are predicated on whether A and B used X directly.

(a) If A and B used X directly:

1. A and B must have run in the same guardian, since they directly accessed the same data object.
2. A must have terminated before B, since B could not have accessed X until A released its lock and locks are released only when an action terminates.
3. The termination numbers given by a guardian are monotonically increasing.

$\therefore$ A# < B#.

(b) If A and B used X indirectly via descendants A_d and B_d that committed up to A and B respectively (note that one of these descendants could be A or B itself):

1. A was assigned a termination number at the beginning of the *preparing* phase.
2. A# was sent to all participant guardians in *prepare* messages.
3. A's lock on X was released only after receipt of the *prepare* message with A#.
4. Therefore, $A\# < B_d\#$, since B_d could have secured a lock on X only after A released its lock.
5. Also, $B_d\# \leq B\#$, since termination numbers are sent in reply messages and a parent does not commit before a child.

$\therefore$ $A\# < B\#$

Case 3:

By cases 1 and 2 above, $A\# < T_1\# < ... < T_n\# < B\#$.

$\therefore$ $A\# < B\#$.

∎

Theorem 2:
For two committed subactions, A and B, that are siblings, if A is serialized before B, then $A\# < B\#$.

Proof:

The proofs for Cases 1(a), 2(a) and 3 carry over directly from Theorem 1. There is no Case 1(b) in this theorem because A and B must have run in the same guardian since they are siblings. So, we just have to prove Case 2(b) to prove this theorem. (This is the case where A and B used X in conflicting modes via descendants A_d and B_d that committed up to A and B respectively.) The proof for Case 2(b) in this theorem differs from that in Theorem 1 because a subaction does not go through the two phase commit protocol when it commits.

Case 2(b):

1. A must have terminated before B_d, since B_d could not have accessed X until A released its lock and locks are released only when an action terminates.
2. If B_d is not B, B_d must have terminated before B, since an action terminates after all descendants that commit up to it.
3. Therefore, A must have terminated before B.
4. A and B must run in the same guardian, since they are siblings.
5. The termination numbers given by a guardian are monotonically increasing.

$\therefore$ A# < B#

∎

Notice that just because A is serialized before B does not imply that the termination numbers of B's descendants are greater than A#. (A and B might have run concurrently.) To extend the serialization order to actions A and B that are not siblings but that commit up to their LCA, it does not work to simply compare their termination numbers. Instead, we have to compare the termination numbers of their ancestors that are siblings.

Therefore, we have the following definition of serialization order:

Definition:

> For two actions M and R that are not ancestor-related and that commit up to their LCA, M is *serialized before* R if the termination number of $GA_M(R)$, i.e., the child of LCA(M,R) that is M's ancestor, is less than $GA_R(M)$#.

Recall that if M and R are not descendants of the same topaction, their LCA is the root of the universal tree. Using the definition, we have to compare the termination numbers of their topactions to find the serialization order of M and R in this case. We also note that the serialization order is defined only for actions that commit up to their LCA.

4.4 A Strawman Scheme for Action Views of Atomic Objects

In this section and the next, we discuss two schemes for saving and using versions to compute the pre- and post- action values of built-in atomic objects. In both of these schemes, we aim to be able to compute pre- and post- values of all actions, including those that abort, at all built-in atomic objects, i.e.,

1. objects that an action read or modified, either directly or through a descendant that committed up to it, as well as
2. objects that an action did not use at all.

The pre-value that our schemes compute for an object that an action did not use must be consistent with values that the action saw at other objects. In other words, the pre-value must be one that the action could have seen if it had read the object. These pre-values are sometimes useful for debugging, e.g., the user may want to know the value of an object that an action was supposed to read but did not. More importantly, our method for computing pre- and post- action values of user-defined atomic objects (to be presented in Chapter Six) depends on the availability of these pre-values.

In this section, we present a straightforward but inefficient scheme for supporting pre- and post- action values of built-in atomic objects. We improve on this simple scheme in the next section.

The information saving aspect of the simple scheme is as follows:

1. Whenever an action A acquires a write lock, save a pointer to a copy of the current version as the pre-A value. (This rule does not apply when a write lock is inherited from a descendant.[6])
2. Whenever an action A acquires a read lock, save a pointer to a copy of the current version as the pre-A value. (Again, this rule does not apply when the lock is inherited from a descendant.)

---

[6]Note the distinction between "acquires" and "inherits". An action *acquires* a write lock when it modifies an object without possessing a write lock prior to the modification. An action *inherits* a lock on an object when a descendant that possesses the lock commits up to it.

3. Whenever an action A terminates, i.e., commits or aborts, save a pointer to a copy of the current version for all objects at which A has a read or write lock, regardless of whether the lock was directly acquired or inherited. Label this saved version as "post-A".

Using the information collected by the algorithm to deduce pre- and post-action values for objects that an action used is straightforward. If an action has a recorded pre-value, then that is its pre-value. However, an action, say A, that did not use the object directly will not have a recorded pre-value. Then, the pre-A value is given by the first recorded pre-value that belongs to a descendant that committed up to A. The post-value of A is A's recorded post-value if A committed; if A aborted, its post-value is its pre-value. Notice that in the case where A aborted, A's recorded post-value is really the *pseudo-post* value of A, i.e., the value that held just before A aborted. Since the pseudo-post value may hold modifications by A, it is not A's post-value. Pseudo-post values are useful as a hint of where an action reached before it aborted.

To deduce pre- and post- values for objects that an action, say A, did not access, we will need to use the serialization order; the post-value of the latest writer serialized before A is typically A's pre-value. We defer discussion of the details to the next section.

The simple scheme of this section is inefficient. It does more copying and saves more information than is needed. The following are examples of some of the unnecessary work:

1. Instead of saving a fresh copy of the current version when a write lock is acquired, the version created by the run-time system for recovery could have been used.
2. A write lock acquired by a subaction may be propagated through many ancestors, thus causing many post values to be saved. However, only one post value is needed if the ancestors did not write the object.
3. Suppose some topaction B modifies an atomic object X that is then immediately read by another topaction A. Then the pre-value for A is equal to the post-value for B; it is not necessary to save both post-B and pre-A.

4. The pre-value of an action that read but did not modify an object need not always be saved. We can sometimes use the serialization order to compute this pre-value.

Below, we present an optimized scheme that uses recovery versions and avoids saving duplicate information. Post-values for actions are stored only when necessary, i.e., just before the next time the atomic object is modified. Pre-values for read-only actions are usually not saved at all; usually, we use the post-value of the latest writer serialized before the reading action.

4.5 An Optimized Scheme for Action Views of Atomic Objects

This section begins by enumerating and justifying the versions of atomic objects that we save for the optimized scheme. Next, it presents the algorithm that uses the versions to calculate pre- and post- action values. Finally, it explains how the algorithm can search versions quickly for the appropriate pre- or post- value.

4.5.1 Saving Versions of Atomic Objects

We present a set of rules that determine the versions of a built-in atomic object to save. In the rules, the versions are saved in a special data structure, called a *pre-post* log in the internal representation of the object.

When a built-in atomic object is created in Argus, the system creates a special nested topaction that writes the initial value into the object and then commits immediately. We initialize the pre-post log when the object is created so that we can distinguish actions that ran before the creation from actions that ran after the creation.

Saving Rule 0:

When a built-in atomic object is created

- save a pointer to a copy of the initial value
- tag the newly inserted entry in the pre-post log as "Pre-T" where T is the nested topaction that writes the initial value.

We label the initial version in the log as "Pre-T" and not "Post-T" because we save pre-action values in our scheme and not post-action versions. "Pre-T", therefore, would fit more easily into our rules for using the saved versions. Confusing the initial version in the log as a "Pre-T" (when it is actually T's post-value) will not cause problems because T is not an action that the user will ever want to debug.

Saving Rule 1:

When an action A acquires a write lock,

- save a pointer to the recovery version created for A,
- label the newly inserted entry in the pre-post log as "Pre-A".

Note that the above rule applies only to the direct acquisition of locks and not to the case where locks are inherited. Also, we avoid copying by using recovery versions. Of course, now when an action aborts, the current version of the object has to be reset to a *copy* of the recovery version, since the recovery version itself is being used as saved history and must not be modified. But aborts are expected to be infrequent, so the need to copy should be rare.

We also need to tag the "pre-A" version created in Saving Rule 1 by the *aid* of A's last child to terminate before the write, if any. Otherwise, we will not be able to tell whether a child read A's modification. For example, if Saving Rule 1 is left as is, the pre-post log of X that is created by the following two sequences of events will be identical.

Sequence 1: A modifies X
A calls A.1
A.1 reads X
A.1 commits
A commits

Sequence 2: A calls A.1
A.1 reads X
A.1 commits
A modifies X
A commits

The debugging system, therefore, will not be able to determine the value A.1 read given just the history that is saved. So, we have the following amendment to Saving Rule 1.

Amendment to Saving Rule 1:

Tag the saved "Pre-A" entry with the *aid* of A's last child to terminate before the write, if any.

We will use "Pre-A, C" to denote a "Pre-A" entry that was created at a time when C was the last terminated child of A.

In contrast to pre-values, we will save post-values only when needed. In particular, we will not save the post-value when an action A commits, since the post-value continues to be available as the current version. Post-values need to be saved only when this current version is later modified. This modification can happen in two ways: when an ancestor B aborts or when an action C writes into the object after A.

Saving Rule 2:

When a write lock is discarded because an action B aborts,

- save a pointer to the current version,
- tag the entry in the log as "Post-B".

Again, we note that a copy does not have to be made. Instead, we use the version about to be discarded. Also, Saving Rule 2 saves "Post-B" regardless of whether B has a descendant that needs the saved version for a post-value. This is because the saved "Post-B" value is the value that holds just before B aborts and is useful to the user when debugging B, for a sense of where B reached before it aborted.

In the case of an action C that writes into an atomic object after A, we need not save a post-A value explicitly if C is not an ancestor of A. This is because a pre-C value will be saved by Saving Rule 1 and this value is post-A, as well. However, if C is an ancestor, C does not acquire a write lock, but merely uses the lock it inherits when A committed up to it. Therefore, we must save a post-value for A at this point.

Saving Rule 3:

When an action C modifies an atomic object, if the following conditions are true

1. C has a write lock before this write
2. C has child actions before this write, and
3. the most recent entry in the log is not tagged "After-T"

then,

- make a copy of the current version and save a pointer to the copy,
- tag the entry as "After-T".

We also use "After" versions to provide for pre-values of actions that read but did not modify an atomic object. As discussed before, a read-only action R typically reads the modification of the most recent action that is serialized before it. This value is the pre-value stored in the pre-post log of the action with the least termination number that is greater than R#, or the current version if there is no such pre-value. However, there is a problem in the case of a parent that acquires a write lock, calls a read-only child, and then modifies the object after that child commits. The pre-value of the child includes modifications made by the parent before the child is called, but not those made after the call. Therefore, we must record this intermediate value of the parent just before the parent modifies the object after the child commits. In fact, we save the intermediate value even if the child aborts; this is because we support pre-values for aborted actions as well. "After" versions provide these intermediate values.

At first glance, Saving Rule 3 seems expensive because of the number of copies that might have to be made. However, on closer examination, we find that a copy is made only if

1. an action modifies an atomic object, calls a child action, and then later modifies the same object again, or
2. a descendant that commits up to action A modifies an atomic object and then A modifies the same object.

These conditions are rather unlikely, especially since we expect most modifications in Argus to be made by leaf actions, i.e., actions that do not have subactions.

We summarize our rules for saving versions in Figure 4-1 for easier reference. We give an example of the history that is saved in the pre-post log of an atomic array of integers in Figure 4-2.

Note that we do not save information about reads explicitly. So, we cannot tell whether or not an action actually read an atomic object. In the case where an action read an atomic object X, our optimized scheme will present the value that the action actually saw at X. In the case where an action did not read X, our optimized scheme will present the value that the action might have seen at X: the value displayed will be consistent with values that the action actually saw at other atomic objects.

In the rest of the chapter, we say that a log entry *belongs to* an action A if the entry is either "Pre-A", "Post-A", or "After-C", where C is a child of A.

4.5.2 Using the Saved Versions

This subsection presents a set of rules for deducing pre- and post- action values, given the information we save in the pre-post logs. The rules are dependent on the meaning of serialization order and action nesting. For example, we depend on the following facts:

1. The pre-value of an action B is the post value of the sibling action A that immediately precedes it in the serialization order, assuming that no ancestor of A modifies the object between the calls to A and B.
2. The pre-value of an action A equals the pre-value of the first descendant D that committed up to A, if D accesses the object before A.
3. The post-value of A equals the post-value of the last descendant D' that committed up to A, if A does not modify the object after D' terminates.

The rules also use the fact that information is pushed on a pre-post log in the order

Figure 4-1: Rules for Saving Versions of Atomic Objects

We add a new data structure, called a *pre-post log*, into the internal representation of each built-in atomic object. In the following rules, all history that is saved about an atomic object's state changes is kept in its pre-post log.

Saving Rule 0:
When a built-in atomic object is created

- save a pointer to a copy of the initial value
- tag the newly inserted entry in the pre-post log as "Pre-T" where T is the nested topaction that writes the initial value.

Saving Rule 1:
When an action A acquires a write lock

- save a pointer to the recovery version created for A.
- label the newly inserted entry in the pre-post log as "Pre-A",
- tag this "pre-A" entry with the *aid* of A's last child to terminate before this write, if any.

Saving Rule 2:
When a write lock is discarded because an action B aborts,

- save a pointer to the current version,
- tag the entry in the log as "Post-B".

Saving Rule 3:
When an action C modifies an atomic object, if the following conditions are true

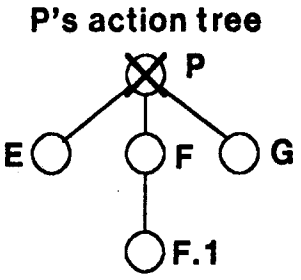
1. C has a write lock before this write
2. C has child actions before this write, and
3. the most recent entry in the log is not tagged "After-T"

then,

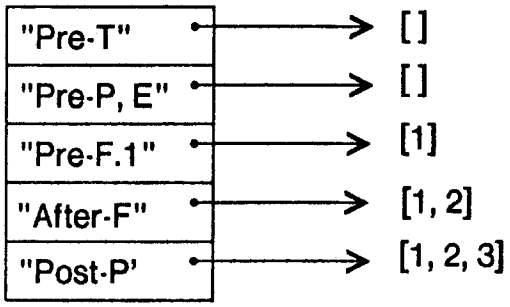
- make a copy of the current version and save a pointer to the copy,
- tag the entry as "After-T".

The sequence of events

P calls E
    E commits
P appends 1 to the array, which is initially empty
P calls F
    F calls F.1
        F.1 appends 2 to the array
        F.1 commits
    F commits
P appends 3 to the array
P calls G
    G commits
P aborts



The array's pre-post log after the sequence of events



Note: T is the nested topaction that created the atomic array

Figure 4-2: An Example of the History Saved at a Pre-Post Log

that the object is used, so the order of versions in a pre-post log is consistent with the serialization order of the versions' actions.

4.5.2.1 Pre-Action Values

Suppose we want the pre-A value of an atomic object X. Then, there are the following cases to consider:

1. A modified X, either directly or through a descendant that committed up to it.
2. A did not modify X but there is an action B that modified X and is serialized before A.
3. A and actions that are serialized before A did not modify X but there is an action B such that B modified X and $GA_B(A)\# < GA_A(B)\#$. (In case (2) above, A and B both committed up to their LCA. In this case, B committed up to LCA(B,A) but A did not.)
4. No action modified X before A was called.

Below, we explain how to compute pre-A for each of these cases.

(1) If A modified X, either directly or indirectly, then its pre-state is stored via Saving Rule 1 as

1. "Pre-A" if A modified X before any of the descendants that committed up to it,
2. or "Pre-D" otherwise, where D is the first descendant that modified X and committed up to A.

So,

Pre Rule 1:

If an entry tagged "Pre-A" exists in the pre-post log,

pre-A = version in the "Pre-A" entry.

Pre Rule 2:

Else if "Pre-D" exists where D is a descendant that committed up to A,

pre-A = version in the first such "Pre-D" entry in the log.

(2) This is the case where A did not modify X but some action B that is serialized before A did. Note that in this case B and A both committed up to their LCA. Also, B and A can be descendants of different topactions. (In this situation, their LCA is the root of the universal tree.)

There are two subcases to consider: either

(a) A read (or could have read) the modification by the action B' that is the latest among the actions serialized before A to modify X, or

(b) A read (or could have read) the modification that was made by some ancestor of A after B' committed.

Subcase (a):

Since B' modified X directly, there must be a "Pre-B'" (by Saving Rule 1) or an "After-T" entry, where T is a child of B' (by Saving Rule 3). The value that was read by A, therefore, is given by the entry after the latest "Pre-" or "After-" entry in the log that belongs to an action serialized before A.

Subcase (b):

This subcase applies only if an ancestor modified X after $GA_{B'}(A)$ committed but before A ran. If the subcase applies, there will be one of the following kinds of entries in the log after all the entries that belong to B':

1. a "Pre-" entry of an ancestor of A that is not tagged with a child's *aid*
2. a "Pre-P, C" entry, where P is an ancestor of A and C is P's child such that C itself is not an ancestor of A and $C\# < GA_A(C)\#$.
3. an "After-T" entry, where T is a child of an ancestor of A such that T itself is not an ancestor of A and $T\# < GA_A(T)\#$.

The entry after the latest such "Pre-" or "After-" is the value that A read.

We present subcases (a) and (b) as one algorithm in the following Pre Rule 3.

Pre Rule 3:

Else if there is an action B such that

1. B modified X,
2. A and B are not ancestor-related,
3. A and B committed up to their LCA, and
4. $GA_B(A)\# < GA_A(B)\#$

find the latest entry E in the log that either

1. belongs to such a B, or
2. is a "Pre-" entry of an ancestor of A that is not tagged with a child's *aid*, or
3. a "Pre-P, C" entry, where P is an ancestor of A and C is P's child such that C itself is not an ancestor of A and $C\# < GA_A(C)\#$, or
4. an "After-T" entry, where T is a child of an ancestor of A such that T itself is not an ancestor of A and $T\# < GA_A(T)\#$.

pre-A = version after E in the log, or the current version if E is the last entry in the log.

