AXIOMATIC VERIFICATION TO ENHANCE SOFTWARE RELIABILITY

Richard Dale Schlichting
Ph.D. Thesis

TR 82-480
January 1982

Department of Computer Science
Cornell University
Ithaca, New York 14853

AXIOMATIC VERIFICATION TO ENHANCE SOFTWARE RELIABILITY

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment for the Degree of

Doctor of Philosophy

by

Richard Dale Schlichting

January 1982

AXIOMATIC VERIFICATION TO ENHANCE SOFTWARE RELIABILITY

Richard Dale Schlichting, Ph.D.

Cornell University 1982

Techniques that facilitate the design of reliable software are described. Two distinct phenomena that can cause execution of a program to deviate from its specifications are considered. The first is the failure of the computing system on which the program is running. When this occurs, the system might not be capable of following the instructions specified by the program. The second phenomenon is that the program is written so that it will not execute consistently with its specifications, even on a failure-free computing system.

A methodology is presented for designing programs that can cope with failures in the underlying system. It is based on the notion of a fail-stop processor -- a processor with well-defined failure mode operating characteristics. Axiomatic program verification techniques are extended for use in developing provably correct programs for such processors. The problem of meeting response time goals in light of failures is discussed. Lastly, the problem of implementing processors that, with high probability, behave like fail-stop processors is addressed.

Programming logics have already been developed to reason about sequential programs, and parallel programs that use shared memory or synchronous message-passing. That work is extended to facilitate reasoning about programs that use asynchronous message-passing. Two benefits accrue from this. The obvious one is that partial correctness proofs can be written

for concurrent programs that use such primitives. This allows such programs to be understood as predicate transformers, instead of by contemplating all possible execution interleavings -- often an intractable task. The other benefit is that these proof rules and their derivation shed light onto how interference arises when message-passing operations are used, and how this interference can be controlled. In particular, disciplines that make asynchronous message-passing primitives simple and safe to use are explored.

A partial correctness proof of the two-phase commit protocol illustrates the application of the new techniques. This protocol, widely used in database applications, ensures that a specified action is performed either at all sites in a distributed system, or at no site, despite failures.

Biographical Sketch

Richard Dale Schlichting was born on April 17, 1955 in Chicago, Illinois. He graduated from Rutherford B. Hayes High School in Delaware, Ohio in 1973. That fall he enrolled at the College of William and Mary in Williamsburg, Virginia, from which he received in 1977 a BA with Honors in Mathematics (Computer Science option), and History. He married Cynthia S. Meyer of Delaware, Ohio on May 28, 1978. In 1979, he received an MS in Computer Science from Cornell University. He is a member of Phi Alpha Theta, Phi Beta Kappa, IEEE, and the ACM.

To Cindy

Acknowledgments

I would like to thank my advisor, Fred B. Schneider, for his immeasurable contributions to this dissertation and to my development both as a scientist and as an individual. Under his tutelage, I have learned valuable techniques for conducting research in this field. Moreover, I have learned that the best ideas are worthless if they cannot be explained in a clear and concise manner. The discussions we have had on this and other subjects have always been enlightening, and invariably left me with a clearer understanding of the issues involved. I am indebted to him for the enormous amount of time and effort he has spent attempting to mold me into a competent researcher and writer; there is no doubt I am better person for it.

I would also like to thank the other member of my special committee, R. Conway and L. Schruben, for their careful reading of this dissertation.

G. Andrews, H. Boehm, E. Dijkstra, L. Lamport, R. Reitman, D. Wall, and S. Worona all made positive contributions to the methodology presented in chapter 2. D. Gries graciously read an earlier version of that chapter, and made many valuable comments. The process-control application was first suggested by J. Kemp, W. Comfort and M. Kushner of IBM (FSD/Owego). The presentation in chapter 4 was substantially improved by the careful reading given to an earlier draft by D. Wright.

Special thanks go to G. Levin and D. Zlatin, for providing technical advice, moral support, and friendship during my years at Cornell.

I would also like to thank the University of Arizona for graciously allowing me to remain at Cornell an extra semester to finish this dissertation.

This section would not be complete without acknowledging the role of my mother. From her I learned the necessity of hard work, and the desire to always perform to the utmost of my ability. Without her support, I would never have progressed this far.

Finally, I would like to thank my wife, Cindy. Her infinite patience and constant encouragement have sustained me throughout the past four and a half years. To her goes my eternal gratitude, as well as my eternal love.

Table of Contents

Chapter 1

Introduction

1.1. <u>Overview</u>

There is a great need for computing systems that provide reliable service -- systems that can correctly execute despite failures. This is because increasing reliance is being placed on computers to perform crucial tasks. For example, should a failure occur in a computer that controls the critical functions of an aircraft or a nuclear reactor, lives might be lost. In other situations, computer failures might lead to economic loss or to inconvenience.

The construction of a system that minimizes the effects of failures requires highly reliable hardware, and software that is both correct and robust to failures of the hardware and supporting software. A *failure* occurs in a component when its behavior deviates from its specifications. Program failures are due to two distinct phenomena. The first is *design errors* in the program. A program with a *design* (sic) does not satisfy its specifications, hence it will not behave correctly even if run on a failure-free computing system. The other phenomenon is the failure of the computing system on which the program is running. A system that has failed might not be capable of following the instructions specified by the program.

Both of these phenomena are equally important for the construction of reliable computing systems. Hence, it is appropriate to consider various techniques to ensure that both types of failures can be tolerated.

1.2. Correctness of Design

Since writing correct programs is a long-standing concern of computer scientists, it is not surprising that numerous techniques have been proposed for dealing with design errors. They can be divided into two basic approaches: avoiding the errors *a priori*, and writing the program in such a way that design errors do not adversely affect execution.

1.2.1. Avoiding Design Errors

The foremost method for avoiding design errors is the use of axiomatic program verification. By using a programming logic that relates programs to the state transformations they implement, a program can be developed whose execution on a failure-free system will be consistent with its formal specifications. Of course, to do this requires formulating within a logic the effects of executing each statement type within the language, as well as defining rules to allow the synthesis of such statements into programs.

Since first proposed by Hoare [Hoare 1969], programming logics have been developed to allow formal reasoning about a large number of programming languages, including PASCAL [Hoare & Wirth 1973], and EUCLID [Lampson *et al.* 1977]. Programs written using (only) sequential programming constructs -- assignment, **if**, **do**, and certain types of procedures -- can now be formally verified [Gries 1981]. Recent research has addressed inference rules to facilitate proofs of systems of concurrently executing processes. One approach, due to Owicki and Gries, allows verification of parallel programs that use shared memory for synchronization and communication [Owicki & Gries 1976]. This work has been extended in [Levin & Gries 1981], [Apt *et al.* 1980], and [Misra & Chandy 1981] to deal with programs that synchronize and communicate using a restricted form of message-passing first proposed by Hoare in his Communicating Sequential Processes notation [Hoare 1978].

Another method that potentially facilitates the construction of correct programs is program testing. The goal of testing is to identify design errors so they can be corrected. This is done by observing the behavior of a program when it is executed on each element in some selected subset of its input domain. Such a collection of data is called a _test set_; each element in this set is called a _test case_. A test is _successful_ if the program executes correctly when each test case in the given test set is used as input; otherwise, it is _unsuccessful_.

If a test is unsuccessful, then the program must contain at least one design error. On the other hand, if it is successful, the immediate conclusion can only be that the test cases in the test set do not reveal design errors in the program -- not that the program is devoid of such errors. The most important and difficult part in the testing process, then, is performing the required induction step -- concluding that the program will execute correctly for any data in its input domain, given that a test set has been successfully executed. Clearly, the validity of such a conclusion depends on the composition of the test set.

In [Goodenough & Gerhart 1975], the question of selecting test cases to form a test set is addressed. There, three properties that a successful test must have in order to conclude that a program is correct are defined. The test set must be _complete_ with respect to the _selection criterion_. That is, the test set must satisfy all of the properties specified by the criterion. For example, suppose the selection criterion is that each statement in the program be executed at least once [Huang 1975]. Then, a complete test set would be any set of input data TS satisfying the following:

$$(\forall S: S \text{ a statement in the program}: (\exists tc: tc \in TS \wedge tc \text{ in input domain}: tc \text{ causes } S \text{ to be executed})).$$

In addition, the selection criterion itself must be both _reliable_ and _valid_. A criterion is _reliable_ if and only if the program will either execute each complete test set successfully or execute each complete test set unsuccessfully. A criterion is _valid_ if and only if every error in the program can be revealed by executing some complete test set.

In light of these requirements, Goodenough and Gerhart examine one approach to developing a valid and reliable selection criterion. It involves examining the program and its specifications to determine a set of _test predicates_. Given these predicates, a selection criterion can be determined and test sets computed. While showing the validity of this or any criterion is usually easy, the same cannot be said for the reliability of a criterion. It should not be surprising that proving reliability is difficult, since this proof corresponds to making the induction step mentioned above. In fact, showing the reliability of a selection criterion based on test predicates turns out to be comparable to proving the correctness of the program using axiomatic verification techniques.

Because of the difficulty of constructing a reliable and valid selection criterion, approximation techniques have been devised for program testing. One notable effort is _mutation analysis_ developed by Budd _et al_. [Budd _et al_. 1980]. A _mutant_ of a program P is a program obtained by changing P in some small, specified way. A test case _differentiates_ P from a mutant if P and the mutant behave differently when executed on that test case. Clearly, a good test set for detecting design errors in P contains enough test cases to differentiate P from each of its mutants. The premise of this argument -- supported by experimental evidence -- is that either P or one of its mutants is correct. Thus, there is a high probability that P is correct if it successfully executes such a test set.

1.2.2. Tolerating Design Errors

Another approach to handling design errors is to structure a program so that some design errors can be tolerated. This is done primarily by redundant execution of independently developed code.

A general scheme for doing this is outlined in [Fischler and Firschein 1973]. It consists of executing several different implementations of an algorithm to produce a set of resulting states. Then, a majority consensus scheme is used to determine which of the states is to be considered the outcome of that execution. This scheme is based on the assumption that design faults are so rare that programs containing mistakes will consistently be outvoted by correct colleagues. Moreover, it is also assumed that there are no design errors present in the voting mechanism.

A variation of this scheme has been developed at the University of Newcastle-upon-Tyne [Randell 1978]. A program is constructed using _recovery blocks_. A _recovery block_ consists of a _primary block_, an _acceptance test_, and one or more _alternate blocks_. Upon entry to a recovery block, the primary block is executed. After its completion, the _acceptance test_ is executed to determine if the primary block has performed acceptably. If the test is passed, the recovery block is exited. Otherwise, an alternate block -- generally a different implementation of the same algorithm -- is attempted and the acceptance test repeated. Execution of each alternate block is attempted in sequence until one produces a state in which the acceptance test succeeds. Execution of an alternate block is always begun in the recovery block's initial state.

An implicit assumption in this technique is that any block whose resulting state passes the acceptance test is devoid of design errors. For this to be true, the acceptance test must, in some sense, be an encoding of the output specification of the program. Unfortunately, devising a test that meets this criterion may be non-trivial. Also, the necessity of restoring the system state to its initial value before attempting execution of an alternate complicates the implementation [Anderson & Kerr 1976].