Pre Rule 3 does not apply if A did not commit up to Q, the youngest ancestor that modified X either directly or via a descendant, because an aborted descendant's pre-value is not necessarily tied to that of an ancestor's. For example, consider the following scenario:

Topaction Q calls a child action A
    A reads an atomic object X
    A aborts
Topaction M modifies X
M commits
Q modifies X

When Q modifies X, a "Pre-Q, A" is inserted into the log by Saving Rule 1. If we use Pre Rule 3 without requiring that A commits up to Q, we will choose the "Pre-Q" version as the pre-A value. This is wrong because "Pre-Q" reflects M's modification, whereas A read X before M modified it. This leads us to the next case.

(3) This case is illustrated in Figure 4-3. In this case, A did not modify X, either directly or indirectly. Furthermore, P (the youngest aborted ancestor of A) has no descendant that is serialized before A and that modified X. There is, however, an action B that modified X such that $GA_B(A)\# < GA_A(B)\#$.

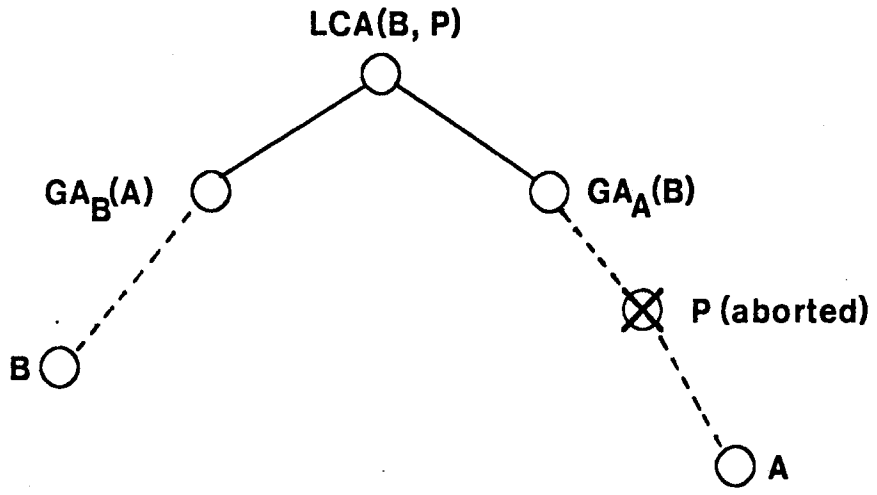


Figure 4-3: Action Tree to Illustrate Pre-Values of Aborted Actions

There are three subcases to consider:

(a) A read (or could have read) the post-value of such a B.

(b) A read (or could have read) a modification that was made by some ancestor of A.

(c) X was not created before A ran.

Subcase (a):

Suppose A read (or could have read) the post-value of B'. How does one find this B'?

For A to have seen the effects of B', A must have gotten a lock on the object after B' modified the object. Therefore, B' must be "at least" an action B_i that modified X such that $GA_{B_i}(A)\# < A\#$, since termination numbers flow in messages that allow the lock to be propagated from B' to A. Furthermore, $A\# < P\#$ because a child commits before its parent and termination numbers are sent in reply messages. So, $GA_{B_i}(A)\# < P\#$.

In addition, B' must be "at most" the latest such B_i, i.e., it cannot be a B_{i+1} where $GA_{B_i}(A)\# < P\# < GA_{B_{i+1}}(A)\#$. We show this by contradiction. Assume that A read post-B_{i+1}. Then $GA_{B_{i+1}}(A)$ must terminate before A could have gotten a read lock on X. This implies that $GA_{B_{i+1}}(A)\# < A\# < P\#$. We have our contradiction.

Note that just because $GA_{B_i}(A)\# < P\#$ does not necessarily mean that P actually saw some effects of $GA_{B_i}(A)$, either directly or through other actions. Nevertheless, it is always consistent to include $GA_{B_i}(A)$'s effects in pre-P if $GA_{B_i}(A)\# < P\#$. As a result, we choose the latest such B_i as B', the action whose post-value is used as pre-A.

Subcase (b):

This is like subcase (b) of (2) above. It applies to the case when an ancestor modified X after $GA_{B'}(A)$ committed but before A ran.

Subcase (c):

If there is no B_i such that $GA_{B_i}(A)\# < P\#$, then X could not have been created before A ran. Therefore, it is an error to ask for the pre-A of X.

We translate subcases (a) and (b) into an algorithm that uses entries in the pre-post log in Pre Rule 4. (We leave subcase (c) to the next Pre Rule.) Pre Rule 4 is much like Pre Rule 3, differing only in the definition of B.

Pre Rule 4:

Else if A has an aborted ancestor, let P be the youngest aborted ancestor of A. If there is an action B such that

1. B modified X,
2. P and B are not ancestor-related,
3. B committed up to LCA(B,P), and
4. $GA_B(P)\# < P\#$

find the latest entry E in the log that either

1. belongs to such a B, or
2. is a "Pre-" entry of an ancestor of A that is not tagged with a child's *aid*, or
3. a "Pre-Q, C" entry, where Q is an ancestor of A and C is Q's child such that C itself is not an ancestor of A and $C\# < GA_A(C)\#$, or
4. an "After-T" entry, where T is a child of an ancestor of A such that T itself is not an ancestor of A and $T\# < GA_A(T)\#$.

pre-A = version after E in the log, or the current version if E is the last entry in the log.

Example:

Consider the computation of Figure 4-4.

If $G1\# < P\# < G2\#$ and object X's pre-post log is <"Pre-G1", "Pre-G2.1">, indicating that G1 and then G2.1 modified X, then

pre-A = version in the "Pre-G2.1" entry.

If $G1\# < G2\# < P\#$ and the pre-post log is <"Pre-G1", "Pre-G2.1", "Pre-N">, indicating that N modified X after G1 and G2.1 but before P is called, then

pre-A = the current version, which holds the result of N's modification.

However, if the pre-post log is <"Pre-G1", "Pre-G2.1", "Pre-N, P">, indicating that N's modification was made after P ran, then

pre-A = version in "Pre-N, P".

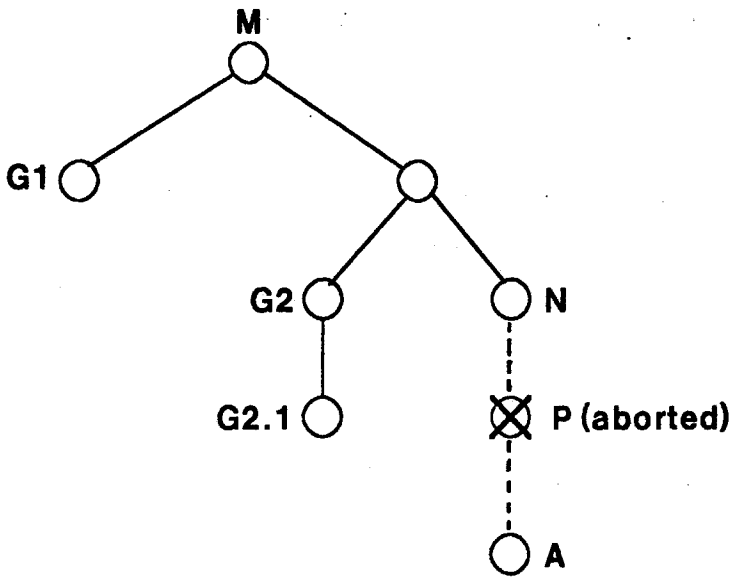


Figure 4-4: Example Illustrating Pre Rule 4

(4) Finally, we are left with the case where no action modified X before A was called. Recall that we create an initial "Pre-T" entry in the pre-post log for the nested topaction T that writes the initial value, by Saving Rule 0. Since A ran before T, X must not have been created yet when A ran. It is therefore an error to ask for the pre-A value of such an X. So, we have the following final Pre Rule.

Pre Rule 5:
Else,

pre-A = error.

In the rest of the thesis, we say that (the effect of) B is visible to A, where A and B are not ancestor-related, if B could have affected the pre-state of A. This is stated more formally in the following definition.

Definition:

B *is visible to* an action A that is not ancestor-related to it if

1. A and B committed up to their LCA and $GA_B(A)\# < GA_A(B)\#$, or
2. A did not commit up to LCA(A,B) but B committed up to LCA(A,B) and $GA_B(A)\# < P\#$, where P is the youngest aborted ancestor of A.

4.5.2.2 Post-Action Values

Now suppose we want the post-A value of X. If A modified X, either directly or indirectly, then there are the following ways that its post-value can be recorded in the log:

1. An ancestor is the first action to modify X after A ran. Then an "After-T" entry holding the post-A value will be saved by Saving Rule 3, where T is either an ancestor of A or is a child of an ancestor of A such that $GA_A(T)\# < T\#$.
2. An action B is the first action to modify X after A ran, where A and B are not ancestor-related and $GA_A(B)\# < GA_B(A)\#$. Then, by Saving Rule 1, a "Pre-B" entry holding the post-A value is stored.
3. An ancestor (maybe A itself) aborts before any other action has a chance to modify X after A ran. Then Saving Rule 2 saves a "Post" version for this ancestor.
4. X is not modified after A, and A and its ancestors did not abort. Therefore, the "Pre-A" entry is the last entry in the log and post-A continues to be in the current version.

If A aborted, we use the "Post-A" entry as its "post-value". This pseudo-post value will include the changes that A made. We choose to return this value because it is more useful to the user when debugging A. If there is no "Post-A" in the log, the post-A value is the first entry after pre-A that does not belong to either A or a descendant of A, or the current version if there is no such entry. So,

Post Rule:

If there is no "Pre-A" or "Pre-D" entry, where D is a descendant that committed up to A, then A did not modify X, either directly or indirectly. So,

post-A = pre-A, as computed by the Pre Rules.

else if there is a "Post-A" entry, then

post-A = the version in the "Post-A" entry

else searching forward from the "Pre-A" entry or a "Pre-D" entry, where D is a descendant that committed up to A,

post-A = version in the first entry that does not belong to either A or a descendant of A. If there is no such entry then post-A is given by the current version.

Example:

Using the action tree of Figure 4-4, if the pre-post log is <"Pre-G1", "Pre-G2.1"> then

post-M = the current version, by the Post Rule

The Pre and Post Rules are repeated in Figure 4-5 for easier reference.

4.5.3 Practical Considerations

In this subsection, we discuss two optimizations: one concerns the saving of initial entries in pre-post logs and the other concerns the searching of a pre-post log for an action's pre- or post- value.

Instead of saving a "Pre-T" version when an atomic object is created, where T is the nested topaction that writes the initial value, we propose to save an "Init T#" entry, where T# is the termination number of T. The "Init T#" will still be used as a "Pre-" entry, but it will not point to any saved version. We do not need a version in "Init T#" because "Init T#" will never be referenced by the Pre and Post Rules for a version. Instead, the Rules use it to determine whether an action ran before or after the creation of the object. If the action ran before the object's creation, it is an error to ask for the view of the action at the object; if the action ran after the object's creation, the pre- or post- action value is given by a version in an entry after "Init T#".

Figure 4-5: Rules for Calculating Pre-A and Post-A

Pre Rule 1:

If an entry tagged "Pre-A" exists in the pre-post log,

pre-A = version in the "Pre-A" entry.

Pre Rule 2:

Else if "Pre-D" exists where D is a descendant that committed up to A,

pre-A = version in the first such "Pre-D" entry in the log.

Pre Rule 3:

Else if there is an action B such that

1. B modified X,
2. A and B are not ancestor-related,
3. A and B committed up to their LCA, and
4. $GA_B(A)\# < GA_A(B)\#$

find the latest entry E in the log that either

1. belongs to such a B, or
2. is a "Pre-" entry of an ancestor of A that is not tagged with a child's *aid*, or
3. a "Pre-P, C" entry, where P is an ancestor of A and C is P's child such that C itself is not an ancestor of A and $C\# < GA_A(C)\#$, or
4. an "After-T" entry, where T is a child of an ancestor of A such that T itself is not an ancestor of A and $T\# < GA_A(T)\#$.

pre-A = version after E in the log, or the current version if E is the last entry in the log.

<<continued on next page>>

Figure 4-5: continued

Pre Rule 4:

Else if A has an aborted ancestor, let P be the youngest aborted ancestor of A. If there is an action B such that

1. B modified X,
2. P and B are not ancestor-related,
3. B committed up to LCA(B,P), and
4. $GA_B(P)\# < P\#$

find the latest entry E in the log that either

1. belongs to such a B, or
2. is a "Pre-" entry of an ancestor of A that is not tagged with a child's *aid*, or
3. a "Pre-Q, C" entry, where Q is an ancestor of A and C is Q's child such that C itself is not an ancestor of A and $C\# < GA_A(C)\#$, or
4. an "After-T" entry, where T is a child of an ancestor of A such that T itself is not an ancestor of A and $T\# < GA_A(T)\#$.

pre-A = version after E in the log, or the current version if E is the last entry in the log.

Pre Rule 5:

Else,

pre-A = error.

Post Rule:

If there is no "Pre-A" or "Pre-D" entry, where D is a descendant that committed up to A, then A did not modify X, either directly or indirectly. So,

post-A = pre-A, as computed by the Pre Rules.

else if there is a "Post-A" entry, then

post-A = the version in the "Post-A" entry

else searching forward from the "Pre-A" entry or a "Pre-D" entry, where D is a descendant that committed up to A,

post-A = version in the first entry that does not belong to either A or a descendant of A. If there is no such entry then post-A is given by the current version.

With an "Init T#", the debugging system will not have to remember T's termination number elsewhere. Furthermore, the Pre and Post Rules can use an "Init T#" more expeditiously than a "Pre-T". As we shall see in Sections 4.9 and 4.10, tagging the creation entry "Init T#" also fits in nicely with the way we propose to cope with crashes and storage reclamation.

We now present an efficient method that the debugging system can use to search a pre-post log for an action's pre- or post- value. In particular, the method quickly identifies that (small) part of the log where the value will most likely be. Like all algorithms for speeding up searches, the method is useful only if pre-post logs tend to be long.

We introduce a new type of entry, called a *top-marker*, into the pre-post log. When a topaction commits, we append a top-marker entry to the logs of all objects at which the topaction holds a write lock. The top-marker entry will contain the topaction's termination number. (Recall that the termination number of a committing topaction is sent from the coordinator to the participants during the *preparing* (i.e., first) phase of the two-phase commit protocol.) In addition, we singly-link all top-markers in a pre-post log; a top-marker will point to the next one in the log.

Using top-markers, this is how we isolate the sub-log within which the pre- and post- values of an action A, with topaction T, will most likely be found.

Chase the chain of top-markers, comparing the termination numbers in them against that of T until we find the last top-marker that has a termination number less than T#. If A committed up to its topaction ancestor, its pre- and post- values will be ahead of the identified marker in the log. Assuming that only a small number of descendants of the same topaction modify any one object, we do not have to search forward very far for A's pre- and post- action values.

If A did not commit up to its topaction ancestor, its pre- and post- values may be given by entries that are before the identified marker M. So, the debugging system

will have to search either forward or backward from M to find the pre- and post-values for such an A.

4.6 Implementing and Using the Universal Tree

All information about an action is saved in a record, called an *a-record*, in the volatile memory of the action's guardian. There is one a-record per action. The termination number of an action, for example, is stored in the action's a-record. For simplicity, we do not try to situate a-records optimally, nor replicate any of the information in them. Keeping information that is collected about an action with the action's guardian, rather than filing it in some remote repository, is consistent with the design principle of deferring debugging activities until debugging time.

When using the versions in a pre-post log to deduce pre- and post- action values, the debugging system has to be able to get information about the universal tree. In particular, it has to know

[1] whether A and B are ancestor-related,

[2] whether A# < B#, for actions A and B that are not ancestor-related, and

[3] whether action A committed up to ancestor P, and if not, who is A's youngest aborted ancestor.

[1] is provided by information contained in action identifiers (*aids*). Recall that we can determine the *aid*s of all of an action's ancestors, and their guardians, from examining the action's *aid*. We can also tell whether an *aid*'s action is a topaction.

[2] is determined by sending a *query* message to the guardians of A and B.

[3] can be decided by querying for the termination status of all descendants of P that are proper ancestors of A. (The status of an action is in the action's a-record.) This brute-force approach, however, may generate an unacceptable load on net traffic. The rest of the section presents an alternative scheme for deciding [3]. This scheme requires just one query in the typical case.

We keep a list of aborted descendants, called an *aborts-list*, in each handler action and topaction's a-record. The aborts-list of an action H does not necessarily contain all the aborted descendants of H. However, it contains at least all of H's (local) in-line subactions that abort. Call subactions of H that are forcibly aborted (because of communication problems or because an ancestor was aborted by another action) and therefore may have aborted descendants in other guardians that H does not know about are flagged in the aborts-list. The list of all aborted descendants of H, therefore, is the closure of H's aborts-list and the aborts-lists of forcibly aborted handler calls.

*Aborts-list*s are maintained as follows:

1. When a handler action or topaction first runs, its aborts-list is empty.
2. When an in-line subaction S aborts, S's *aid* is inserted into the aborts-list of the handler action or topaction of which it is an in-line subaction.
3. When a call action C is forcibly aborted, C's *aid* is inserted into the aborts-list of the handler action or topaction of which C is an in-line subaction. C's *aid* is flagged as potentially having aborted descendants elsewhere.
4. When a handler action H terminates, its aborts-list is forwarded to its caller. H's *aid* is included in the sent aborts-list, if H's termination is an abort. The aborts-list when received is merged into the aborts-list of the handler action or topaction of which the call action is an in-line subaction.

To decide whether A committed up to P, we query for the aborts-list of the handler action or topaction of which P is an in-line subaction. A committed up to P iff the aborts-list does not contain any ancestor of A that is a proper descendant of P. So, in the typical case where A commits up to P, only one query is needed. If A did not commit up to P, we may still be able to tell the youngest aborted ancestor of A from P's aborts-list: if the youngest ancestor of A in the aborts-list is not a call action that is flagged as possibly having aborted descendants elsewhere, it is the youngest aborted ancestor of A.

4.7 Arguments, Results, Input, and Output

In addition to objects in an action's environment, the pre- and post- states also consist of arguments and results, and terminal input and output. Recall that we allow the user to begin debugging with any handler action. As such, the debugging system must save arguments/results and input/output of all handler calls.

We save pointers to the decoded atomic argument objects that were originally used by a handler action, and the received encoded message, as well. When the value of a non-atomic argument is requested in the course of debugging, the debugging system will decode a copy from the saved message. (As an optimization, arguments once decoded can be saved so that subsequent requests for the arguments can be satisfied without a repeat decoding of the message.)

Note that we do not save pointers to the original decoded non-atomic arguments. A non-atomic argument may be (non-recoverably) modified during the course of the handler's execution; if all we saved was a pointer to the original decoded non-atomic argument, we would lose the argument's pre-handler action value. We could make copies of the non-atomic arguments and save these. But this would increase the computation's run-time unnecessarily, violating one of our design principles. We also note that the encoded message has to be saved anyway because our debugging system allows the user to retrace a handler action from its very beginning, i.e., the decoding of arguments. (See the next chapter for details.)

We save pointers to the original atomic argument objects and pointers to copies of the original non-atomic argument objects in the call action's a-record of the sending guardian, as well. This is to help in debugging the *encode* and *decode* procedures of the arguments' types. Comparing an argument at the call action with the corresponding argument at the handler action, the user can decide immediately whether the *encode/decode* of the argument had been done right.

Results are saved in a similar fashion to arguments and for the same reasons.

Pointers to the atomic result objects that were originally used in the handler and call actions are saved in their respective actions' a-records. Copies of non-atomic results are saved at the handler action but not at the call action. At the call action, the encoded result message is saved instead; non-atomic results are decoded from this saved message as needed. We note that saving results with the call action also helps in retracing; the user can skip the retrace of a handler call, when the target guardian is not available, without disrupting the rest of the retrace. (Of course, some retracing has to be done by the system to decode the non-atomic results from the saved message.)

Terminal input and output are restricted to **strings** in Argus. **Strings** are immutable and therefore atomic. We can simply save pointers to them. The sequence of terminal input to a handler activation or a topaction is saved in the action's a-record. These input values include those read by the action's in-line subactions. To be able to decide which subsequence was really read by a particular subaction (and its descendants), as is needed when presenting the pre-state of a subaction and when retracing, we tag each saved input with the *aid* of the action that read it. The sequence of terminal output that is associated with a handler activation or topaction is managed in precisely the same way as the sequence of input.

We could have saved input and output with the subactions that actually read and wrote it, and not with the handler action or topaction. But then constructing the sequence of input/output that is associated with a handler activation or a topaction becomes complicated; we will have to know how the input/output of the action's subactions interleave. Furthermore, because of the handler activation bias of our method, it is much more likely that users would query for the pre-state of a handler activation than an in-line subaction.

4.8 Summary of Information Kept About an Action

This section summarizes the information, other than the versions in the atomic objects, that is kept for deciding pre- and post- action values. All the information to be described is kept in a-records of actions.

Information used by the Pre and Post Rules is as follows:

1. *aid* of the action
2. status of the action, i.e., whether the action committed, aborted, or is still active
3. the action's termination number, if the action is terminated
4. if the action is a handler action or topaction
    a. the aborts-list, i.e., the list of known aborted descendants
    b. ordered list of input read by the action and its in-line subactions
    c. ordered list of output written by the action and its in-line subactions. Each entry in the input and output lists is tagged with the identifier of the subaction that read or wrote it.
5. if the action is a handler action
    a. name of handler
    b. pointers to the atomic argument objects that were originally decoded from the call message
    c. saved argument message, from which non-atomic arguments are decoded as needed
    d. pointers to the original atomic result objects returned by the handler action
    e. pointers to copies of non-atomic results
6. if the action is a call action
    a. pointers to the original atomic argument objects sent in the handler call

b. pointers to copies of non-atomic arguments

c. pointers to the atomic result objects that were originally decoded from the result message

d. saved result message, from which non-atomic results are decoded as needed

e. *aid* of the remote child

It is not too hard to come up with reasonable algorithms for collecting all the information required to build an action tree for the user, given the information we save in a-records. We omit the details here.

Of course, within a guardian, a-records should be grouped together to allow easier search. One way is to group a-records by their topaction ancestors and then to hash the groupings.

Finally, we note one further piece of information that goes into an a-record, namely, crash counts of all guardians visited by descendants that committed up to the action. As we shall see in the next section, crash counts help the debugging system know when versions that are crucial to the inference of a pre- or post action value have been lost because of a crash.

4.9 Effects of Crashes

The history that we collect for debugging is kept in volatile memory, and not the more expensive stable storage. This is in line with our design principles of not incurring unnecessary costs. As a result, when a guardian crashes and recovers, all pre-post logs and a-records in the guardian will be lost. From that point until some action writes into atomic object X in the guardian, we will not be able to deduce pre- and post- values at X of actions that ran and read X after the crash because X's pre-post log will be empty. We expect that writes are infrequent, when compared to reads, so that our ability to debug actions at the guardian will be seriously impaired for quite some time after a crash.

We propose the following solution. When a topaction that has a write lock on an object prepares to commit at the object's guardian, we save the topaction's termination number in the *prepared* record that is written into stable storage. When we recover an object after a crash, we create an "Init T#" entry, where T# is the termination number of the last topaction T that modified the object and that committed, i.e., finished the second phase of the commit protocol. As with object creation, this "Init T#" is treated as a "Pre-T" entry. The debugging system will use it to give the action views for actions that run after the crash only. In particular, the debugging system will not use an "Init T#" entry to give action views for T or any of T's descendants.

For actions that run after a crash to use "Init T#" entries correctly, these actions must have termination numbers that are greater than any of the "Init T#" entries that are created on recovery. So, when recovering from a crash, we set the termination counter of the guardian to one greater than the greatest termination number that is recovered. (Termination counters, like saved history, are volatile.)

B calls B.1
    B.1 reads object X at guardian G1
    B.1 commits
G1 crashes and then recovers
A calls A.1
    A.1 writes X at G1
    A.1 commits
A calls A.2
    A.2 writes Y at guardian G2
    A.2 commits
A commits
B aborts

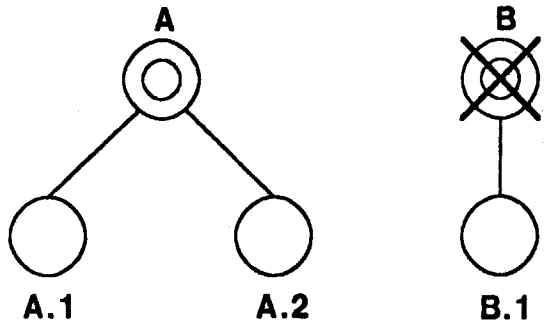


Figure 4-6: Example Motivating the Need to Remember Crash Counts

---

However, there may be actions that ran before a crash that have termination numbers greater than some of the "Init T.# " entries. We have to distinguish these actions if we are to avoid giving the wrong action views for them. We give an example of the kinds of problems that can arise in Figure 4-6. In the example, the Pre and Post Rules will give the pre-value of B as the post-value of A at objects X and Y, since B terminated after A and thus has a termination number that is greater than A's. This is wrong, since B.1 read X before A modified it. The problem is that B lost the read lock it inherited from B.1 because of the crash. B ended up terminating after A even though B.1 read X before A modified it.

So, we remember crash counts of guardians visited by descendants that committed up to an action, at the time the guardians were visited. A guardian's crash count is a monotonically increasing number that is kept stably and increased by one each time the guardian crashes and recovers. The orphan detection algorithm of Argus [Liskov 84] already maintains and uses these crash counts. Therefore, it requires little work on our part to incorporate them into our scheme.

We can use crash counts as follows. Suppose that we are looking for pre- or post- A of an object Z at G. If the crash count that A has for G is less than the current crash count of G, then a crash has occurred since A ran descendants in G. So, we must not use the Pre and Post Rules to give the action views of A at objects in G.

The above rule will prevent the debugging system from giving the wrong pre-B value at X in our example of Figure 4-6, but it does not apply to pre-B at Y. That is, the above rule does not take care of the case when B does not have a crash count for G, i.e., when B did not have descendants that visited G. We present a more general rule below.

In the orphan detection algorithm of Argus, each guardian G keeps a *map* in its stable storage that lists all guardians and their largest crash counts that are known to G. A guardian's *map* is sent in all messages from a guardian. When a guardian receives a *map* from another guardian, it merges the received *map* into its own *map*.

When merging the *maps*, if the two *maps* disagree about the crash count of a guardian, the higher crash count is retained.

The general rule to incorporate crashes in the way we compute pre- and post-action values is as follows:

> If the crash counts in G's *map* for guardians that are visited by A (and descendants that committed up to A) are greater than the crash counts that these guardians had when A (or descendants) ran there, then we cannot give the views of A and descendants that committed up to A at any object in G.

This rule will prevent the debugging system from giving pre-B values for objects X and Y in our example of Figure 4-6 because when A calls A.2 at G2, the call message carries with it the new crash count for G1.

4.10 Reclaiming Storage Space

In any practical system, saved history cannot be kept forever. In this section, we present a scheme for reclaiming storage from saved history. The goal of the scheme is to discard history in such a way that the debuggability of as few actions as possible is affected. The proposed scheme is adapted from Reed's scheme for pruning object histories [Reed 83].

In our scheme, the debugging system removes local a-records by the handler action or topaction. That is, when a handler action or topaction A is chosen as the victim, all a-records of local descendants are also removed with A's a-record.

Handler actions and topactions with earlier termination numbers are removed before later ones. When a topaction or handler action terminated some system- or user-defined time *t* before the current real time in the guardian, it (and all local handler actions and topactions that have termination numbers less than it) can be garbage collected.

The scheme, therefore, requires some mechanism for timestamping the termination numbers of topactions and handler actions with the real time of when the numbers were created. In addition, the "lag" time *t* and the real-time clocks of the guardians should be (approximately) synchronized so that the a-records of all of a topaction or handler action's descendants, including those that are remote, become eligible for garbage collection at approximately the same time.

When an action's a-record is removed, we can also remove the action's entries in the pre-post logs since these entries are no longer usable by the Pre and Post Rules. When all the entries of a topaction T (and descendants of T) are removed from a pre-post log, we replace them with an "Init T# ". In this way, the Pre and Post Rules will continue to be able to give action views of actions that read the object after T.

Entries in pre-post logs can be removed in two ways: either *lazily* or *aggressively*. In the lazy way, entries are removed when an object is accessed by an action or swept by the garbage collector in Argus. In the aggresive mode, the debugging system remembers in an action's a-record the objects that were modified by the action. When the a-record is garbage collected, the debugging system immediately prunes the pre-post logs of the objects that the action modified. We note that the debugging system does not have to build the lists of modified objects itself. The Argus run-time system already maintains such a list for each active action; all that the debugging system needs to do is save these lists.

Finally, we note that our proposed scheme does not require the Pre and Post Rules to be changed. This is because a-records and history in pre-post logs are truncated in much the same way that they would be truncated in crashes.

4.11 Summary and Discussion

In the chapter, we presented

1. a method for computing a serialization order,
2. two schemes for supporting pre- and post- action values of built-in atomic objects,
3. the implementation and use of the universal tree,
4. a way of saving arguments, results, input and output for debugging,
5. the effects of crashes on our debugging support, and ways to minimize them, and
6. a method that is based on aging for reclaiming space used by saved history.

We did not discuss support for pre- and post- action values of user-defined atomic objects. This discussion is deferred to Chapter Six.

In the rest of this section, we first compare and contrast the two schemes for supporting pre- and post- action values of built-in atomic objects and then we discuss pre- and post- action values of non-atomic objects and the effects of crashes on our ability to debug aborted actions.

In both schemes for computing pre- and post- values of atomic objects, the debugging system determines a pre- or post- action value simply by examining the local pre-post log of the object, and making (usually) one or two queries to other guardians. There is no searching through other objects' pre-post logs.

Both schemes give not only pre- and post- values of objects that were used by a committed or aborted action but also pre-values of objects that the action did not use. The latter pre-values are what the action might have seen if it had read the objects. This information is sometimes useful in debugging. More importantly, this information is crucial to our method for computing action views of user-defined atomic objects, as will be discussed in Chapter Six.

The first scheme is straightforward but inefficient; it serves mainly to motivate the optimized second scheme. The first scheme explicitly saves the pre- and post-values for all actions that modify an object, whether directly or through descendants. It also explicitly saves the read values for all actions that read an object. In all, it saves more information than is necessary.

The optimized scheme, on the other hand, does not save duplicate information. Post-values are stored only when necessary, i.e., just before the next time the object is modified. Pre-values for read-only actions are usually not saved at all. Thus, under what we expect to be normal circumstances, almost all of the versions that are saved and used in the optimized scheme are recovery versions created by the action system; the debugging system will not have to create very many copies of its own. Nevertheless, the debugging system has to use storage space to keep recovery versions around longer than they might otherwise be kept.

Notice that because we do not keep pre-values for read-only actions in the optimized scheme, we can no longer distinguish an action that read an object X from an action that did not use X at all. However, if this lost information is really important, the user can deduce it by retracing the action. Admittedly, this imposes more work and inconvenience on the user. However, the savings in stored history make it worthwhile.

In the case of an action that is still active, we can tell whether or not the action has accessed an object. If it has a lock, then it has accessed the object. So, its pre-value and those of descendants that committed up to it are determined; otherwise, the pre-values are not well-defined. For example, consider a topaction A that is active and that does not have a (read or write) lock on an object X. At the time we ask for the pre-A value of X, an action B may be the committed topaction to have modified X last, so that pre-A of X will be the post-value of B. However, there is no guarantee that pre-A of X will stay the post-value of B. If another topaction C were to modify X after B and then commit while A is still active, pre-A would change to the post-value of C.

Similarly, actions that are still active do not have well-defined post-values. Terminated descendants that committed up to an active action also do not have well-defined post-values, if the active action does not have a lock on the object. However, if the active action has a lock on the object, the post-value of a terminated descendant is defined.

The discussion above regarding when pre- and post- values of a terminated subaction are defined does not apply to aborted subactions (and their descendants). An aborted subaction's pre- and post- action values become defined as soon as the subaction terminates; its action values never depend on what ancestors do after it is called.

The algorithms for deducing pre- and post- action values of atomic objects do not apply to non-atomic objects since no history is saved in these objects. To get the value of a non-atomic object as seen in the pre-state of an action A, the user has to retrace from A's local handler action down to the call of A. (Support for retracing is discussed in the next chapter.) To get the non-atomic object's post-A value, A has to be retraced to completion.

Finally, we discuss a subtle effect of crashes on our ability to debug aborted actions and their descendants. Recall that our debugging system will refuse to give pre- and post- values of action A (and descendants that committed up to A) at objects in guardian G if the crash counts that A has of guardians A visited are less than those for the same guardians in G's *map*. This implies that we might not be able to debug descendants of A if A was aborted by a crash. This is undesirable if A was the one that caused its guardian to crash.

We propose the following solution: a guardian that crashes for reasons other than a node failure should continue to be available for debugging, with saved history intact; the crash count of the guardian remains the same while it stays "crashed." Unfortunately, this solution exacts a relatively high price: while the guardian stays "crashed", it is accessible only to the debugger and cannot be used by actions.

Chapter Five

Retracing an Action

In Phase Two, the debugging system recreates the execution of a previously executed action on direction from the user. We refer to this activity as a *retrace* of the action. A retrace has the following properties:

1. It is a serial execution that is equivalent, in the sense of serializability theory [Eswaran et al. 76, Bernstein & Goodman 81], to the original computation.
2. The user is able to examine intermediate states by stopping the progress of a retrace with breakpoints.
3. A retrace does not interfere with the regular use of data objects and guardians. In particular, it does not delay the progress of actions.

In this chapter, we discuss how retracing with all of the properties above can be implemented using an action's tree and the history that is saved in built-in atomic objects. We begin with an overview of the implementation in Section 5.1, and then divide the details into three sections: one on action creation and termination, one on object creation, and one on accessing objects. In this chapter, we concentrate on actions that use only non-atomic and built-in atomic objects. We discuss retracing actions that use user-defined atomic objects in the next chapter.

Since the saved history is kept in volatile memory, an action is retraceable only if no crash has occurred since it ran.

5.1 Overview

A retrace is implemented by re-executing code and has to begin with a handler action. This is because the debugging system does not save environments. The local environments of an in-line subaction's ancestors, therefore, have to be

recreated if the in-line subaction is to be retraced. On the other hand, the environment of a handler action consists solely of its guardian's variables, which are never de-allocated provided the guardian does not terminate.

When the user retraces a handler action, the debugging system creates a *retrace* process to run the handler whose name is in the handler action's a-record. This retrace process is used to retrace the in-line subactions that are called from the handler action, as well.

A retrace process maintains information about the state of its retrace. For example, it keeps track of the *aid* of the action that it is currently retracing. (A summary of the retracing state is given in Section 5.5.) Since a retrace's state is kept with its own process, concurrent retraces will not interfere with each other.

The user has the option of skipping the retrace of subactions (in-line subactions or handler calls) in a retrace. The user may want to skip a subaction if he or she feels that the subaction has nothing to do with the bug that is being isolated. When a handler call is skipped, no call message is sent. Instead, the result message of the original call, which was saved in the call action's a-record, is decoded and used as if the handler call had just returned it.

Terminal input during retrace of an action is taken from the input values that are saved from the original computation. These input values are tagged with the *aids* of the subactions that read them and are kept as an ordered list in the a-record of the handler action or topaction of which the action being retraced is an in-line subaction. As a result, when an action reads from the terminal it will be given exactly the same input it originally received. To help the user follow the progress of a retrace, the debugging system will send input values that are read to the user's terminal. For the same reason, terminal output during retrace is directed to the user's terminal.

5.2 Action Creation and Termination

A retrace process can be actively retracing only one action at a time. We call this action the current action of the retrace process.

When a retrace process executes an **enter** statement, which gives rise to a child action, it has to find the *aid* of the child. To do this, a retrace process keeps track of its position in the action tree. In particular, it remembers the current action's last child to terminate in the retrace, i.e., the last child to be retraced or skipped.

When a retrace process executes a **coenter** statement, which is Argus' statement for spawning a group of concurrent child actions (called *coarms*), it retraces the coarms one after another in their termination order. Active coarms, i.e., those that are stopped but not terminated, are retraced in some arbitrary order after the terminated coarms in the group; the order does not matter because an active coarm cannot possibly have seen the effects of another active coarm.