1.3. Correctness of Operation

When a system fails, any program executing on it may also fail unless it can make allowances for the effects of the underlying failure. Unfortunately, programming a computer system that is subject to failures is an extremely difficult task. A malfunctioning processor might perform arbitrary and spontaneous state transformations instead of the transformations specified by the programs it executes. Thus, even a correct program cannot be counted on to implement a desired input-output relation when it is run on a malfunctioning system. Furthermore, by using only a finite amount of hardware, it is impossible to build a computer system that always operates correctly in spite of failures in its components[1]. Thus, the goal of implementing a failure-free computing systems is unattainable.

---

[1] _Sed quis custodiet ipsos Custodes_? (But who is to guard the guards themselves?) [Juvenal 130]

Fortunately, such perfection is unnecessary. Rather, it is sufficient that a system work correctly provided no more than k failures occur within some time interval, or provided that certain types of catastrophic failures do not occur. This more modest goal _is_ attainable. By using hardware and software redundancy, it is possible to approximate a system that exhibits predictable failure-mode operating characteristics. It then becomes feasible to write programs that will execute correctly, provided that too many failures do not occur.

In [Lampson 1981], this approach is used to design a crash resistant data storage system. First, the physical components of the system -- disks, processors, and the communications system -- are modeled. All events in these models are characterized as either _desired_ or _undesired_. Only desired events would occur if all components were failure-free. Undesired events are further divided into _errors_, which are expected and can be tolerated, and _disasters_, which cannot be tolerated. Then, approximations of virtual devices that possess more predictable operating characteristics are implemented based on these models. Such devices operate correctly to the extent that the model approximates reality. Thus, if only desired events and errors occur, correct operation will be observed. However, if a disaster occurs, arbitrary behavior might be observed. It is hoped that most common failures will be viewed as errors.

Disk storage is modeled as a set of pages, each of which contains a block of data and has associated with it a _status_. The status of a page is either _good_ or _bad_. To access a page, read and write procedures are defined. The desired result of reading a page is for its status and correct data to be returned. If the page is good but execution of the operation returns bad, then either an error or a disaster has occurred. It is considered a disaster if some specified number of unsuccessful attempts at reading the page have already been made; otherwise, it is an error. The desired result of a write operation is to update a page and set its status to good. If no action occurs on a write, or if the update incorrectly sets the status of the page to bad, then it is considered an error. The remaining disk related events are _decays_ -- undesired spontaneous transformations of the page contents. If decay is infrequent (a concept made more precise in [Lampson 1981]), or if the transformation changes the status of a page with correct data from bad to good, it is considered an error. A disaster occurs if decay is frequent or if the page suffers an undetectable change.

A processor is modeled as a collection of processes, each with some _local state_ and a _shared state_. The only error defined is a _crash_. This occurs when all local states and the shared state spontaneously become reset to some predefined value. All other undesired events are disasters. Note that since disasters are supposed to occur with low probability, the implicit assumption is made that most processor malfunctions can be transformed into crashes by internal consistency checks.

The final physical component modeled is the communications system. This is represented by a set of messages, each containing a _status_, _destination_, and _block of data_. As with the disk model, the status of a message is either _good_ or _bad_. To modify the set of messages, **send** and **receive** operations are defined. The effect of executing the former is to create a message, while a process executing the latter blocks for an arbitrary amount of time, and then receives a message. After execution of a receive terminates, the message that was received is removed from the set.

Undesired events in the communications system are all spontaneous. Errors include loss or duplication of a message, and the change of a message status from good to bad. A disaster happens when an undetected transformation occurs in the contents of a message.

Virtual devices that mask the effects of most errors (but not disasters) from higher levels are then constructed. Disk storage is replaced by <u>stable storage</u> -- an abstraction where read and write operations result in the occurrence of only desired events. Stable storage is used in implementing <u>stable processors</u>. While a stable processor can crash, it provides a more convenient interface for programming robust programs. In particular, it allows processes to save information in stable storage, and implements <u>stable monitors</u> -- monitors [Hoare 1974] in which all shared variables are in stable storage and whose procedures are executed atomically with respect to failures. The communications system is replaced by a more structured interface in which <u>remote procedure calls</u> are used instead of explicit message-passing operations. Such calls are identical to local procedure calls, except that the action taken by the procedure may be performed more than once for a given invocation. Such multiple execution could occur if processors crash or messages are lost or duplicated.

Finally, a protocol to preserve the consistency of the data in the storage system is devised. Because these devices are almost error-free, the protocol itself need only be robust to processor crashes. As a result, both the development of the protocol and its correctness arguments are relatively straightforward.

The key to this design methodology is the use of devices that have predictable failure-mode operating characteristics. With such devices, the problem of writing programs that operate correctly despite failures is simplified considerably. In fact, when the action a processor takes upon failure can be predicted, it becomes possible to apply axiomatic verification techniques to the development of correct programs that run on that processor. As we show in this dissertation, this can be used to formulate a methodology for the design of reliable systems.

1.4. Dissertation Outline

In chapter 2, a methodology for designing fault-tolerant computing systems is presented. It is based on the notion of a *fail-stop processor* -- a processor whose only failure is a crash. First, we describe axiomatic program verification techniques for use in developing provably correct programs for fail-stop processors. Then, the problem of implementing processors that, with high probability, behave like fail-stop processors is addressed. Finally, the use of our methodology is illustrated in the design of a prototype fault-tolerant process-control system.