Since a retrace process remembers its current action's last child to terminate in the retrace, it can deduce the *aid*s that are associated with a group of coarms from the current action's tree. However, coarms in Argus may run different code or be given different "arguments".[7] So, how do we tell which *aid* to use with which coarm when retracing? (Remember, we do not keep pointers to code or "arguments" in a-records of coarms.) We assume that the system assigns *aids* in such a way that we could tell the order of the *aids*' creation by simply comparing the *aids*. We also assume that coarms are created in the same total order each time the same coenter statement is executed. Based on these two assumptions,[8] it is clear that we can match *aid*s to coarms when retracing. For example, if the system assigns *aids* in

---

[7] In Argus, a parent may provide values for some local variables in a coarm. This is very much like argument passing. However, we do not allow the user to ask for these "arguments" in Phase One of the debugging method. Instead, he or she has to retrace the coarm if values of these "arguments" are desired.

[8] both of which are satisfied in the current implementation of Argus

increasing order, then the smallest *aid* among the group of coarm *aid*s is the one associated with the first coarm generated by the **coenter** statement, the second smallest with the second coarm, and so forth.

When the retrace process executes a call action, it encodes and sends the arguments off in a message that is tagged as a retrace but otherwise is like that of a regular call message. The process then waits for the call's results. On getting the results, it decodes them and proceeds with the rest of the retrace.

When a retrace process is created to retrace a handler action, it first decodes the arguments either from the retrace call message if the handler action is called from another retrace process, or from the original call message saved in the handler action's a-record if the retrace is directly initiated by the user. When the retrace of a handler action ends in the encoding of results, the retrace process sends the result message to the call action's guardian if the retrace was initiated by a retrace call message. Otherwise, the retrace process sends the result message to the user's controlling terminal. A retrace process terminates when it completes the retrace of its handler action.

When a retrace process completes the retrace of an in-line subaction S, it updates the parent's last child to terminate to S and continues with the parent's retrace.

Finally, we note that a retrace process has to be careful not to re-execute noticeably past the point where its current action is stopped in the original computation. Below, we give the conditions for determining when the retrace process has come to the end of its current action. Note that the retrace of an action may also end because needed history has been discarded and is no longer available.

The retrace of a terminated action is complete when one of the following events occurs:

1. A child action is to be created but we have run out of children *aid*s from the original computation.
2. A built-in atomic object is to be created but there is no record that this particular creation was ever made in the original computation. (Retracing create calls is discussed in the next section.)
3. The result of an equality comparison between two non-atomic objects is needed but there is no record of such a comparison in the action's list of "equal results". (See Section 5.4.2.)
4. The action attempts an I/O and there is no record of that particular I/O in its handler action's a-record.
5. The action attempts to write into a built-in atomic object that it did not write in the original computation.[9]
6. The action terminates.

The retrace of a stopped but unterminated action is complete when one of the following events occurs:

1. Any one of the first four of the events listed above for ending the retrace of a terminated action.
2. The action attempts to read from a built-in atomic object for which it does not have a read lock.
3. The action attempts to write into a built-in atomic object for which it does not have a write lock.
4. The action attempts to terminate.

5.3 Creating Objects

A call to the create operation of a built-in atomic type in a retrace must return the same object as the one that was returned by the corresponding create call in the original computation. This is because the history collected in an atomic object may be needed to create a serial execution that is equivalent to the original computation. As discussed in the previous section, we need the history to tell where an active

[9] An action A wrote into an object in the original computation iff the object's pre-post log has either a "pre-A" or "After-T" entry, where T is a child of A. See Section 4.5 of Chapter Four for details.

action is currently stopped. In addition, a retrace process can skip the retrace of subactions without adversely affecting the rest of the retrace only if the saved history in the original built-in atomic objects is available. We give details on how a retrace process skips subactions in the next section.

A simple method to ensure that original objects are returned in retraced create operations of built-in atomic types is as follows. We keep in an action's a-record a log of pointers to built-in atomic objects that the action created in the original computation, in the order they were created. In a retrace of the action, the retrace process picks objects from this log when create calls of built-in atomic types are executed. Objects are selected in FIFO order. A retrace process maintains an index into the corresponding list of created atomic objects for each action that it has begun to retrace but has not completed.

In contrast to atomic objects, new non-atomic objects must be created in a retrace. Otherwise, the retrace of an action may modify the regular system state and interfere with running actions. New non-atomic objects do not cause problems in the retrace of actions, as we shall see in the next section.

5.4 Accessing Objects

When a retrace process reads an object in a retrace, it has to see the same value as the one that was read by its current action in the original computation. Similarly, when the user asks for the value of an object in a retrace, he or she must receive the value that reflects the modifications of the serial execution up to the point where the retrace is stopped. In this section, we show how values of an object at different points in a retrace can be determined.

5.4.1 Atomic Objects

In a retrace, accesses to atomic objects do not secure locks. This is because we are reenacting history when retracing an action; the place of the action in the serialization order is already defined.

Each retrace process has a map that gives the *current retracing versions* (CRV) of atomic objects. The CRV is the retracing counterpart to an object's current version. It holds the current value of the object in the retrace. Only atomic objects that have been accessed thus far in the retrace are in the retrace process's CRV map.

When a retrace process reads or modifies an atomic object, say X, it reads or modifies the object in X's CRV. If the retrace process does not have an CRV for X, it creates one; the newly created CRV is initialized to the post-value of the current action's latest child to commit in the retrace, or the pre-value of the current action if no committed child has been retraced thus far. Since the retrace process already keeps the *aid* of the latest child to be retraced (see Section 5.2), it can find the latest child to commit in the retrace from the action tree. Obviously, we can optimize by explicitly remembering the latest committed child to be retraced, if necessary.

When a handler call is made from the current action, it is possible that some of the retrace process's CRVs will not be current anymore when the call returns. For example, consider the retrace of the following sequence of events:

1. A modifies atomic object X
2. A makes handler call H
3. H modifies X, either directly or via a descendant
4. H returns
5. A reads X

A's retrace process will create an CRV for X when step 1 is retraced. Since each handler action has its own retrace process, the process that retraces A will differ from the one that retraces H. So, X's CRV in A's retrace process will not reflect H's modification even after H is retraced. A's retrace process, therefore, will read the wrong value in step 5.

Notice that a retrace process's CRVs can be invalidated only if a handler call has descendants in the same guardian as the retrace process. However, we propose

a solution that does not take advantage of this information, mainly because the solution is simple and it works for skipping subactions, as well. The solution is as follows:

> When a committed subaction is skipped or when the current action makes a handler call that commits, the retrace process discards its CRV map.

After the subaction or handler call, the retrace process re-builds the CRV map with correct current values as atomic objects are accessed. Note that the retrace process does not have to discard its CRV map after an aborted handler call or a skipped aborted subaction: aborted handler calls and aborted in-line subactions cannot affect the values in the retrace process's CRVs.

When the retrace process retraces an in-line aborted subaction, it has to use a new (initially empty) CRV map. This is because the view of an aborted subaction is not tied to that of its parent's: the CRVs in the old map may not be the values that the aborted subaction saw. When the aborted subaction returns to the parent, the retrace process must get rid of the CRV map that it used when retracing the subaction. The retrace process can either re-use the CRV map it had just before retracing the subaction, or begin afresh with an empty CRV map.

When a retrace is stopped because of a breakpoint and the user asks for an atomic object's value in that intermediate state of the retrace, the debugging system displays the object in the CRV if the atomic object has an CRV in the retrace process. If there is no CRV for the atomic object, the debugging system displays the post-value of the last committed child in the retrace, or the pre-value of the current action if no committed child has been retraced thus far.

5.4.2 Non-Atomic Objects

Recall that by our assumptions (Section 2.3), an action that writes into a non-atomic object must also be the one to have created it. So, when we come to a write of a non-atomic object during a retrace, that non-atomic object must have been created by the same action earlier in the retrace. It is alright for the retrace process to go ahead and modify the non-atomic object — the regular system state cannot be affected in any way. Also, since the assumptions ensure that actions continue to be serializable even when they share non-atomic objects, reading and writing directly into the current (and only) versions of non-atomic objects in a retrace will give results that match those of the original computation. In particular, non-terminated concurrent siblings can be retraced in any arbitrary order without any problems. (Our assumptions disallow concurrent siblings from communicating information to each other through non-atomic objects; they may read but not write non-atomic objects that are not in their local environments.)

In Argus, two objects are equal if they are the same object. Since new non-atomic objects are created in a retrace, a problem with equality comparisons may arise. Consider the following example.

1. Action P creates a non-atomic object X
2. P assigns X to variable V
3. P calls a subaction S
4. S inserts the object referred to by V into an atomic object Y
5. S commits
6. P asks whether V's object equals the object in Y

In step 6 of the original computation, P will find that the object in V equals the object in Y. However, in a retrace of P, step 6 will return a negative result if the user skips the retrace of S or if S is a handler call. This is because P's retrace process is forced to use the post-S value of Y in step 6; the non-atomic object saved in this post-S value does not equal the one that is created by the retrace process in step 1.

One solution that the debugging system can take is to save the results of all

equality comparisons between non-atomic objects that are made by an action. These results are saved as a list in the action's a-record and are used to reproduce the results of equality comparisons between non-atomic objects in a retrace. This list of "equal results" is used in a retrace in precisely the same way that the list of pointers to created atomic objects (Section 5.3) is used.

5.5 Summary and Discussion

We first summarize this chapter with a list of history that is saved to support the retrace of actions. We then discuss some properties of retracing. Finally, we end with a discussion of how other kinds of actions and activities in Argus might be debugged.

For the most part, history that is saved for Phase One of our debugging method is what we use for supporting the retrace of an action. However, there are additional information, data structures, and capabilities that are needed to make retracing work. These are:

1. A list of pointers to built-in atomic objects that an action creates is saved in the action's a-record.
2. The results of equality comparisons of non-atomic objects that are made by an action is saved in a list (called "equal results") in the action's a-record.
3. The following information is maintained in a retrace process's state:

    a. whether the retrace process was initiated by the user or by a retrace call message

    b. *aid* of the action that is currently being retraced

    c. *aid* of the current action's last child to be retraced (or skipped),

    d. a map (called CRV map) of atomic objects to their current values in the retrace, and

    e. for each action that the retrace process has begun to retrace but has not completed:

        i. whether the action is active

ii. an index into the action's list of created atomic objects, and

iii. an index into the action's list of "equal results".

4. The implementations of built-in atomic types must understand "retrace" mode so as to use the saved history described above correctly.

We note that a retrace process does not use any of the data structures in an atomic object that an action uses, other than to read from the pre-post log. Furthermore, it does not modify non-atomic objects in the regular system state. Therefore, retracing cannot interfere with the regular system state and cannot delay the progress of actions.

To retrace a nested topaction, the user has to retrace from the handler action of which the topaction is an in-line action. This is because a nested topaction may use objects in its parent's local environment; this environment has to be set up before the nested topaction can be retraced. In addition, a retrace process will use a new (initially empty) CRV map when retracing the topaction since a nested topaction's view is not connected to that of its parent's. Retracing nested topactions, therefore, is no different from retracing in-line aborted subactions.

When retracing an action A, the user may skip the retrace of any nested topaction or subaction, without adversely affecting the retrace of the rest of A. This is because a nested topaction or subaction, by our assumptions, can communicate information to A only through atomic objects. This information is available in A's environment even if the nested topaction or subaction is not retraced.

In the previous chapter and in this chapter, we considered only actions that are either handler activations, subactions of handler activations, or nested topactions. We will now briefly discuss other kinds of actions and activities in Argus and how they might be debugged:

1. A guardian in Argus may have code that is not run as an action. However, this **background** code serves mainly as a mechanism for

initiating (non-nested) topactions and tends to be simple, as a result. Debugging background code, therefore, often requires no special aid.

If a non-nested topaction does not use any of its **background** code's local environment, then the debugging system presented thus far can be used to debug the topaction, as is. Otherwise, we will have to save a checkpoint of the topaction's pre-state at the time the topaction is called. This involves saving the parent's local environment and copies of non-atomic objects referenced from the local environment. Atomic objects need not be copied because their values in the pre-state of the topaction can be deduced.

2. To create a guardian, an action calls a **creator** of the guardian's type. A subaction of the calling action is created to run the creator's code, much like a handler call. When a guardian recovers from a crash, a topaction is created to run the guardian's **recover** code and initialize its volatile variables. If a creator or recover action aborts, the guardian it is creating or recovering crashes.

   Debugging creator and recover actions requires some extra machinery but otherwise is much like debugging handler activations. For instance, when retracing a creator or recover action, the debugging system must not assign to the guardian's variables; otherwise, actions that use the guardian will be affected. Instead, a different set of temporary variables must be used for the retrace. Also, to allow the user to debug creator and recover actions that abort, guardians that crash should continue to be accessible for debugging. (We already require this capability for debugging orphans in Chapter Four. In general, guardians that crash for reasons other than node failures should be available for debugging, with saved history intact.)

3. We have assumed that only atomic objects are shared among actions that are not ancestor-related. If we were to relax this assumption and allow non-atomic objects to be shared, we will have to save the result of every read if actions are to be retraceable. Alternatively, if reads are more frequent than writes, we can record writes instead of reads, and use placeholders to tell us which value an action read.[10] It is not clear that there is any way around the expense of saving values on every read or every write.

   We note that saving on every read will work just as well for non-atomic objects that are not deterministic: actions that use these non-deterministic objects will continue to be retraceable. We also note

---

[10] This is how we save state changes at mutex objects. See Chapter Six.

that shared non-atomic objects, in general, will not have well-defined values in the pre-states of actions: consecutive reads by an action may return different values. Consequently, Phase One of the method will not be as effective if too many non-atomic objects are shared.

Chapter Six

Debugging Programs That Use User-Defined Atomic Objects

In Argus, users can program their own atomic abstract data types if they want types that are closer to an application's domain or that allow more concurrency than built-in atomic types. Like regular abstract data types, the implementation of a user-defined atomic type has to satisfy the type's sequential specifications, i.e., the implementation has to be correct in the absence of concurrency and node crashes. In addition, the implementation of an atomic type has to ensure that:

1. actions that access user-defined atomic objects continue to be recoverable and serializable, and
2. user-defined atomic objects are restored after a crash to the states they had at the end of the last committed topaction before the crash.

We refer to the two properties above as *atomicity* and *resilience*, respectively.

In this chapter, we consider the debugging of actions that use not only built-in atomic objects but those of user-defined atomic types as well. We begin, in Section 6.1, with a discussion of the difference between an object's *concrete* and *abstract* values. This distinction is especially important in the case of user-defined atomic objects in Argus because an "undo" of an aborted action's effect will often restore the *abstract* value of a user-defined atomic object without also completely undoing the action's modification of the object's *concrete* value.

In Section 6.2, we review the implementation of user-defined atomic types in Argus. In Section 6.3, we give the history that must be saved to provide pre- and post- action values of user-defined atomic objects and to allow the retrace of operations at these objects. We discuss the use of this saved history in Sections 6.4

and 6.5. Finally, in Sections 6.6 and 6.7, we discuss how the effects of crashes and storage reclamation can be minimized.

In this chapter, we consider only user-defined atomic types that are implemented correctly. The debugging of the implementation of a user-defined atomic type is covered in the next chapter.

6.1 Abstract Values

In this section, we discuss the *concrete* and *abstract* values of objects and how the two values are related to each other. The distinction between the values are crucial to the understanding of Argus's mechanism for implementing user-defined atomic types.

An implementation of an abstract data type will use an *internal representation* (**rep**) to represent an object of the type [Guttag et al. 78]. A **rep** itself is an object. For example, a *set* may be implemented with an array as its **rep**. The *concrete* value of an object is the value of its **rep**. For example, the concrete value of a set object discussed above is the value of the array that is its **rep**.

Each implementation of an abstract data type T has an *abstraction function* that maps a concrete value of an object of type T into a value in the abstract domain. These values in the abstract domain are called *abstract* values. We note that different concrete values may map into the same abstract value. For example, a set object S1 whose concrete value is the array [1, 2] may have an abstract value that is the mathematical set {1, 2}. A set S2 whose concrete value is the array [2, 1] may also map into the same mathematical set {1, 2}. Thus, the abstract values of S1 and S2 may be equal even though their concrete values are not.

The meaning of equality among abstract values is dependent on the specification of the type. We assume that the user knows the specification of the type and can tell whether two values that are *displayed* are equal. Recall that we are

assuming that each implementation of a type has a *display* operation. This operation functions as a (kind of) abstraction function. In this thesis, we assume that *display* operations are implemented correctly. A person who is debugging should be able to deduce the abstract value of an object from the result of the *display* operation without having to know the details of the implementation.

We have used and will continue to use the word "value" without qualification to mean the *abstract* value of an object in this thesis.

6.2 User-Defined Atomic Objects

In this section, we discuss the mechanisms provided by Argus for implementing atomicity in user-defined atomic types. Resilience will be discussed in the next chapter.

We will be referring to semiqueues in examples throughout this chapter and the next. A semiqueue is a bag or multi-set. It can be regarded as a kind of queue where dequeuing is not restricted to FIFO order. A *deq* operation can return any item in the semiqueue that was inserted by an action that committed relative to the dequeuing action and that has not yet been dequeued. An implementation of the semiqueue type is given in Figure 6-1. The implementation is taken from [Weihl & Liskov 82].

A user-defined atomic object, e.g. a semiqueue, may have a non-atomic object in its **rep**. Unless the non-atomic object is immutable or not modified at all by using actions, an operation execution will need to exclude other concurrent operation executions from accessing the non-atomic object while it uses the object. Argus introduces a parameterized data type called **mutex** for exactly this kind of mutual exclusion. A mutex object consists of an exclusive lock and a resource (an object of the type specified in the mutex's parameter). If all operations are coded to **seize** the mutex's lock before using the resource, a process will have exclusive use of the resource while it has the lock. A lock is released at the end of the **seize** statement

Figure 6-1: Implementation of the Semiqueue Type

```
semiqueue = cluster is create, enq, deq

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

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

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

   deq = proc (q: cvt) returns (int)
            cleanup(q)
            seize q do
               while true do
                  for item: qitem in buffer$elements(q.value) do
                     % look at all items in the buffer
                        tagtest item
                           % for an item that can be dequeued by this action
                        wtag enqueued (i: int): qitem$change_dequeued(item,nil)
                                                return(i)
                        end % tagtest
                     end % for
                  pause
                  end % while
               end % seize
            end deq
```

```
    cleanup = proc (q: rep)
              enter topaction                    % start an independent action
                 seize q do
                    b: buffer := q.value
                    for item: qitem in buffer$elements(b) do
                       tagtest item              % to remove items in the dequeued state
                          tag dequeued: buffer$reml(b)
                          others: return
                          end % tagtest
                       end % for
                    end % seize
                 end % enter — commit cleanup action here
              end cleanup

end semiqueue
```

---

that acquires it, or when a **pause** is executed. In the latter case, the executing action will stop for some system-defined amount of time before attempting to re-acquire the lock and continue execution. For example, in the definition of the semiqueue type, all operations (*enq*, *deq*, and the internal routine *cleanup*) **seize** the **rep**'s lock before accessing the buffer in the **rep**. Thus, whoever has the lock is guaranteed exclusive use of the buffer.

Mutexes can prevent unwanted interference between concurrent operation executions. However, to synchronize actions, the programmer has to take advantage of the atomicity properties of built-in atomic objects. Actions can be synchronized through their use of some common built-in atomic object that is in the **rep** of the user-defined atomic object. Furthermore, any change to the abstract state[11] of a user-defined atomic object must include modifying a built-in atomic component in such a way that should the modifying action abort, the automatic recovery of the built-in atomic component is sufficient to undo the action's modification of the object's abstract state. For example, in the *enq* of an item into a semiqueue object, the atomic variant inserted into the (non-atomic) buffer of the representation will

---

[11] An object's state is its value.

revert to a "<dequeued: **nil**>" state if the *enq*'ing action aborts. Even though the atomic variant continues to be in the buffer, it is treated as a "non-item" by the semiqueue operations, so the abstract state of the semiqueue is effectively restored to what it was before the *enq* operation. Note that no user code is run to undo an aborted action's modification of a user-defined atomic object.

The *deq* operation of the semiqueue uses a **tagtest** statement to skip over items in the buffer that are not candidates for dequeuing. A **tagtest** statement is like a case statement except the arm of a **tagtest** additionally requires that a lock on the atomic variant be secured in a specified mode. A **tag** arm specifies a read lock, a **wtag** arm a write lock. If the appropriate lock cannot be secured, the condition for the arm fails. An **others** arm, if provided, will be chosen if all others fail; without an **others** arm, the **tagtest** statement will simply terminate if no arm succeeds. The **tagtest** statement helps in the implementation of non-determinism in user-defined atomic types.

Because of the requirements of action synchronization, we assume that all user-defined atomic types will have a **rep** that is one of the following:

1. an atomic object (built-in or user-defined)
2. a mutex object, or
3. an immutable structure (**struct** or **sequence**) of mutex and/or atomic objects.

In addition, we assume that an operation accesses non-atomic objects in a mutex only when it has the mutex's lock.

It is obvious that the restriction on the structure of a user-defined atomic type's **rep** can be checked statically. However, it is not as easy to verify that non-atomic objects in a mutex are used only when the mutex's lock is held; we have to be careful that there are no "hidden" pointers so that the non-atomic objects are accessible via a different path. This task is beyond the scope of the thesis.

We have already shown, in the previous two chapters, how our debugging

technique works with built-in atomic objects. User-defined atomic types with built-in atomic **reps** do not present any new problems. Therefore, in the rest of this chapter, we concern ourselves only with user-defined atomic objects that contain mutexes in their **reps**. In particular, we work out details for the following:

1. pre- and post- action views of user-defined atomic objects that contain mutexes, and
2. recreating the results of accesses at these mutex objects during retracing.

We begin with a study of what part of a mutex's history to record.

6.3 Recording History

6.3.1 State Changes of Mutex Objects

History regarding changes of a mutex's state is kept in a special *pre-post* log in the mutex object.

As with built-in atomic objects, we assume that when a mutex object is created, the system runs a nested topaction that writes the initial value into the object and then immediately commits. We also save an initial ("Init") entry in a mutex's pre-post log when the mutex is created to distinguish actions that ran before the creation from those that ran after the creation.

Saving Rule 0:
When a mutex object is created:

- Save a pointer to a copy of the initial value.
- Tag the newly inserted entry in the pre-post log as "Init T#", where T# is the termination number of the nested topaction T that writes the initial value.

Accesses to a mutex object from one action may interleave with those from other actions at that mutex object. As a result, we have to save a mutex object's state either before every write or after every write if we are to reproduce the results of an

action's operations through retracing the action. In addition, we need to remember the timing of reads in relation to writes.

Saving Rule 1:

When an action A releases a mutex lock:

1. If the compiler can tell that the mutex's resource was not modified in the **seize**, create a *read placeholder* in the mutex's pre-post log. Label this newly inserted entry with A's *aid*.
2. Otherwise, make a copy of the current version[12] and save a pointer to the copy in the mutex's pre-post log. Label this newly inserted *post-mod* entry with A's *aid*.

We choose to save mutex versions after an access instead of before because the compiler is better able to tell whether the mutex resource has been modified within a **seize** given the exit that is being taken.

In the algorithms for computing pre- and post- action values, to be described in Section 6.4, we will treat an "Init T# " entry like a post-mod entry. Note that an "Init" entry always belongs to a topaction and that a termination number and not an *aid* is saved in the entry.

We will also treat read placeholders as post-mod entries; the mutex version that is associated with a read placeholder is the version in the last real post-mod entry before the placeholder in the log.

6.3.2 Results of Lock Tests at Built-in Atomic Objects

Much like accesses of a mutex object, tests of the status of a built-in atomic object's lock by an action may interleave with changes to the lock status by other actions. The results of the executions of **tagtests** and other lock testing operations, therefore, must be saved if we are to retrace actions in serialization order. For

---

[12] We copy down to contained atomic and mutex objects. Contained atomic and mutex objects do not have to be copied because they maintain their own history.

example, assume that the following sequence of operations were made at an initially empty semiqueue object:

A calls *enq* with 1 as argument
    A's *enq* returns
B calls *enq* with 2 as argument
    B's *enq* returns
B commits at the semiqueue object
C calls *deq*
    C's *deq* returns 2
A commits at the semiqueue object
C commits at the semiqueue object

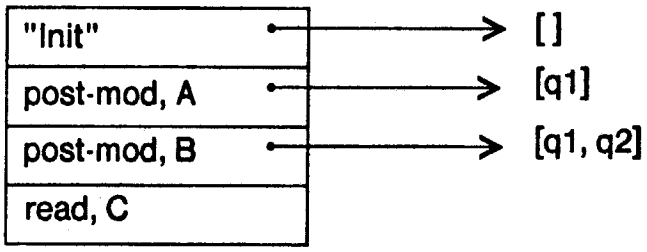
Figure 6-2: Example 1

A, B and C are topactions. C's *deq* returned the element inserted by B, even though the very first element it examined was that of A's. This is because the **tagtest** statement that C executed on A's element failed to secure a write lock; A was still active then. Now, if we were to retrace C after A, B and C all committed, C will *deq* A's element instead of B's, if the original response to the **tagtest** on A's element was not saved! Our retrace will not be equivalent to the original execution.

So, we add a "lock-test" list to the a-record of an action; responses to lock tests made by the action during normal execution are saved in this log. This list is much like two other lists that we already keep in an action's a-record, namely, the list of pointers to created atomic objects (Section 5.3) and the list of "equal results" (Section 5.4.2). Our debugging system uses a lock-test list in precisely the same way it uses these other lists when retracing an action. We note that the history in a lock-test list does not contribute in any way to the calculation of pre- and post- action values.

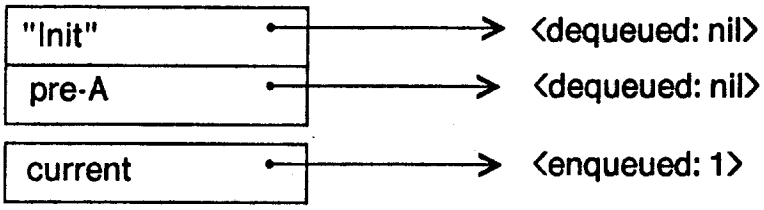
Figure 6-3 gives the pre-post and lock-test list of C that result from the example in Figure 6-2. Notice that the names of the actions that created the various objects have been omitted from the "Init" entries in Figure 6-3.

pre-post log of **buffer** (**rep** of the semiqueue object):

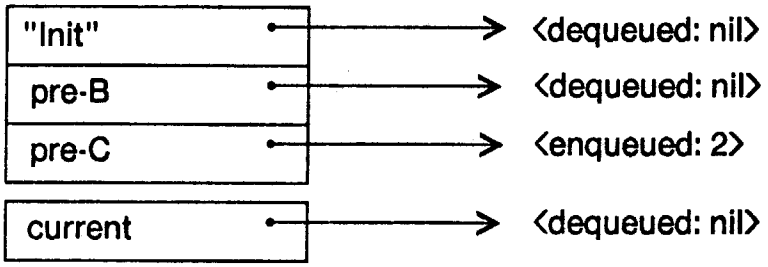


q1 and q2 are qitems (i.e. atomic variants) inserted by A and B respectively, and they have their own history as shown below.

pre-post log of **q1** (qitem inserted by A):



pre-post log of **q2** (qitem inserted by B):



C's lock-test list:

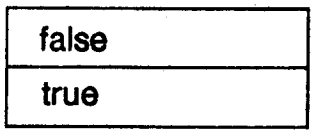


Figure 6-3: Pre-post and "Lock-test" Logs of Example 1

6.3.3 Created Mutex Objects

To retrace an action that accessed user-defined atomic objects, we must be able to reproduce the results of the action's operations at mutex objects. A create call of a mutex type, when retraced, must therefore return the object that was created in the original computation. Otherwise, we will not have the crucial history that was so painstakingly recorded in the pre-post log of the original objects.

Recall that we have already encountered in Section 5.3 the need to return original built-in atomic objects when retracing create calls. The solution we presented there keeps, in an action's a-record, a log of pointers to built-in atomic objects that the action created. During retracing we maintain an index into the log that gives the object to return when the next create of a built-in atomic type is retraced. To have original mutex objects returned when create calls are retraced, we simply include mutex objects in this log of created objects, and have the mutex type manager access the log for the appropriate object when its create operation is called in retrace mode.

6.4 Pre- and Post- Action Views

In this section and the next, we show how the history saved in the pre-post log and lock-test list can be used to extend our debugging method to include actions that use user-defined atomic objects. This section discusses the support for pre- and post- action views of user-defined atomic objects; the next section covers retracing operations at these objects.

6.4.1 Rep with a Single Mutex Object

We begin with user-defined atomic objects whose **reps** consist of a single mutex object.

The mutex version that we choose to represent the concrete pre-value of an action A at the mutex's user-defined atomic object must be one from which we can

deduce the net effect of all accesses visible to A, i.e., accesses made by actions that are visible to A and accesses made by (proper) ancestors before A was called.

Definition:

> The pre-A value of a mutex object is the latest post-mod version in the mutex's pre-post log that is tagged with either
>
> 1. the *aid* of a action B that is not ancestor-related to A such that $GA_B(A)\# < GA_A(B)\#$, or
> 2. the *aid* of a proper ancestor of A.
>
> If there is no such post-mod version, then it is an error to ask for pre-A of the mutex.

Note that the "Init" version is considered a post-mod entry. So, if there is no post-mod version that satisfies condition 1 in the definition above, then the mutex object must not have been created before A ran.

In the example of Figures 6-2 and 6-3, A is serialized after B even though A inserted an element into the semiqueue object before B. Using our definition, the concrete pre-A value of X will be given by B's post-mod version in the pre-post log of the **rep**. Notice that this version includes not only the atomic variant that represents the element inserted by B but also the atomic variant that represents the element inserted by A itself! However, the atomic variant that represents the element inserted by A has value "<dequeued: **nil**>" in the concrete pre-A state. Since an element that has value "<dequeued: **nil**>" is treated as a "non-item" by the semiqueue operations, the element inserted by A will not be in the abstract pre-A value of X. The abstract value of X in the pre-A state, therefore, will correctly contain the single element inserted by B.

In general, the version of a mutex object chosen as a pre-action view of a user-defined atomic object may contain modifications by actions (including ancestors) that are not supposed to be visible in the view. Since we are assuming that atomicity and *display* operations are implemented correctly in user-defined

atomic types, these modifications to the **rep** will not be visible in the abstract value that is displayed as the pre-action view of the atomic object.

In addition to the accesses that must be reflected in the pre-action value, the mutex version we choose to represent the post-value of an action A must also include the modifications of A and descendants that committed up to A.

Definition:

> The post-A value of a mutex object is the latest post-mod version in the mutex's pre-post log that is tagged with either
>
> 1. the *aid* of a action B that is not ancestor-related to A such that $GA_B(A)\# < GA_A(B)\#$, or
> 2. the *aid* of A, an ancestor of A, or a descendant that committed up to A.
>
> If there is no such post-mod version, then it is an error to ask for post-A of the mutex.

Here too, it is possible for the chosen mutex version to contain modifications by actions that are not visible in the post-state of interest. As in the case of pre-states, these modifications will be abstracted away by the type's *display* operation, assuming that both atomicity and the *display* operation are implemented correctly in the type.

Referring again to the example given in Figures 6-2 and 6-3, B's post-mod version will be the one chosen to give the post-A value of the **rep**. The abstract post-value of A will correctly show a semiqueue with two elements, namely those inserted by A and B.

Note that we could use for a pre-action value of a mutex object the same version as that of the post-value, and then depend on the implementation of atomicity and the *display* operation to filter away the modifications that are not supposed to be visible in the abstract pre-action value of the containing user-defined atomic

object.[13] In fact, any post-mod version between the entries we chose for pre-A and post-A will also work. However, we cannot choose a post-mod version, say V, that is after the one we use for post-A because some atomic object that contributes to the abstract pre-A value of the user-defined atomic object might have been removed from V.

6.4.2 Rep with Multiple Mutex Objects

To date, there are very few examples of user-defined atomic types that are built with multiple mutex objects in the **rep**. In all the known examples, the previous subsection's choice of mutex versions when applied locally to each mutex object in the **rep**, continues to give correct pre- and post- action values of the atomic objects. For example, consider the implementation of an *amap* given in Figure 6-4.[14] A map binds names (*uid*s) to objects and provides operations to store, remove, and lookup bindings. An amap is an atomic map. It uses an intentions list (called *intentions*) and a regular (non-atomic) map (called *bindings*) for the database of bindings; both of these data structures are mutex objects in the implementation. *Intentions* holds the recent activities performed on the amap, e.g. insertion of a new binding, and is the data structure through which actions are synchronized. (The main purpose of *intentions* is to reduce the amount of information that must be written to stable storage to make an amap object resilient. The implementation of resilience has no effect on pre- and post- action values of user-defined atomic objects.) To keep *intentions* small, a nested topaction (generated from the *rebuild* procedure) is triggered periodically to trim *intentions* and update *bindings* accordingly.

For example, suppose the following sequence of events occurs at an amap object:

---

[13] In fact, we do use this observation when calculating the values of user-defined atomic objects in the intermediate states of a retrace. See Section 6.5.

[14] The implementation is adapted with some modifications from [Weihl & Liskov 82].

Figure 6-4: Implementation of the Amap Type

```
amap = cluster [vtype: type] is create, insert, delete, lookup

   status = atomic_variant[present: vtype, absent: null]
   table = map[vtype]
   list = map[status]

   rep = struct[intentions: mutex[list], bindings: mutex[table]]

   create = proc () returns(cvt)
      return(rep${intentions: mutex[list]$create(list$create()),
                  bindings: mutex[table]$create(table$create())})
      end create

   insert = proc (m: cvt, u: uid, v: vtype)
            seize m.intentions do
               while true do
                  l: list := m.intentions.value
                  if list$size(l) > 1000
                     then rebuild(m) end     % trim list if too large
                  s: status := list$lookup(l, u)
                     except when not_found: list$insert(l, u, status$make_present(v))
                                            return
                                            end
                  tagtest s
                     wtag present, absent: status$change_present(s, v)
                                           return
                     end % tagtest
                  pause % couldn't lock s; wait and try again
                  end % while
               end % seize of intentions
            end insert

   delete = proc (m: cvt, u: uid) signals (not_found)
            seize m.intentions do
               while true do
                  s: status := find_status(m, u)
                  tagtest s
                     wtag present: status$change_absent(s, nil)
                                           return
                     tag absent: signal not_found
                     end % tagtest
                  pause % couldn't lock s; wait and try again
                  end % while
               end % seize of intentions
            end delete
```

```
lookup = proc (m: cvt, u: uid) returns (vtype) signals (not_found)
        seize m.intentions do
            while true do
                tagtest find_status(m, u)
                    tag present (v: vtype): return(v)
                    tag absent: signal not_found
                    end % tagtest
                pause % couldn't lock status object; wait, and try again
                end % while
            end % seize of intentions
        end lookup

find_status = proc (m: rep, u: uid) returns (status)
        % m.intentions has been seized in caller
        l: list := m.intentions.value
        if list$size(l) > 1000
            then rebuild(m) end    % trim list if too large
        return(list$lookup(l, u))
            except when not_found: end
        s: status
        seize m.bindings do
            s := status$make_present(table$lookup(m.bindings.value, u))
            end except when not_found: s := status$make_absent(nil) end
        list$insert(l, u, s)
        return(s)
        end find_status

rebuild = proc (m: rep)
        % m.intentions has been seized in caller
        l: list := m.intentions.value
        enter topaction % start a new "rebuild" action
            seize m.bindings do
                t: table := m.bindings.value
                for u: uid, s: status in list$pairs(l) do
                    tagtest s
                        wtag present (v: vtype):
                            list$delete(l, u)
                            table$alter(t, u, v)
                        wtag absent:
                            list$delete(l, u)
                            table$delete(t, u)
                        end % tagtest
                            except when unchanged, not_found: end
                    end % for
                end % seize
            end % enter — rebuild action commits here
        end rebuild
end amap
```

1. A *inserts* "x = 1" into the amap
2. A commits
3. B calls the *insert* operation to insert "y = 1" into the amap
    a. B calls nested topaction C to trim the intentions list
    b. C updates *bindings* to map x to 1 and accordingly removes "x = 1" from *intentions*
    c. C commits
    d. B inserts "y = 1" into *intentions*
4. D *inserts* "z = 1" into the amap
5. D commits
6. B commits

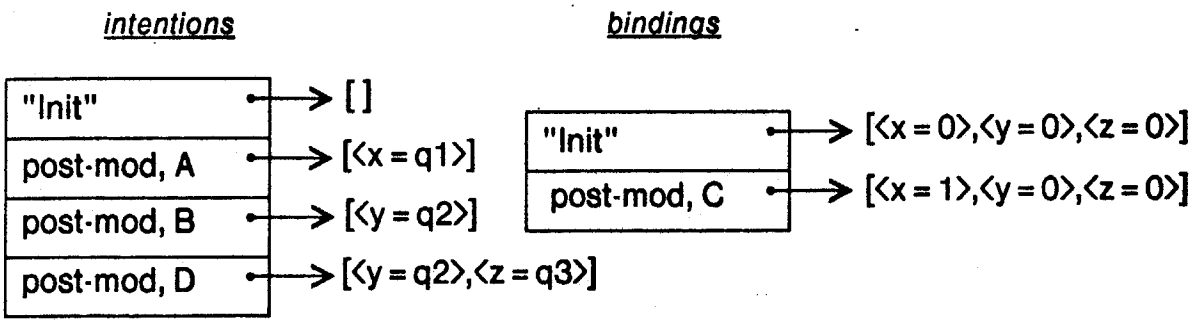
Notice that C removes the binding "x = 1" from *intentions* without directly locking the mutex. Instead, C takes advantage of the fact that its caller B already has a lock on *intentions*. The post-mod version that is created when B releases its lock, therefore, will contain not only the modifications of B but also those of C, as well.