Chapter 3 describes extensions to axiomatic verification to faciliate reasoning about programs that use asynchronous message-passing for communication. In addition to providing the means to verify the partial correctness of such programs, the development of these rules provides insight into why the use of asynchronous message-passing primitives makes programs difficult to understand. We also explore disciplines that make asynchronous message-passing primitives simple and safe to use.

The use of the verification techniques derived in chapters 2 and 3 is then illustrated by presenting a partial correctness proof of the *two-phase commit protocol* [Gray 1978]. This protocol, widely used in database applications, ensures that a specified action is performed either at all sites in a distributed system, or at no site, despite failures.

Lastly, some conclusions are presented and topics for further research are described.

Chapter 2

Designing Fault-Tolerant Computing Systems

2.1. Introduction

The use of computing systems to control complex devices or physical processes is becoming increasingly important. Such systems must satisfy real-time response constraints and be fault-tolerant in addition to implementing a specified relation between input and output.

In this chapter, we present an approach to designing fault-tolerant computing systems based on the notion of a *fail-stop processor* -- a processor with well-defined failure-mode operating characteristics. We extend axiomatic verification techniques to facilitate development of provably correct programs for such processors, and describe how response time constraints can be met in light of failures. Then, the problem of implementing -- actually, approximating -- fail-stop processors is addressed. Next, we apply this approach to a non-trivial problem: the design of a fault-tolerant process-control system. Lastly, our approach is contrasted with other related work.

2.2. Fail-Stop Processors

A processor is characterized by the instruction set it supports. Each instruction causes a well-defined transformation on the internal state of the processor and/or the connected storage and peripheral devices. Thus, the effects of executing each instruction can be described by a precise semantic definition -- be it a temporal axiomatization of the instruction [Pnueli 1979] or a "Principles of Operation" manual [IBM]. A *failure* occurs when the behavior of the processor is not consistent with this semantic definition.

The internal state of a fail-stop processor and some predefined portion of the connected storage is assumed to be *volatile*. The contents of volatile storage are irretrievably lost whenever a failure occurs. The remaining storage is defined to be *stable*; it is unaffected by any kind of failure. Stable storage is likely to be relatively slow.

A fail-stop processor is also distinguished by its failure-mode operating characteristics, which are extremely simple. In contrast to a real processor, a fail-stop processor never performs an erroneous state transformation due to a failure. Instead, the processor simply halts. Thus, the only visible effects of a failure in a fail-stop processor are:

FS1: It stops executing.

FS2: The internal state and contents of the volatile storage connected to it are lost.

2.3. Programming a Fail-Stop Processor

2.3.1. Recovery Protocols

A program executing on a fail-stop processor is halted when a failure occurs. Execution may then be *restarted* on a correctly functioning fail-stop processor. (This may be the original processor if the cause of the failure has been repaired, or it may be another fail-stop processor.) When a program is restarted, the internal processor state and the contents of volatile storage are unavailable (due to FS2). Thus, some routine is needed that can complete the state transformation that was in progress at the time of the failure and restore storage to a well-defined state. Such a routine is called a _recovery protocol_.

Clearly, a recovery protocol (i) must execute correctly when started in any intermediate state that could be visible after a failure and (ii) can use only information that is in stable storage. In addition, because the code for a recovery protocol must be available after a failure, it must be kept in stable storage. We might postulate a convention that the recovery protocol in effect is stored at a fixed position in stable storage. Then, the relevant recovery protocol can be found when it is needed.

We associate a recovery protocol with a sequence of statements implementing a state transformation to form a _fault-tolerant action_ as follows:

```
<action-name>: action
               <action-statement>
               recovery
               <recovery protocol>
               end
```

Execution of <action-name> consists of establishing <recovery protocol> as the recovery protocol to be in effect when <action-statement> is executed and then executing <action-statement>. When <action-name> terminates, the recovery protocol in effect when it was started is reestablished. If execution of <action-name> is interrupted by a failure, upon restart execution continues with the recovery protocol. Subsequent failures cause execution of <action-name> to be halted and execution of the recovery protocol to begin anew when the program is restarted. Execution of <action-name> terminates when execution of either <action-statement> or <recovery protocol> is performed in its entirety without interruption.

The following syntactic abbreviation will be used to denote that an action-statement serves as its own recovery protocol:

```
<action-name>: action, recovery
               <action-statement>
               end
```

Such a fault-tolerant action is called a *restartable action*[1].

A program running on a fail-stop processor must at all times have a recovery protocol in effect. This will be the case if the program itself is a single fault-tolerant action. Alternatively, assuming that establishment of a recovery protocol can be done atomically, a program can be structured as a sequence of fault-tolerant actions. Then, between execution of one fault-tolerant action and the next, either the old recovery protocol or the new one will be in effect.

2.3.2. Axioms for Fault-Tolerant Actions

Following the Floyd-Hoare axiomatic approach [Hoare 1969], the state of a program is characterized by a function from program variables to values. An assertion P is a Boolean-valued expression involving program and logical variables. The syntactic object:

$$\{P\}\ S\ \{Q\}$$

where P and Q are assertions and S is a statement in a programming language, is called a *triple*. The triple {P} S {Q} is a *theorem* if there exists a proof of it in a specified formal deductive system, usually called a *programming logic*. A programming logic consists of a set of axioms and rules of inference that relate assertions, programming language statements,

[1] Others have used the term *idempotent* to describe this notion.

and triples. Of particular interest are those logics that are sound with respect to execution of programming language statements on the program state -- i.e., deductive systems that are consistent with the operation of a "real" machine. Then, the notation {P} S {Q} is usually taken to mean:

> If execution of S begins in a state in which P is true, and terminates, then Q will be true in the resulting state.

The language and programming logic described in [Owicki & Gries 1976], extended to include guarded commands, is used in this paper.