Assume that initially *intentions* is empty, and *bindings* contains [<x = 0>, <y = 0>, <z = 0>], i.e., a map with three bindings, namely "x = 0", "y = 0", and "z = 0". Also assume that A, B, C, and D are topactions. The order of their terminations will serialize them in the order A, C, D, B.

Figure 6-5 shows the pre-post logs of *intentions* and *bindings* that results. By the previous subsection's definition of pre-values of mutex objects

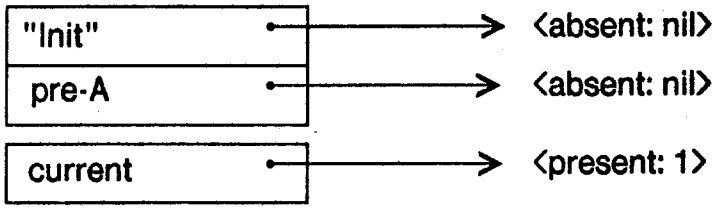
- pre-D of *intentions* is given by post-mod of A and equals [<x = q1>], where pre-D of q1 is "<present: 1>".
- pre-D of *bindings* is given by post-mod of C and equals [<x = 1>, <y = 0>, <z = 0>]

so that the abstract pre-D value of the amap correctly shows a state where x = 1, y = 0 and z = 0. Notice that our algorithm for pre-values picks up the effects of C on *bindings* but not the effects of C on *intentions* when calculating the pre-D value of the amap. This is alright because C's modification on *intentions* has no effect on the value of the amap, once C's modification on *bindings* is taken into account.

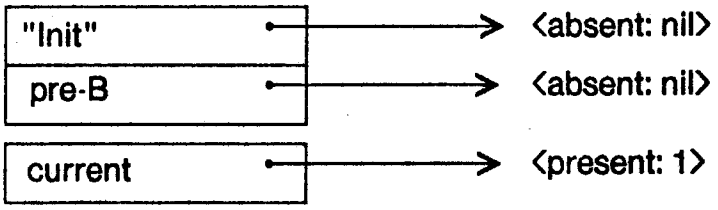


q1, q2, and q3 are *status* objects (i.e. atomic variants) and they have their own history as shown below.

q1:



q2:



q3:

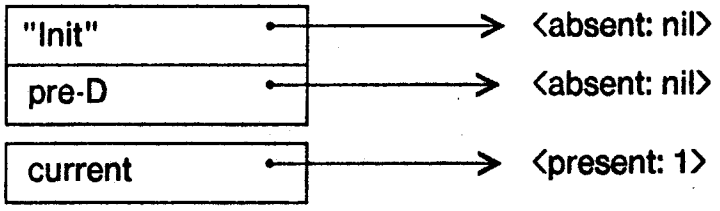


Figure 6-5: Pre-post Logs of an Amap Object

Our algorithm for post-values also gives the correct value for the amap in the post-D state, i.e. [⟨x = 1⟩, ⟨y = 0⟩, ⟨z = 1⟩]:

- post-D of *intentions* is given by post-mod of D and equals [⟨y = q2⟩, ⟨z = q3⟩], where the post-D value of q2 is "⟨absent: **nil**⟩" and the post-D value of q3 is "⟨present: 1⟩"
- post-D of *bindings* is given by the post-mod version of C and equals [⟨x = 1⟩, ⟨y = 0⟩, ⟨z = 0⟩].

6.5 Retracing

In this section, we discuss the use of the pre-post logs in retracing operations at mutex objects.

Recall that each retrace process has an CRV map that gives the current versions of built-in atomic objects in the retrace (Section 5.4.1). We include mutex objects in this CRV map. When a retrace process reads or modifies a mutex's resource, the version that it accesses is always the mutex's CRV in the process's map.

A retrace process initializes a mutex's CRV (*current retracing version*) in its map each time it executes a **seize** of the mutex. The version to which the CRV is initialized is that of the value that held when the **seize** was executed in the original computation. Calling the action that is being retraced A, this value is the latest post-mod version before A's n^{th} entry in the mutex's pre-post log, where n is the number of times the action has **seized** the mutex's lock so far in the retrace (the **seize** that is currently being retraced is included in the count). If A has no n^{th} entry in the log, then the retrace process is done with the retrace of A.

A retrace process, therefore, has to keep tab of the number of times a *retracing-active* action has called **seize** on a mutex object in the retrace. A *retracing-active* action is one that the retrace process has started retracing but has not completed. The retracing process does this by maintaining a *seized map* for

each of its retracing-active actions. An action's *seized map* maps a mutex object to the number of times the action has seized the mutex in the retrace.

6.5.1 Current Values

When the user asks for the current value of a user-defined atomic object, say X, in a retrace, what view should we present to the user? We assume that the retrace is not stopped in the midst of an operation of the user-defined atomic object. Otherwise, the **rep** of X might not be internally consistent, so the current abstract value of X in the retrace will not be meaningful. In any case, if the user stops a retrace in the middle of an operation of a user-defined atomic object, it is probably because he or she wants to debug the implementation of the atomic object's type. It will then be more appropriate to allow the user access to the pre-post logs of objects in X's **rep** and the CRV and *seized* maps of the retrace process. We will discuss the debugging of implementations of user-defined atomic types in the next chapter.

The current abstract value of a user-defined atomic object in a retrace must be the value that reflects the modifications of the serial execution up to the point where the retrace is stopped. We compute this value as follows. If the retrace process is stopped in action S, the version we choose to represent the current retracing value of a mutex object is that of the post-S value. This version may refer to some built-in atomic objects for which an CRV exists in the retrace process's CRV map and others for which it does not. If the CRV exists, we use its value as part of the retracing value of the mutex object. If the CRV does not exist, we use the pre-action value of the atomic object, instead. (Current values of built-in atomic objects in a retrace are discussed in Section 5.4 of Chapter Five.)

For example, suppose the following sequence of operations is executed at an initially empty semiqueue object X:

A *enq*s 1
A *enq*s 2

Figure 6-6 shows a snapshot of the history and relevant CRVs in X for a retrace that is stopped just before A's second *enq*. To calculate the current value of X in the retrace, we use A's second post-mod version since it is the post-A value of the **rep**. This version refers to both q1 and q2. An CRV exists for q1, so we use it. An CRV does not exist for q2, so we use the pre-A value here. Thus the current concrete value of X in the retrace is

[current retracing value of q1, pre-A value of q2]
= [<enqueued: 1>, <dequeued: **nil**>]

Therefore, the current abstract value of the semiqueue in the retrace has 1 as its sole element. (Recall that elements that have value "dequeued: **nil**" are non-items in a semiqueue.)

If the retrace is stopped after A's second *enq*, the concrete state of the semiqueue now becomes

[<enqueued: 1>, <enqueued: 2>].

This is because the current retracing value of q2 has changed in keeping with the retrace's progress. The value of the semiqueue that is displayed, therefore, will show both of A's insertions.

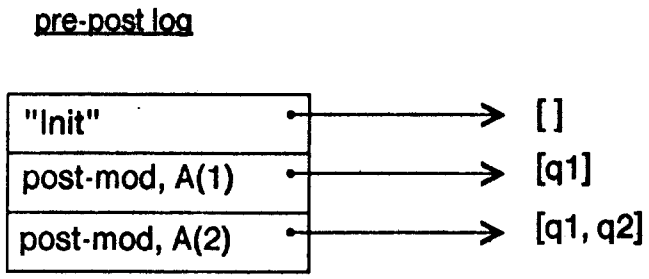
Notice that the CRV of a mutex object does not play a role in the computation of the current abstract value of a user-defined atomic object: we always use the post-A version of the mutex object, regardless of where the retrace is stopped in A.

6.6 Effects of Crashes

In a crash, all versions in a pre-post log will be lost. To help the calculation of pre- and post- values of mutex objects for actions that run immediately after a crash, we can use a scheme analogous to that used for built-in atomic objects (Section 4.9).

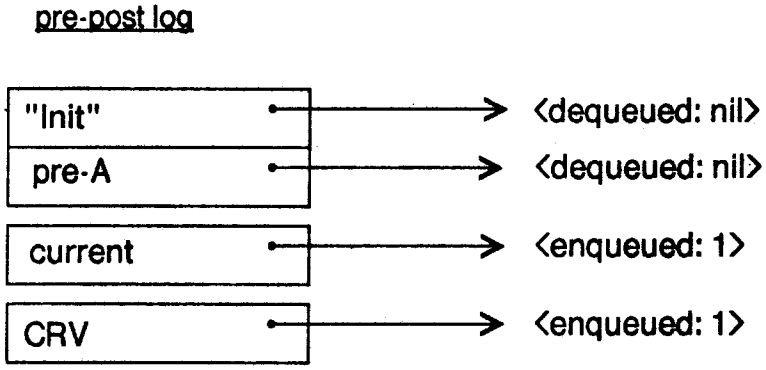
Recall that a topaction's termination number is saved in its *prepared* record on stable storage, if the action has a *prepared* record. When the system recovers a mutex object after a crash, we require that an "Init T#" be created in the mutex's

buffer (rep of the semiqueue object):



q1 and q2 are qitems (i.e. atomic variants).

q1:



q2:

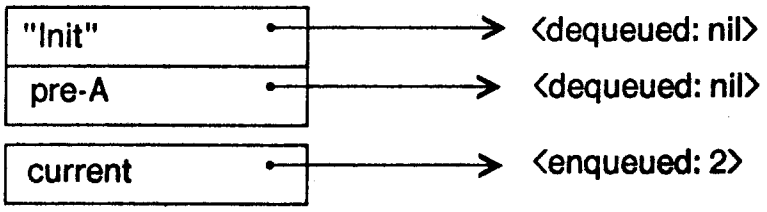


Figure 6-6: Example Illustrating the Current Value in a Retrace

pre-post log, where the mutex value that is recovered from stable storage is the one that was written on behalf of topaction T. (We will discuss the use of mutexes for resilience in the next chapter.) The "Init T#" will point to a copy of the recovered value.

This "Init" entry will be treated exactly like the object creation's "Init" to give pre-values of actions that run after the crash. (See Section 4.9 for details of how termination numbers and crash counts can be used to distinguish actions that run after the crash from those that run before the crash.)

6.7 Reclaiming Storage Space

In Section 4.10, we presented our scheme for reclaiming storage from saved history. In the scheme, the saved history of handler actions and topactions with earlier termination numbers are removed before the saved history of handler actions and topactions with later termination numbers. Also, a-records of a topaction or handler action and its local descendants are removed together.

We propose to reclaim entries from a mutex's pre-post log as follows. When A's a-record is discarded, for each mutex object M that was modified by A, we remove as many entries from M's pre-post log as possible without violating either one of the following two conditions:[15]

1. An entry must be removed before another that comes after it in the log.
2. Only entries that are tagged with *aid*s of actions whose a-records have been discarded can be removed.

The first condition is necessary because our rules for deducing action views assume that there is no gap in the log. The second condition prevents the retrace process from initializing a mutex's CRV with the wrong value when a **seize** is retraced. Recall that the retrace process matches the n^{th} **seize** by an action with the

---

[15] This removal of entries can be done either *lazily* or *aggressively*. See Section 4.10.

action's n^{th} entry in the log to determine the value of the mutex at the **seize**. If any of an action's entries is missing, the action's n^{th} entry in the log will not correspond to the action's n^{th} **seize**. We also note that only actions whose a-records are still available can be retraced.

For an example, consider the following pre-post log of a mutex object:

post-mod, A(1)
post-mod, B
post-mod, A(2)
post-mod, C

Suppose that A, B and C are topactions and that A# < B# < C#. If A's a-record is removed, only A's first post-mod entry can be reclaimed because B is still retraceable. When B's a-record is removed, "post-mod, B" and "post-mod, A(2)" can now be removed.

Our scheme for reclaiming pre-post entries has a minor problem. When the last post-mod of an action T is removed, we will not be able to determine the pre-values of actions that are serialized after T and that are dependent on T's post-mod entry. For instance, in the example above, we will not be able to deduce the pre-values of actions that are serialized after B but before C, once "post-mod, B" and "post-mod, A(2)" are removed. (Note that we can use "post-mod, C" as the pre-C value, even though it is not the version that our definition would choose. This is because modifications by C are not visible in the abstract pre-C value of the containing user-defined atomic object, assuming that atomicity and the *display* function are implemented correctly.)

We propose the following amendment. When the last entry that belongs to a descendant of a topaction T is removed, we replace the entry, say E, with an "Init T#". The "Init T#" will refer to the same version as E. An "Init T#" can only be

removed if another "Init" entry can be created to replace it, i.e., if enough entries can be removed, together with the "Init T# ", so that an "Init" can be created for another topaction. Note that if an "Init" entry cannot be removed, then no entry beyond the "Init" can be removed.

In the example above, an "Init A# " will be created when "post-mod, B" and "post-mod, A(2)" are removed. This entry will serve to give the pre-values of actions serialized between B and C.

6.8 Summary and Discussion

In this chapter, we considered how the debugging method can continue to work when we allow actions to share user-defined atomic objects, in addition to those that are built-in atomic. We began by arguing that if user-defined atomic types are to synchronize concurrent accesses properly, they must have reps in which all mutable objects are either built-in atomic or mutexes. This observation allows us to concentrate on state changes at mutex objects.

Because a mutex object synchronizes only processes and not actions, we found that we have to save a post-mod version when a mutex lock is released if the resource is modified in the **seize**. The compiler can sometimes tell when a mutex's resource is not modified within a **seize**; in this case we simply save a "placeholder" in the pre-post log, and not a copy of the resource. Otherwise, we assume the worst and save a post-mod version.

We showed how the recorded post-mod versions can be used to present the pre- and post- action values of a user-defined atomic object. This presupposes that each type has a correctly implemented *display* operation that serves as its abstraction function. The next chapter will further exploit these recorded versions to help the user debug the implementations of user-defined atomic types.

We also showed how the post-mod versions can be used to retrace **seizes** and

to present the current retracing values of user-defined atomic objects. Finally, we discussed methods for minimizing the impact of crashes and storage reclamation on our ability to provide pre- and post- action values of user-defined atomic objects, and to retrace operations at these objects.

We stress that the chapter's algorithms for pre- and post- action values and retracing apply not only to committed actions but aborted ones, as well. We also note that the ability to calculate the pre-A value of a built-in atomic object that was not read or modified by action A is used quite heavily to present A's views of user-defined atomic objects.

Chapter Seven

Debugging User-Defined Atomic Types

There is a bug in the implementation of a user-defined atomic type T if the map

$$\text{pre-A value of X} \xrightarrow{op_1 \ \cdots \ op_n} \text{post-A value of X}$$

contradicts behavior that is expected of X, where X is an object of type T and op_1 ... op_n is the sequence of operations called by action A on X. Note that the map is considered incorrect not only if the post-A value of X is unexpected but also if the results returned by any of the op_i are unexpected, given pre-A of X and A's operations before op_i. The user may have to retrace A to know the operations that A called on X and the results returned by each of the operations.

An incorrect pre-A of X to post-A of X map can be caused by either a sequential or atomicity bug in the implementation of X's type. An implementation has a sequential bug if it fails to satisfy its specification even when there is neither concurrency nor node crashes.[16] Being able to retrace A and to breakpoint and examine the retracing structures of the object will help the user to isolate sequential bugs.

There is an atomicity bug if actions that use X become non-serializable. In Section 7.1, we discuss what the user has to do to isolate an atomicity bug in the implementation of a user-defined atomic type, given an incorrect map. History that is already saved for supporting pre- and post- action views is used.

In Section 7.2, we explain the support that the debugging system can provide to help the user test for and isolate resilience bugs in the implementation of a

---

[16]Notice that a bug in the type's *display* operation (i.e., abstraction function) is a sequential bug.

user-defined atomic type. A resilience bug occurs when what was stored in stable storage is not equivalent to the most recent value that was written into X by a committed topaction. A resilience bug is revealed only if there is a crash and subsequent recovery of X's guardian; a resilience bug is not manifested as an incorrect pre-A of X to post-A of X map. As with atomicity bugs, the user isolates a resilience bug using versions saved in the pre-post logs of mutex objects.

7.1 Atomicity Bugs

In this section, we continue to assume that *display* functions are implemented correctly. This assumption allows the user to believe the abstract pre- and post-values that are displayed and to concentrate instead on sequential bugs that concern incorrect use of the **rep** and on atomicity bugs. We discuss isolation of atomicity bugs in this section.

An incorrect pre-A of X to post-A of X map does not necessarily imply that the problem is within A, i.e, with the code that A ran. In fact, the culprit action that caused the map to be incorrect can be some other action B that accessed X concurrently with A. Below, we explain how the culprit action and the kind of atomicity bug it exposes can be determined.

We have argued that all mutable objects in a properly implemented user-defined atomic type should be either atomic objects or mutexes (Section 6.2). Atomicity bugs, therefore, are always attributable to incorrect modifications of mutexes in the **rep**, assuming of course that all atomic objects in the **rep** are implemented correctly.

The user has to examine only a specific region in a mutex's pre-post log to determine whether an incorrect map is due to an atomicity bug. From the definitions of pre-A and post-A of a mutex object given in the previous chapter

1. the entry chosen for pre-A must be before or equal to the entry chosen for post-A in the mutex's pre-post log, and

2. the entry that holds the version that was read by A's first access, if any, must be before or equal to the entry chosen for post-A.

These three entries can be related in two representative ways, as given in Figure 7-1. For example, part (a) of the figure will result if an action C that is serialized before A accessed the mutex after A. Part (b), on the other hand, corresponds to the case where all actions D such that $GA_D(A).\# < GA_A(D).\#$ either did not access the mutex or accessed it before all of A's accesses.

For an action B to contribute to an incorrect pre-A of X to post-A of X map, B must run concurrently with A and have modifications at a mutex object that interleave with A's accesses. More specifically, if B does not have a modification in Region 1 or 2 (see Figure 7-1) of any mutex object in X, then B cannot cause the pre-A of X to post-A of X map to be incorrect. In other words, if no action other than A (and descendants that committed up to A) modified mutexes of X in the mutexes' respective Regions 1 and 2, then an incorrect pre-A of X to post-A of X map must be due to either a sequential bug in A's code or an atomicity bug in the code of A's concurrent descendants.

There are two kinds of bugs that the user should look for if there are accesses by actions other than A and its descendants in Region 1 or 2. These are *recovery* and *invalidation* bugs.

7.1.1 Recovery Bugs

A recovery bug might have contributed to the noticed incorrect pre-A of X to post-A of X map if there is a non-recoverable modification in Region 1 or 2 of a pre-post log in X. A modification by an action B is non-recoverable if the abstract pre-B value of X *just before* B modifies the mutex object does not equal the abstract pre-B value of X *just after* the modification.

The abstract pre-B value of X just before a modification of a mutex object M in X (written *pre-B(before modification) of X*) is given by a **rep** whose value is composed as follows:

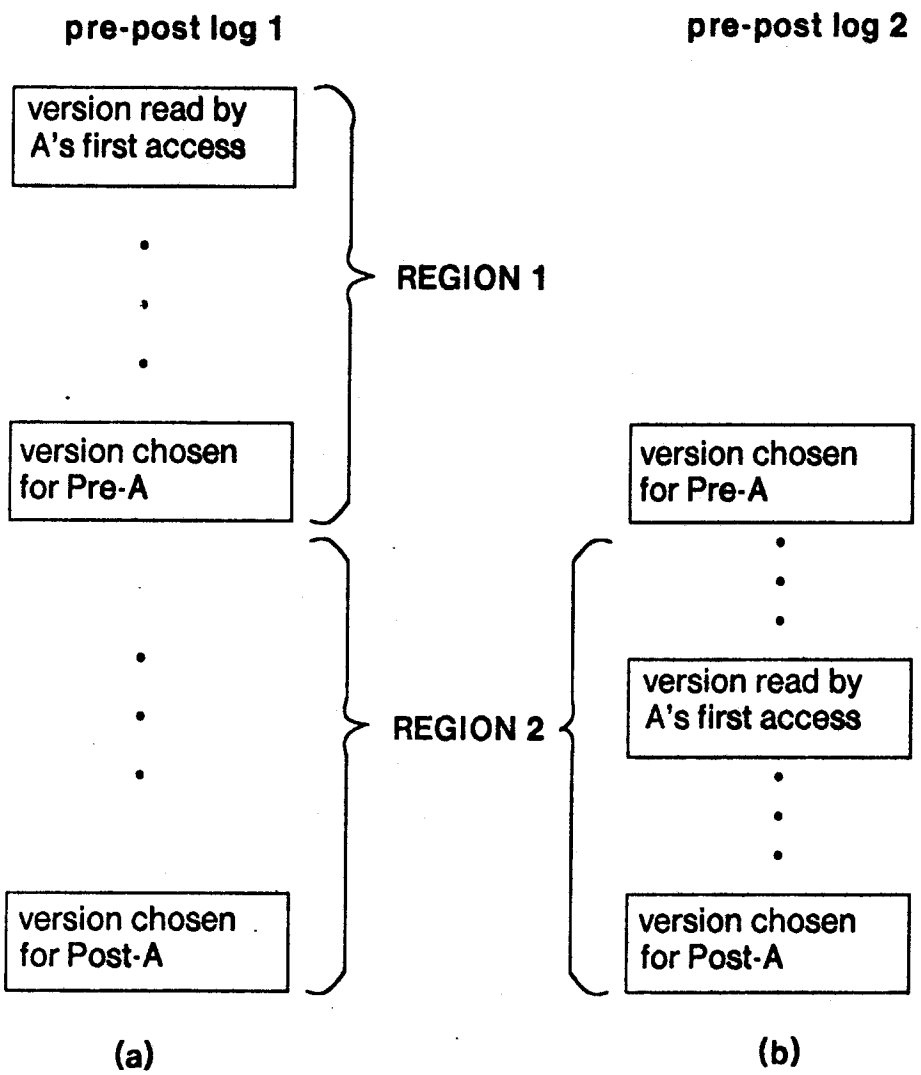


Figure 7-1: Pre-Post Logs Showing Relationship between Versions

1. the value of M is given by the version just before the modification,
2. the values of all other mutex and atomic objects in the **rep** are given by their pre-B values.

The abstract pre-B value of X just after the modification (written *pre-B(after modification) of X*) is given by a **rep** whose value is the same as that of pre-B(before modification), except the version just after the modification is used as the value of M instead of the version just before the modification.

For example, consider B's insertion of the element 1 into a semiqueue[17] object X that is initially empty. The concrete value of pre-B(before B's *enq*) is the empty array. The concrete value of pre-B(after B's *enq*), however, is the singleton array

[Pre-B value of the qitem inserted by B]
= [<dequeued: **nil**>].

Since qitems that have value "<dequeued: **nil**>" are non-items in the abstract values of semiqueues, the abstract pre-B(after B's *enq*) value of X therefore is also the empty queue. Thus, B's *enq* is recoverable.

We now explain how non-recoverable modifications in Regions 1 and 2 of a pre-post log can contribute to an incorrect pre-A of X to post-A of X map.

[1] The effects of a non-recoverable modification in Region 1 are visible in the abstract pre-A value of X. These effects may cause the abstract pre-A value to contradict the values read by by A (or descendants that committed up to A) before the modification and thus, contribute to the noticed inconsistency in the pre-A of X to post-A of X map.

For example, consider a semiqueue that is implemented (incorrectly) with integers as qitems. Call this incorrect implementation *NA_semiqueue*. The *enq* operation of *NA_semiqueue* inserts qitems into the array that is the object's **rep**; the *display* operation returns an abstract semiqueue that contains the abstract values of

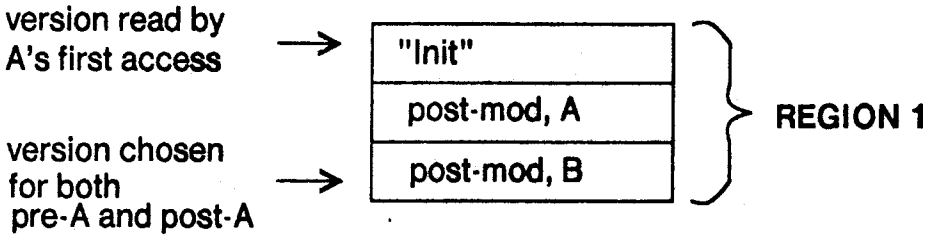
[17] The semiqueue type was introduced in the previous chapter.

the qitems in the **rep**. Thus, all modifications of the mutex object that is the **rep** of a NA_semiqueue object are non-recoverable. (Other details about *NA_semiqueue* can be ignored for our purposes.)

Now, suppose that the following sequence of operations is made at an initially empty NA_semiqueue object X:

A *enq*s 1
B *enq*s 2
B commits
A commits

The pre-post log of X's **rep** that results is as follows:



Assuming that A and B are topactions, their termination order will serialize them in the order B, A. The pre-A (and post-A, as well) value of X's **rep** is given by the post-mod version of B. So,

rep of pre-A of X = [1, 2], and

rep of post-A of X = [1, 2]

Notice that the element A enqueued is included prematurely in pre-A of X precisely because A's modification is not recoverable. Therefore, the pre-A of X to post-A of X map will wrongly show that A "lost" its *enq* of 1. The bug that caused the noticed incorrect map in this case is in the code that A ran.

[2] A non-recoverable modification in Region 2 by an action that is not A or a descendant that committed up to A can make the pre-A of X to post-A of X map incorrect in two ways:

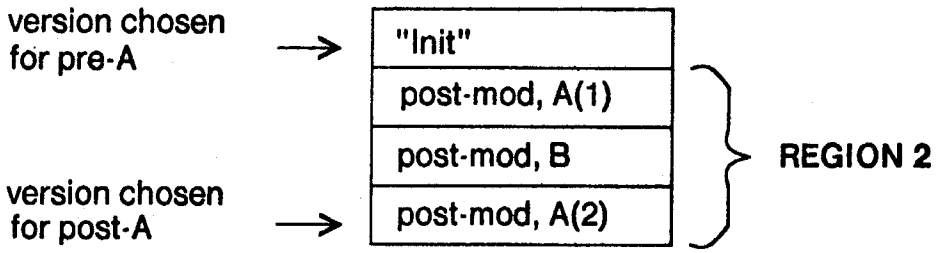
1. the modification is not reflected in the abstract pre-A value of X but is reflected in the abstract post-A value of X. In this case, the modification will be wrongly attributed to A,

2. the modification affects subsequent accesses by A (and descendants that committed up to A).

Take for example, the following sequence of operations at an initially empty NA_semiqueue object X:

A *enqs* 1
B *enqs* 2
B aborts
A *enqs* 3
A commits

The pre-post log of X's **rep** that results from the operations is given below:



The pre- and post- access versions that record B's ill-fated insertion of 2 are between the versions chosen for pre-A and post-A in the pre-post log of X's **rep**. Since B's modification of the mutex object is not recoverable,

rep of pre-A of X = [ ]

rep of post-A of X = [1, 2, 3]

so that A is shown wrongly as having enqueued 1, 2, and 3 into X.

Notice that even though A has two modifications in Region 2 that are not recoverable, they do not contribute at all to the incorrect pre-A of X to post-A of X map. This is true in general: non-recoverable modifications by A (and descendants that commit up to A) in Region 2 do not by themselves explain an incorrect pre-A of X to post-A of X map.

Finally, we note that the user need not retrace any action when checking for recovery bugs in Regions 1 and 2.

7.1.2 Invalidation Bugs

There is an invalidation bug if a modification by an action B that is visible to A contradicts the results of a prior access by A or a descendant that commits up to A. A modification contradicts a read access if the value that is read would have been different if the modification had happened before the read. A modification contradicts a write access if the resulting state of the object would differ if the order of the modification and the write access is reversed.

Note that there cannot be an invalidation bug in Region 2 that could have caused an incorrect pre-A of X to post-A of X map since, by our choice of mutex version for pre-A, the region cannot contain modifications by actions whose effects are visible to A. So, invalidation bugs, if any, are manifested only in Region 1.

For example, consider the following scenario that involves two topactions A and B:

1. A asks for the binding of z at an amap[18] object X and was told that z has no binding.
2. B calls *insert* at X to bind z to 1.

If the amap implementation allows B to complete its *insert* of the binding "z = 1" at X before A terminates, it is possible for B to go on to modify some built-in atomic object Y and then commit, and for A to read B's modification at Y. As a result, A and B become non-serializable with respect to each other. The implementation was wrong to allow B to contradict A's view of X before A terminated.

To decide whether a modification by B contradicts the results of any prior access by A and descendants that commit up to A, the user has to know which parts of X's value were actually seen by the operations that made these prior accesses, and which parts of X's value were superfluous to the operations. In general, the user has to retrace A and its descendants to figure this out.

---

[18] An amap is a user-defined atomic map; it was discussed in the previous chapter.

The user, however, can determine the change to the abstract value of X that is caused by B's modification without needing to retrace B. This change is given by the "difference" between the abstract post-B value of X just after the modification and the abstract post-B value of X just before the modification. The abstract post-B values of X just before and just after a modification of a contained mutex object are defined in a way that is similar to the abstract pre-B values of X before and after a modification.

> The abstract post-B value of X before/after a modification of a contained mutex object M is given by a **rep** whose value is constructed as follows:
>
> 1. the value of M is given by the version just before/after the modification,
> 2. the values of all other mutex and atomic objects in the **rep** are given by their post-B values.

For example, if B *enq*s 1 and then 2 into an initially empty semiqueue object X, then

> **rep** of post-B(before B's second *enq*) = [<enqueued: 1>]
>
> **rep** of post-B(after B's second *enq*) = [<enqueued: 1>, <enqueued: 2>]

So, B's second *enq* caused X's contents to change from a single element 1 to two elements, 1 and 2.

7.2 Resilience Bug

There is a resilience bug if the value of an atomic object that is recovered from stable storage after a crash is not equal to the object's most recent top-level committed value. Unlike atomicity and sequential bugs, the user has to anticipate and check for a resilience bug *before* it manifests itself. Otherwise, he or she will not have the benefit of the history in the pre-post logs to track the bug down; saved history, being in volatile memory, will be lost by the time the user realizes that the state recovered from stable storage is not that at the end of the last committed topaction.

We first describe how mutex objects are used to make user-defined atomic objects resilient. We then present a scheme that takes advantage of the versions saved in pre-post logs to test for and isolate resilience bugs.

7.2.1 The Role of Mutex

In addition to process synchronization, mutex objects also serve to help make user-defined atomic objects resilient to crashes. A programmer can ask explicitly that the resource of a mutex object be written into stable storage by executing the mutex's **changed** operation. When so notified, the system will write the value of the mutex's resource to stable storage sometime between the call of **changed** and the commit of the topaction ancestor of the action that invoked **changed**. After a crash, a mutex object that is reachable from a stable variable will be reinstated and restored to the value that was last written to stable storage on behalf of a topaction that prepared (i.e., completed the first phase of the commit protocol) at the guardian. Note that it does not matter whether or not the action that prepared did in fact commit.

The system synchronizes with user processes before writing a mutex's resource to stable storage. It does this by acquiring the mutex's lock before copying the resource. This ensures that the resource will be in an internally consistent state when written into stable storage. The system writes **changed** mutex objects to stable storage one at a time; one mutex's lock is released before another is acquired. When writing a mutex's resource to stable storage, the system copies down to contained atomic and mutex objects. This is just like the copy that is done when versions are created for a mutex's pre-post log, only in the case of pre-post logs, the copies are kept in volatile memory.

For an example of the use of **changed**, let us study the semiqueue again. The definition given in Figure 7-2 differs only in one respect from the definition we saw earlier in Figure 6-1 of Chapter Six. In the *enq* operation, there is now a call to **rep$changed** after an item is appended to a semiqueue's *buffer*. This is because

the new state of the (non-atomic) buffer must be written into stable storage before the action that invoked *enq* commits past its topaction ancestor. Otherwise, a crash of the guardian, after the topaction ancestor commits but before the buffer is written into stable storage, will lose the inserted item.

7.2.2 Isolating a Resilience Bug

In this subsection we propose a scheme for testing and isolating resilience bugs. We begin with an enumeration of the additional information that needs to be saved and then we discuss the method for using the saved data.

We save additional information to help us determine the value of a mutex object that will be recovered from stable storage if the guardian should crash immediately after a topaction *prepared*. The information helps by relating the time a *prepared* record is written into stable storage to the mutex versions that were then already in stable storage. The additional information is as follows:

[1] We introduce a counter (called a *resilience clock*) into each guardian. Each time the counter is read, it is automatically incremented after the read. The read-and-increment is done indivisibly.

We call a value that is read from the counter a *resilience time*.

[2] We keep a new log (called a *prepare-time* log) in each guardian. The log maps topactions to the resilience time at which their prepared records were written into stable storage in the guardian. So, when a prepared record is written into stable storage on behalf of topaction T, we read the resilience clock and append to the prepare-time log the mapping of T to the value that was read.

The resilience time that is associated with a topaction in the prepare-time log is called the topaction's *prepared time*.

[3] When the system seizes a lock to write a mutex object to stable storage:

Figure 7-2: Implementation of the Semiqueue Type

```
semiqueue = cluster is create, enq, deq

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

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

   enq = proc (q: cvt, i: int)
            item: qitem := qitem$make_dequeued(nil)     % dequeued if action aborts
            qitem$change_enqueued(item, i)              % enqueued if action commits
            seize q do
               buffer$addh(q.value, item)               % add new item to buffer
               end
            rep$changed(q)                              % notify system of modification
                                                        % to buffer
            end enq

   deq = proc (q: cvt) returns (int)
            cleanup(q)
            seize q do
               while true do
                  for item: qitem in buffer$elements(q.value) do
                     % look at all items in the buffer
                        tagtest item
                           % for an item that can be dequeued by this action
                        wtag enqueued (i: int): qitem$change_dequeued(item,nil)
                                                return(i)
                        end % tagtest
                     end % for
                  pause
                  end % while
               end % seize
            end deq
```

```
    cleanup = proc (q: rep)
              enter topaction                    % start an independent action
                 seize q do
                    b: buffer := q.value
                    for item: qitem in buffer$elements(b) do
                       tagtest item              % to remove items in the dequeued state
                          tag dequeued: buffer$reml(b)
                          others: return
                          end % tagtest
                       end % for
                    end % seize
                 end % enter — commit cleanup action here
              end cleanup

end semiqueue
```

---

1. we create a *resilience placeholder* in the mutex's pre-post log to point to the same object as the latest post-mod version in the log.[19] (Note that this latest post-mod version has the same value as the current version of the mutex.)
2. we label the placeholder with the *aid* of the topaction on whose behalf this write to stable storage is being done.

With the saved data, we can deduce the value of a user-defined atomic object X that would be recovered after a crash, without depending on the stable storage subsystem.

The version of a mutex object M that is recovered if there is a crash immediately after a topaction A prepares at the guardian can be computed as follows:

1. Let RP be the resilience placeholder that is the latest in the pre-post log to be labelled with the *aid* of a topaction whose prepared time is less than or equal to A's prepared time.
2. The version that is recovered is the version in RP.

This is because RP represents the last value that was written into stable storage on

---

[19] This placeholder is not to be confused with the read placeholders of the previous chapter.

behalf of a topaction that prepared at the guardian before the crash we are simulating.

So, the **rep** of a user-defined atomic object X that is recovered if there is a crash immediately after a committed topaction A prepares at the guardian can be constructed as follows:

1. For each mutex object in X, use the version as computed above.

2. For each built-in atomic object in X, use the post-A value of the object.

We call the abstract value of X derived from this **rep** the abstract *stable post-A* value.