It is often more convenient to write a <u>proof outline</u> than a formal proof. A <u>proof outline</u> is a sequence of programming language statements interleaved with assertions. Each statement S in a proof outline is preceded directly by one assertion, called its <u>precondition</u> and denoted <u>pre</u>(<u>S</u>), and is directly followed by an assertion, called its <u>postcondition</u> and denoted <u>post</u>(<u>S</u>). A proof outline is an abbreviation for a proof if:

PO1: for every statement S, the triple {pre(S)} S {post(S)} is a theorem in the programming logic -and-

PO2: whenever {P} and {Q} are adjacent in the proof outline, then Q is provable from P.

Let FTA be a fault-tolerant action formed from an action-statement A and a recovery protocol R. We wish to develop an inference rule that will allow the derivation of

{P} FTA {Q}

as a theorem, while preserving the soundness of our programming logic with respect to execution on a fail-stop processor. First assume

F1: {P'} A {Q'} and {P"} R {Q"}

have been proved. Then, for execution of A to establish Q, we must have

F2: $P \Rightarrow P'$ and $Q' \Rightarrow Q$.

Similarly, for the recovery protocol R to establish Q, the following (at least) must hold:

F3: $Q'' \Rightarrow Q$.

Recall that R is invoked only following a failure. By definition, the contents of volatile storage are undefined at that time. Therefore, any program variables needed for execution of R -- those variables occurring in P'' -- must be in stable storage. Thus, we require

F4: All program variables named in P'' must be in stable storage.

We must also ensure that whenever the recovery protocol receives control, stable storage is in a state that satisfies P''. This will be facilitated by constructing a <u>replete proof outline</u>. A <u>replete proof outline</u> is a proof outline in which certain intermediate assertions have been deleted so that:

RPO1: No intermediate assertion appears between adjacent fault-tolerant actions.

RPO2: Every triple {P} S {Q} in the proof outline satisfies either

(a) S is a sequence of fault-tolerant actions, or

(b) $\{P \vee Q\}$ is invariant over execution of S.

Thus, if

$$\{P\}\ \ FTA_1\ \ \{P_1\}\ \ FTA_2\ \ \{P_2\}\ \ \ldots\ \ FTA_n\ \ \{P_n\}$$

is a proof outline, then

$$\{P\}\ \ FTA_1;\ \ FTA_2;\ \ \ldots\ \ FTA_n\ \ \{P_n\}$$

is a replete proof outline. For example, if assignment of an integer value to a variable is performed by executing a single, indivisible, (store) instruction -- as it is on most machines -- then

$$\{x = 3\}\ x := 6\ \{x = 6\}$$

is also a replete proof outline. This is because either the precondition or the postcondition of "x:= 6" is true of every state that occurs during execution of the assignment. Even if assignment is not implemented by execution of a single instruction

$$\{val = 3\}\ x := val\ \{x = 3\}$$

is a replete proof outline, because the assertion val = 3 is not destroyed by assignment to x; it is true before, during and after execution of "x:= val".

Correct operation of a recovery protocol therefore requires:

F5: Given a fault-tolerant action with action-statement A and recovery protocol R satisfying F1, let $a_1, a_2, \ldots, a_n$ be the assertions that appear in a replete proof outline of {P'} A {Q'}, and $r_1, r_2, \ldots, r_m$ be the assertions that appear in a replete proof outline of {P"} R {Q"}. Then:

(i) $(\forall\, i: 1 \le i \le n: a_i \Rightarrow P'')$

(ii) $(\forall\, i: 1 \le i \le m: r_i \Rightarrow P'')$

Lastly, it must be guaranteed that failures at processors other than the one executing FTA do not interfere with (i.e., invalidate) assertions in the proof outline of FTA. Suppose an assertion in FTA names variables stored in the volatile storage of another processor P_F.[2] Then, should P_F

---

[2]This is often necessary when the actions of concurrently executing processes are synchronized. For example, if it is necessary to assert that a collection of processes are all executing in the same "phase" at the same time, then each would include assertions about the state of the others.

fail, such an assertion would no longer be true since the contents of volatile storage would have been lost. Hence, we require that:

> F6: Variables stored in volatile storage may not be named in assertions appearing in programs executing on other processors.

Establishing F1 - F6 is required in order to prove that a given fault-tolerant action will perform a desired state transformation. However, from F3 and F5 it follows that given a fault-tolerant action, a restartable action that implements the same state transformation can be constructed from the recovery protocol alone. Thus, in theory, the action-statement is unnecessary. In practice, the additional flexibility that results from having an action-statement different from the recovery protocol is quite helpful. Presumably, failures are infrequent enough so that a recovery protocol can do a considerable amount of work in order to minimize the amount of (expensive) stable storage used. Use of such algorithms for normal processing would be objectionable.

2.3.3. Fault-Tolerant Programs -- A Simple Example

To illustrate the use of rules F1 - F6 as an aid in developing a recovery protocol, consider the following (artificial) problem.

> Periodically, variables x and y are to be updated based on their previous values. Thus, given an update function G, desired is a routine that runs on a fail-stop processor and satisfies the following specification:
>
> $$\{P\colon x = X \wedge y = Y\} \quad \text{update} \quad \{Q\colon x = G(X) \wedge y = G(Y)\}.$$

Logical variables X and Y represent the initial values of x and y, respectively.

If the possibility of failure is ignored, the following program will suffice:

$$
\begin{array}{l}
S1:\ \{P:\ x = X \ \wedge\ y = Y\} \\
\quad S1a:\ x := G(x); \quad \{P1a:\ x = G(X) \ \wedge\ y = Y\} \\
\quad S1b:\ y := G(y); \quad \{P1b:\ x = G(X) \ \wedge\ y = G(Y)\} \\
\quad \{Q:\ x = G(X) \ \wedge\ y = G(Y)\}.
\end{array}
$$

Note that this is a replete proof outline, provided assignment is implemented as an atomic operation. That way $\{P \vee P1a\}$ is invariant over execution of S1a and $\{P1a \vee P1b\}$ is invariant over execution of S1b.

Things become more complicated when the possibility of failure is considered. In particular, S1 would not constitute the action-statement of a restartable action because F5 is violated (assuming G is not the identity function). The difficulty is that execution of S1a destroys the conjunct $x = X$ in P, and similarly execution of S1b destroys the conjunct $y = Y$. In order to construct a restartable action, we must find a way to make progress -- compute G(X) and G(Y) -- but without destroying the initial values of x and y unless both values are updated. One way to do this is to compute the new values and store them in some temporary variables, giving the following restartable action:

$$
\begin{array}{l}
U1:\ \textbf{action,recovery} \\
\quad \{P:\ x = X \ \wedge\ y = Y\} \\
\quad U1a:\ xnew := G(x); \quad \{x = X \ \wedge\ xnew = G(X) \ \wedge\ y = Y\} \\
\quad U1b:\ ynew := G(y); \quad \{x = X \ \wedge\ xnew = G(X) \ \wedge\ y = Y \ \wedge\ ynew = G(Y)\} \\
\quad \textbf{end} \\
\quad \{Q':\ x = X \ \wedge\ xnew = G(X) \ \wedge\ y = Y \ \wedge\ ynew = G(Y)\}.
\end{array}
$$

Note that x and y must be stored in stable storage, in order to satisfy F4. Having established Q', it is a simple matter to establish Q:

S2: {Q': $xnew = G(X) \wedge ynew = G(Y)$}
    S2a: x:= xnew; {$x = xnew = G(X) \wedge ynew = G(Y)$}
    S2b: y:= ynew; {$x = xnew = G(X) \wedge y = ynew = G(Y)$}
    {Q: $x = xnew = G(X) \wedge y = ynew = G(Y)$}

This is a replete proof outline, and so as long as xnew and ynew are stored in stable storage, F1 - F6 are satisfied. So

U2: **action,recovery**
    {Q': $xnew = G(X) \wedge ynew = G(Y)$}
    U2a: x:= xnew; {$x = xnew = G(X) \wedge ynew = G(Y)$}
    U2b: y:= ynew; {$x = xnew = G(X) \wedge y = ynew = G(Y)$}
    end
    {Q: $x = xnew = G(X) \wedge y = ynew = G(Y)$}

is a restartable action. Thus, the desired program is:

{ P } U1; U2 { Q }

which is also a replete proof outline.

2.4. Termination and Response Time

2.4.1. Total Correctness of Fault-Tolerant Actions

Recall that a proof of the theorem

{P} S {Q}

does not ensure that mechanism S will terminate; only that if it does terminate Q will be true of the resultant state. This is called *partial correctness*. Clearly, *total correctness* -- that S will indeed terminate and that Q will be true of the resultant state -- is both a stronger and more desirable property.

Most statements in our programming notation are guaranteed to terminate, once started. However, loops and fault-tolerant actions are not. Techniques based on the use of variant functions or well-founded sets can be used for proving that a loop will terminate [Dijkstra 1976]. Unfortunately, without knowledge about the frequency of failures, termination of a program written in terms of fault-tolerant actions cannot be proved; if failures occur with sufficiently high frequency, there is no guarantee that the component fault-tolerant actions will terminate. Neither the action-statement nor the recovery protocol of a fault-tolerant action will run uninterrupted, and so the recovery protocol will continually restart.

For a given execution of a program S, define $t(s_i)$ to be the maximum length of time that elapses until execution of the next fault-tolerant action in S is started, once execution of statement s_i is begun[3]. Define

$$T_{max} = \max_{s \in S} t(s).$$

For an execution of S to terminate, it is sufficient that there be enough intervals of length T_{max} during which there are no failures. Then, no fault-tolerant action will be forever restarted due to the (high) frequency of failures.

Of course, this gives no bound on how much time will elapse before S completes. Rather, we have argued that S is guaranteed to terminate if the elapsed time between successive failures is long enough, often enough. This should not be surprising. However, it does provide some insight into how to structure a program in terms of fault-tolerant actions if frequent failures are expected: one should endeavor to minimize T_{max}. This can be achieved by making entry into a fault-tolerant action a frequent event -- either by nesting fault-tolerant actions, or composing them in sequence.

---

[3] It is possible to characterize $t(s_i)$ formally. However, for our present purposes, the informal definition will suffice.

2.4.2. Failures and Real-Time Response Constraints

In many applications a computing system must respond to events in a timely manner. In [Wirth 1977] a discipline for writing programs that must meet real-time constraints was proposed. More recently, [Harter & Bernstein 1981] describe extensions to temporal logic [Lamport & Owicki 1980] that allow construction of a proof that a program will meet some specific response-time goals.

Given a collection of fail-stop processors, it is possible to configure a system that not only implements a given relation between input and output, but performs this state transformation in a timely manner despite the occurrence of failures. After the failure of a processor fsp, a *reconfiguration rule* is used to assign programs that were running on fsp to working fail-stop processors. The *recovery protocol* in effect at the time of the failure facilitates restart of the program. Thus, processor failures are transparent, but for possibly increased execution times. Execution delays from the following sources are incurred as a result of a failure:

(1) Some time t_{detect} will elapse after the fail-stop processor halts until that fact is detected and reconfiguration is begun.