There is a resilience bug in the implementation of a user-defined atomic type if the abstract stable post-A value of X is not equal to the abstract post-A value of X, for some committed topaction A and object X of the type. (The abstract post-A value of X is as defined in the previous chapter.)

But once the user has determined that there is a resilience bug, how does he or she isolate it? We give three guidelines below.

[1] A resilience bug may in fact be due to recovery problems. For example, A modifies a mutex object M and then calls **changed**. However, before the system writes M into stable storage, another action B that is serialized after A modifies M. If B's modification is not recoverable, then the abstract value of the user-defined atomic object that is saved into stable storage will not have the same value as post-A.

So, if the stable post-A version in a pre-post log is after the post-A version, the user should verify that all of the modifications in between the two versions in the log are recoverable.

[2] If A modifies a mutex object M and does not call **changed** after the modification, the modification must be "benevolent", i.e. have no effect on the abstract value of the user-defined atomic object.

So, if the stable post-A version in a pre-post log is before the post-A version, the user should verify that all modifications between the two versions in the log that were made by A or a descendant that committed up to A or an action that is visible to A have no effect on the abstract value of the user-defined atomic object. I.e., for each modification by such an action B, make sure that the abstract post-B value of the user-defined atomic object just before the modification is equal to the abstract post-B value just after the modification. If there is a problem, then the bug is with the code run by the action that made the unexpectedly non-"benevolent" modification.

[3] If the stable post-A version in a pre-post log is before the post-A version, the inequality between the abstract stable post-A and post-A values of the containing user-defined atomic object might not be explained by either of the above two checks. The inequality might in fact be due to a non-recoverable modification made in between the stable post-A and post-A versions by an action that is not A or a descendant that committed up to A.

So, the user should check to make sure that all modifications in between the stable post-A and post-A versions made by an action that is not A or a descendant that committed up to A are recoverable.

Note how sensitive a resilience bug is to when a mutex value is written into stable storage in the guidelines above. Therefore, the user should have control over when a mutex value is written to stable storage if he or she is to test that resilience is implemented correctly. (Of course, the user must be constrained to the range between an action's call of **changed** and when the action prepares at the guardian.)

The algorithm for reclaiming space in a pre-post log should treat a resilience placeholder as a post-mod version of the placeholder's topaction. The algorithms for computing pre- and post- action values, on the other hand, should ignore these resilience placeholders and not treat them as post-mod versions.

Space in a guardian's prepare-time log can be reclaimed in much the same

way as space in a mutex's pre-post log, i.e., earlier entries must be removed before later ones and no entry that belongs to a topaction T such that T# is younger than the cut-off threshold (see Section 4.10) can be removed.

7.3 Summary and Discussion

In this chapter, we discussed how the user can use the versions saved in a mutex's pre-post log to isolate an atomicity bug in the implementation of a user-defined atomic type. We also discussed how modest additions to the history saved in a mutex object can aid the user in finding resilience bugs. Since saved history is kept in volatile memory, our scheme requires that a resilience bug be anticipated and tracked before it shows itself (after a crash).

Actually, what the algorithms isolate is a particular incorrect access of a mutex object, e.g., a non-recoverable modification. Often this is enough for the user to pin-point the atomicity or resilience bug in the code. If not, the user will have to retrace the operation that made the incorrect access. This requires retracing the action that called the operation.

The debugging system has to allow the user access to the pre-post logs of mutex objects and the data structures of a retrace process when he or she is debugging the implementation of a user-defined atomic type. In particular, the user needs to examine and reference versions in a pre-post log when tracking a recovery bug. The user also may need to know which versions in a pre-post log are used by a retraced operation when tracking an invalidation bug.

The user has to be able to use the *display* operation of a user-defined atomic type to get a committed topaction's abstract stable post-value of an object X of the type, and the abstract pre-A and post-A values of X just before and just after a modification of a mutex object in X by an action A. The user should not have to perform the abstraction function himself/herself.

Notice that the algorithms for tracking atomicity and resilience bugs require the user to know whether one action is visible to another. The debugging system can help by answering such queries from the user.

Finally, we stress that where there are multiple mutexes in the **rep** of a user-defined atomic object, the user looks for atomicity and resilience bugs by examining the mutex objects one at a time; he or she does not relate the versions of one pre-post log to those of another. This "local" isolation of bugs works for all known examples of implementations of user-defined atomic types with multiple mutexes in their **reps**. However, whether it does indeed work generally in Argus will have to await further experience in building user-defined atomic types.

Chapter Eight

Conclusions

The atomic action abstraction is gaining popularity as a tool for dealing with complexity introduced by concurrent processing of long-lived data in a distributed system. With atomic actions it is easier to structure programs to constrain possible interactions among anticipated activities and to limit the effects of partial failures.

In this thesis, we have argued that atomic actions are also beneficial for debugging. We showed that properties of atomic actions help a person who is debugging inspect and repeat a concurrent and distributed computation much like he or she would a sequential computation.

8.1 Summary

We presented a method for debugging nested atomic actions in this thesis. The method is applicable to other action systems since it depends only on the atomicity properties of actions.

In our method, the user is able to repeat an action of interest. In this section, we first describe how the debugging system supports this retrace of an action. The history that is saved for retracing can also serve to help the user isolate the action that exposes the bug. We describe this phase of our method in the latter part of this section.

Our debugging system supports the retrace of actions by saving a limited amount of history as an action runs. It uses this saved data to re-execute code and create an equivalent serial execution, as needed. We stress that only one locus of control is used in retracing an action's history; concurrent subactions are retraced in

serialization order. The user can use standard debugging tools, such as breakpoints and single-stepping, on a retrace to isolate a bug.

The *pre-* state of an action is well-defined, i.e., when an action reads an atomic object twice it is guaranteed to see the same value if it does not modify the object in between the reads. Therefore, an action's pre-state, together with the action's tree of subactions and a serialization order of siblings in the tree, is sufficient to repeat an action. In the following paragraphs, we describe the implementation of a serialization order and of the pre- and post- action values of objects.

Our debugging system timestamps the termination of actions with Lamport clocks [Lamport 78]. From this termination order, we derive a serialization order for siblings and topactions.

We save multiple versions of built-in atomic objects and mutexes. We use the serialization order and the nesting of actions to determine the version that represents the value of an object in a pre- (or post-) state. Our scheme supports pre-values at an object even for actions that did not use the object. This capability is critical to our method for providing pre- and post- action values of user-defined atomic objects. In addition, it is sometimes useful to be able to see what an action would have read if it had accessed an object.

Usually, the versions saved in a built-in atomic object are recovery versions that are created by the run-time system to prepare for action aborts. These are available without cost to the debugging system. Recovery versions have been used for concurrency control; our work, however, is the first to use them for debugging.

Versions in a mutex object are created after every modification of the mutex's value. These versions help not only in retracing but also in detecting and isolating atomicity and resilience bugs in the implementations of user-defined atomic types.

Our debugging system does not save environments. Instead, we depend on

the fact that a handler action's environment is always available, as long as its guardian has not terminated. This is because a handler action's environment consists only of guardian variables. Therefore, retracing must begin with a handler action in our system.

With the availability of pre-states, it is not much more work for our debugging system to also support *post-* states: the value of an object in the post-state of A is the value in the pre-state of the action serialized immediately after A. With pre- and post-states, the user can make an initial pass through the computation to narrow a bug to a faulty handler action before retracing; there is often no need to retrace all of a computation. An action is faulty if it maps a correct pre-state to an incorrect post-state. It is up to the user to sample the right objects in an action's pre- and post-states to decide correctness. Inasmuch as a guardian variable is also part of the environment of in-line subactions, the user may even be able to narrow a fault to a subaction within a handler action.

We note that aborted actions are of interest to the person who is debugging. For example, he or she may want to know why calls of a particular handler seem to abort with alarming frequency. Our debugging system supports pre- and post- states for aborted actions, and retracing as well.

8.2 Further Work

In this section, we discuss briefly four areas for further work. These are

1. implementing our debugging method in Argus,
2. testing alternative mechanisms for building user-defined atomic types,
3. implementing our debugging method in other action systems, and
4. developing methods and systems for other debugging-related tasks.

Implementation.

The debugging system presented in this thesis needs to be implemented so we can verify the feasibility of our design. There are two questions of particular interest:

1. Can our method be supported in Argus without penalizing the run-time of a computation intolerably?

   We feel that the answer to this question is yes if only built-in atomic objects are used. This is because the history that needs to be saved consists almost entirely of recovery versions, a universal action tree, and termination numbers.

   We are not as sure about the answer if user-defined atomic objects are used. The versions that are saved at mutex objects are much more costly: the debugging system has to make and save a version after every modification.

2. In our system, old recorded history is discarded to make room for newer versions and action records. Therefore, there is a time window within which a computation must be debugged. Can this window be kept reasonably wide for typical applications and primary storage allocations?

   If the answer to this question is no, alternatives will have to be explored. These include 1) use of secondary storage to widen the window, and 2) selective archiving of recorded history for later examination. Selective archiving may be triggered automatically when signs of abnormality are recognized.

User-defined atomic types.

As noted aboved, it is expensive to support the debugging of programs that use user-defined atomic objects in Argus. Work should be done to find out whether any of the proposed alternatives to Argus's mechanism for implementing user-defined atomic types, e.g., those in [Weihl 84], [Schwarz & Spector 84], [Allchin & McKendry 83], and [Herlihy 84], can support our debugging method more cost effectively.

Generality of the method.

Our debugging method is also applicable to other action systems since it depends only on properties of actions. However, it is not clear how easy it is to implement the method in these other systems. We make some preliminary observations below.

Our method should translate into a system that uses time-domain concurrency control such as Reed's [Reed 78, Reed 83] quite easily. Versions, timestamps, and a method to derive action views of objects are already available in such systems.

Timestamps and recovery versions are also used in systems that use optimistic concurrency control such as Kung & Robinson's [Kung & Robinson 81]. Therefore, we believe that our debugging method can also be incorporated into such systems in a straightforward manner. However, there are complications with aborted actions. An aborted action can see inconsistent views in systems that use optimistic concurrency control. As a result, values that are read will have to be saved explicitly for aborted actions. In addition, we cannot predict pre-action values at objects that an aborted action did not use.

The above two systems, namely Reed's and Kung and Robinson's, use recovery versions, as does Argus. It will be interesting to see how pre- and post-action values might be computed in systems that employ alternative recovery techniques, e.g., System R [Gray et al. 81] and Herlihy's system for replicated data [Herlihy 84], both of which use logs.

Other debugging-related tasks.

We note three aspects of debugging that are not covered in this thesis:

1. Our work has concentrated exclusively on debugging computations. To debug a program, the user must be able to generate computations with desired characteristics. It may be the case that the user should be allowed to specify the ordering of concurrent siblings. Or, it may be helpful to the user to be able to ask for a computation that is just like another computation but differs in some ordering of descendant subactions. What set of tools would be most useful to the user for controlling a computation, and how these tools might be implemented, needs to be studied. Similarly, the tools for controlling interactions of activities within a user-defined atomic type also need further study.

2. In this thesis, we did not present a user interface to the debugging system. The question of how objects should be displayed and named, and how the user interacts with the debugging system needs to be studied.

3. Our debugging method is not applicable to performance-related bugs. This is because when the user debugs a computation, he or she is really debugging a functionally equivalent serial execution. Such an execution will not reveal that an action had to wait to get a lock from another action, for example, nor that an action was aborted to break a deadlock. Understanding and debugging the performance of concurrent and distributed computations is an important and difficult task that still awaits systematic study.

References

[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*, August 1983, pp. 31-44.

[Balzer 69]
Balzer, R.M. "EXDAMS - EXtendable Debugging and Monitoring System," *Proceedings of the AFIPS Spring Joint Computer Conference*, Vol. 34, 1969, pp. 567-580.

[Bayer et al. 80]
Bayer, R., Heller, H., and Reiser A. "Parallelism and Recovery in Database Systems," *ACM Transactions on Database Systems*, Vol. 5, No. 2, June 1980, pp. 139-156.

[Bernstein & Goodman 81]
Bernstein, P.A. and Goodman, N. "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-221.

[Bernstein & Goodman 83]
Bernstein, P.A. and Goodman, N. "Multiversion Concurrency Control - Theory and Algorithms," *ACM Transactions on Database Systems*, Vol. 8, No. 4, December 1983, pp. 465-483.

[Chan et al. 82]
Chan, A., Fox, S., Lin, W.T.K., Nori A., and Ries, D.R. "The Implementation of an Integrated Concurrency Control and Recovery Scheme," in *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, June 1982, pp. 184-191.

[Davies 73]
Davies, C.T. Jr. "Recovery Semantics for a DB/DC System," in *Proceedings of the 1973 ACM National Conference*, 1973, pp. 136-141.

[DuBourdieu 82]
DuBourdieu, D.J. "Implementation of Distributed Transactions," in *Proceedings of the Sixth Berkeley Workshop on Distributed Data Mananagement and Computer Networks*, February 1982, pp. 81-94.

[Eswaran et al. 76]
Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, November 1976, pp. 624-633.

[Gertner 80]
Gertner, I. *Performance Evaluation of Communicating Processes*, Ph.D. Thesis, Technical Report TR-76, Department of Computer Science, Univ. of Rochester, May 1980.

[Gray 78]
Gray, J.N. "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, Vol. 60, *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1978, pp. 393-481.

[Gray 81]
Gray, J. "The Transaction Concept: Virtues and Limitations," in *Proceedings of the Seventh International Conference on Very Large Data Bases*, September 1981, pp. 144-154.

[Gray et al. 81]
Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I. "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 223-242.

[Guttag et al. 78]
Guttag, J., Horowitz, E., and Musser, D. "Abstract Data Types and Software Validation," *Communications of the ACM*, Vol. 21, No. 12, December 1978, pp. 1048-1064.

[Herlihy 84]
Herlihy, M.P. *Replication Methods for Abstract Data Types*, Ph.D. Thesis, Technical Report MIT/LCS/TR-319, Laboratory for Computer Science, MIT, May 1984.

[Herlihy & Liskov 82]
Herlihy, M. and Liskov, B. "A Value Transmission Method for Abstract Data Types," *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 4, October 1982, pp. 527-551.

[Kung & Robinson 81]
Kung, H.T. and Robinson J.T. "On Optimistic Methods for Concurrency Control" *ACM Transactions on Database Systems*, Vol. 6, No. 2, June 1981, pp. 213-226.

[Lamport 78]
Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, July 1978, pp. 558-565.

[Lampson 81]
Lampson, B. "Atomic Transactions," in *Distributed Systems: Architecture and Implementation*, Vol. 105, *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1981, pp. 246-265.

[Liskov 82]
Liskov, B. "On Linguistic Support for Distributed Programs," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 3, May 1982, pp. 203-210.

[Liskov 84]
Liskov, B. "Overview of the Argus Language and System," Programming Methodology Group Memo 40, Laboratory for Computer Science, MIT, February 1984.

[Liskov et al. 81]
Liskov, B., Atkinson R., Bloom T., Moss, E., Schaffert J.C., Scheifler, R., Snyder, A. *CLU Reference Manual*, Vol. 114, *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1981.

[Liskov & Scheifler 83]
Liskov, B., and Scheifler, R. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July 1983, pp. 381-404.

[Liskov & Zilles 74]
Liskov, B., and Zilles, S.N. "Programming with Abstract Data Types," in *Proceedings of the ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN Notices*, Vol. 9 No. 4, April 1974, pp. 50-59.

[Lomet 77]
Lomet, D.B. "Process Structuring, Synchronization, and Recovery Using Atomic Actions," in *Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices*, Vol. 12, No. 3, March 1977, pp. 128-137.

[Mcdaniel 77]
Mcdaniel, G. "METRIC: a Kernel Instrumentation System for Distributed Environments," in *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, November 1977, pp. 93-99.

[Model 79]
Model, M.L. *Monitoring System Behavior in a Complex Computational Environment*, Stanford U. Ph.D. Thesis, Technical Report CSL-79-1, Xerox Palo Alto Research Center, January 1979. (Also available as Stanford Univ. Computer Science Dept. Report CS-79-701.)

[Moss 81]
Moss, J.E.B. *Nested Transactions: An Approach to Reliable Distributed Computing*, Ph.D. Thesis, Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, MIT, April 1981.

[Moss 81]
Moss, J.E.B. "Nested Transactions and Reliable Distributed Computing," in *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, July 1982, pp. 33-39.

[Nelson 81]
Nelson, B.J. *Remote Procedure Call*, Ph.D. Thesis, Technical Report CMU-CS-81-119, Department of Computer Science, Carnegie-Mellon Univ., May 1981.

[Randell 75]
Randell, B. "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 220-232.

[Reed 78]
Reed, D.P. *Naming and Synchronization in a Decentralized Computer System*, Ph.D. Thesis, Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, MIT, September 1978.

[Reed 83]
Reed, D.P. *Implementing Atomic Actions on Decentralized Data*, *ACM Transactions on Computer Systems*, Vol. 1, No. 1, February 1983, pp. 3-23.

[Schwarz & Spector 84]
Schwarz, P.M., and Spector, A.Z. "Synchronizing Shared Abstract Types," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 223-250.

[Schiffenbauer 81]
Schiffenbauer, R.D. *Interactive Debugging in A Distributed Computational Environment*, M.S. Thesis, Technical Report MIT/LCS/TR-264, Laboratory for Computer Science, MIT, September 1981.

[Schlichting & Schneider 83]
Schlichting, R.D., and Schneider, F.B. "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems*, Vol. 1, No. 3, August 1983, pp. 222-238.

[Smith 81]
Smith, E.T. *Debugging Techniques for Communicating, Loosely-Coupled Processes*, Ph.D. Thesis, Technical Report TR-100, Department of Computer Science, Univ. of Rochester, December 1981.

[Stearns & Rosenkrantz 81]
Stearns, R.E., and Rosenkrantz, D.J. "Distributed Database Concurrency Controls Using Before-Values" in *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, 1981, pp. 74-83.

[Weihl 84]
Weihl, W.E. *Specification and Implementatin of Atomic Data Types*, Ph.D. Thesis, Technical Report MIT/LCS/TR-314, Laboratory for Computer Science, MIT, March 1984.

[Weihl & Liskov 82]
Weihl, W., and Liskov, B. "Implementation of Resilient, Atomic Data Types," Computation Structures Group Memo 223, Laboratory for Computer Science, MIT, December 1982. To appear in *ACM Transactions on Programming Languages and Systems*.