(2) Reconfiguration causes execution delays, as well. First, t_{recon} is required to determine an appropriate assignment of programs to the remaining fail-stop processors. Then, t_{move} might be required to move the program code and contents of its stable storage, if that data is not directly accessible by the processor to which the program is being moved.

(3) In the worst case, the effects of the last T_{max} seconds worth of execution before the failure will be lost.

(4) An additional execution delay $T_{degrade}$ might be incurred because a recovery protocol is likely to take longer to execute than an action-statement. Let $t_{degrade}(fta_i)$ be the difference in elapsed time required for the action-statement and the recovery protocol in fault-tolerant action fta_i to execute to completion, assuming no failures occur. Define:

$$T_{degrade} = \max_{fta \in S} t_{degrade}(fta).$$

Thus, $T_{degrade}$ represents the worst possible delay incurred due to execution of a recovery protocol instead of an action-statement.

This suggests the following strategy for constructing fault-tolerant systems that will continue to behave correctly in spite of up to k failures, for $k > 0$. First, a program is developed (i) that implements the desired state transformations when run on fail-stop processors, (ii) that satisfies its real-time response constraints provided no failures occur, and (iii) in which no process must respond to two events that are separated in time by less than T_F seconds, where:

$$T_F = k(t_{detect} + t_{recon} + t_{move} + T_{max}) + T_{degrade}$$

Suppose R fail-stop processors are required for this. Then, a computing system with R+k fail-stop processors will be able to tolerate up to k fail-stop processor failures and meet its response-time goals. And, the obvious reconfiguration rule must be used.

Note that if stable storage that can be shared by k fail-stop processors is available, then t_{move} can be made 0. Also, by precomputing various configurations, t_{recon} can be made negligible. This, however, requires a sufficient amount of stable storage to store all possible configurations. Lastly, $T_{degrade}$ can be made 0 by using only restartable actions; however, this uniformly degrades execution speed, even if no failures occur.

2.5. Approximating Fail-Stop Processors

While the notion of a fail-stop processor is a useful abstraction, it is impossible to implement using only a finite amount of hardware; with only a finite amount of hardware, a finite number of failures could disable all the error detection mechanisms and thus allow arbitrary and malicious behavior. However, it is possible to construct computing systems that, with high probability, approximate the behavior of a fail-stop processor and its attendant stable storage. Such an approximation would behave as specified in section 2, unless too many failures occurred within some specified time interval, after which no assumptions about its behavior would be possible.

In this section we consider techniques for approximating fail-stop processors and thereby establish the feasibility of constructing a computing system with the necessary properties. We have no illusions that our architecture is optimal in any regard. For one thing, the ideal fail-stop processor approximation for a given application will depend on various factors: cost, the type of components used in the implementation and their failure modes, the amount of interaction between concurrently executing processes, and how close an approximation is desired.

2.5.1. Establishing Feasibility

A k-fail-stop processor is a computing system that behaves like a fail-stop processor unless k+1 or more failures occur in its components. Thus, if k or fewer failures occur while it is executing, it is halted and the state of stable storage prior to the failure remains available. Such a processor can be implemented by replicating real processors and volatile memory, and interconnecting these components using a communications network. In such a system, peripheral devices are viewed as memory-mapped locations in the stable store.

Stable storage is approximated by storing information in $s \geq 2k+1$ independent (volatile) memory units. If k failures occur -- one failure in each of k different memory units -- then we must still be able to determine the contents of the stable storage. For $s \geq 2k+1$, a majority of the memory units will have the correct value, even if k of them do not. The behavior of a peripheral device is governed by a majority consensus of the values written to the corresponding memory-mapped locations.

Each memory unit is assumed to be capable of responding to requests for the value stored at a particular address as well as to requests to change the value stored at a location. Moreover, we assume the existence of T_{resp}, an upper bound on the length of time it takes a memory unit to respond to such a request.

Requests are made by processors and are conveyed to the addressed memory unit through the communications network. It will be convenient to assume that the network never corrupts the contents of messages and that messages are either delivered or discarded within T_D seconds of being sent. Also we will assume that messages cannot be forged -- one processor cannot generate a message that will appear to have been sent by another. Networks that exhibit these properties with high probability as long as k or fewer failures occur can be implemented quite easily. Transmission errors and forgeries are detected by including redundant information in each message. Messages in which errors are detected are then discarded. Similarly, messages that are not delivered within T_D seconds after being submitted are also discarded.

Processor failures are detected by using multiple processors, each running the same program and periodically comparing its results with those of the other processors. Assume a system of N processors, P_1, P_2, ..., P_N. Each processor P_i is permitted to read from all memory units, but is restricted to write to only one, M_i. In addition, each of the s memory units is assumed to be written into by at least one processor. Thus, $N = s$. Failures will eventually cause the appearance of erroneous values in one or more memory units. Therefore, failures can be detected if processors periodically synchronize and compare the contents of the memory units, an operation we shall call _storage validation_. Synchronization is necessary because processors are assumed to run asynchronously; when the contents of two memory units are compared we must be certain that they reflect the same partial execution by both processors.

Once the value stored at a particular address has been updated in every memory unit, it is impossible to recover its previous state. Hence, storage validation must be performed before updating a stable storage value so that if a failure is detected, the prior contents of stable storage can be restored. Storage validation is accomplished as follows: First, the processors synchronize. Then, the copies of stable storage maintained by each is compared with all the other copies. Finally, the processors synchronize again, and (asynchronously) perform the update to their respective copies of stable storage. To perform the necessary synchronization, each processor keeps a variable synchid, which records the number of synchronization operations attempted. The synchronization operation shown in Figure 2.1, executed by processor P_i, terminates if some processor malfunctions by not sending the required synchronization messages, or if all processors have begun executing synchronization operations with the same value of synchid. Correct operation of the synchronization operation is dependent on the value of the timeout-interval in the **waitfor** statement. In order to

---

```
synch:
   synchid:= synchid+1;
   cobegin
      send <synchid, Pi> to P1;
      waitfor ( "receipt of message with text: <synchid, P1>" v
                   timeout );
      if timeout    → signal failure;
                      halt;
       □ ¬timeout   → skip
      fi
   //
   ...
   //
      send <synchid, Pi> to PN;
      waitfor ( "receipt of message with text: <synchid, PN>" v
                   timeout );
      if timeout    → signal failure;
                      halt
       □ ¬timeout   → skip
      fi
   coend
```

Figure 2.1 -- Synchronization Operation

determine this value, we must know an upper bound on the ratio of the different processor execution speeds and the relative speeds at which the clocks on each processor run. That information, in conjunction with the length of time that has elapsed since the last synchronization operation was performed and T_D (the upper bound for message delivery), allows the timeout interval to be computed [Lamport 1981].

The protocol for storage validation is

```
synch;
for  each address A in stable storage do begin;
   read value of A at each memory unit
   if "all values not identical"  →  signal failure;
                                      halt
    □ "all values are identical"  →  skip
   fi
   end
synch;
```

If a failure is detected, it is necessary to determine the correct values of variables stored in stable storage. Failures are signalled from three places in the storage validation protocol: the two synch operations and the value consistency check.

If a failure is signalled during the comparison of values, then some processor P_D detected an inconsistency among the values it read for a given stable storage location. This inconsistency can be due to (1) failures in memory units, (2) failures in other processors that caused them to write erroneous values to their memory units, (3) a failure in the communications network to convey requests or responses to the memory unit and (4) a failure in P_D resulting in announcement of an error when in fact all the values read are identical. Moreover, by hypothesis we know that at most k errors have occurred. As a result of failures of type (1), (2) and (3) at most k erroneous values will be read. Since there are at least 2k+1 values, then a majority of them must be correct. Thus, a simple majority consensus algorithm can be used to determine the values of variables stored in stable storage. Failures of type (4) do not effect values in memory units, and so a majority consensus can be used in that case, as well.

If the failure is signalled during the synchronization protocol by some processor P_D, then all non-faulty processors must either be executing the synchronization protocol or have completed the synchronization protocol. The latter could occur if a faulty processor had sent some, but not all, synchronization messages. Then, it might have sent the correct message to some non-faulty processors, but not to P_D. Given a system of N processors, by hypothesis at least N-k must be non-faulty. We can partition these N-k processors into two classes: n_a processors that are executing in their synchronization protocols at the time the failure is signalled, and n_b processors that have completed the synchronization protocol. Processors that have completed synchronization might have gone on to update their memory units. Thus, in order to use a majority consensus algorithm to determine the values of variables in stable storage we must have:

$$n_a > k \quad \vee \quad n_b > k$$

Clearly, $N-k = 2k+1$ is true of the smallest value of N that will suffice. Thus, at least 3k+1 processors are required.

It is interesting to note that if all the non-faulty processors are synchronized, then only 2k+1 processors are necessary. This is because the invocations of synch in the storage validation protocol would be unnecessary, and so the second type of failure would never be signalled.

2.5.2. Other Approximation Techniques

While the feasibility of implementing fail-stop processors is established in the previous section, the practicality is not. The amount of hardware required and the frequent pauses for storage validation make our implementation impractical. Moreover, the system described above need not halt after a failure is detected: it can continue operating as a (k-1)-fail-stop processor, then a (k-2)-fail-stop processors, and so on, as if the failures had never occurred. However, eventually a point might be reached where so many failures have occurred that subsequent failures could cause undefined state transformations that will go undetected. Then, reconfiguration and execution of a restart protocol on another fail-stop processor would be appropriate.

The problem with our fail-stop processor implementation stems from the choice of doing replication at the processor and memory level, instead of at some lower level. Using redundancy at lower levels can be done effectively only with detailed knowledge of the operation of the lower level components. While such details vary from one processor to the next, some general observations are possible.

An approximation of a fail-stop processor is nothing more than a computing system that will, with high probability, exhibit certain behavior. Following the pioneering work of Lampson and Sturgis [Lampson 1981] the behavior of a system can be characterized in terms of certain *events*. Events are either *desired*, *errors* or *disasters*. *Desired* events correspond to the normal, failure-free operation of the system; *errors* are undesired events that are expected and can be tolerated; *disasters* are undesired events that are not expected or cannot be tolerated. For example, the halting of a fail-stop processor in response to a failure is an error, while allowing it to continue computing is a disaster. The idea, then, is to minimize the probability of the occurrence of a disaster.

It is our belief that practical approximation of fail-stop processors is well within the state of the art. Consider the implementation of the stable storage abstraction. Certain storage media are inherently volatile -- some types of semiconductor memory, for example -- while others are inherently non-volatile -- magnetic tapes, and to a lesser degree, disks. Thus, even if no replication is employed in implementing stable storage, a judicious choice of storage media can reduce the possibility of a disaster. Implementation of a fail-stop processor requires that all failures be detected. For a reasonable approximation of a fail-stop processor, all that is required is that undetected failures (disasters) be rare. By including error detecting circuitry within a processor, this seems possible.

2.6. Fault-Tolerant Process-Control Software

2.6.1. Developing a Correct Program

Software intended to monitor and control physical processes such as nuclear fission and air traffic is called *process-control software*. *Sensors* determine the state of the *environment* by reporting values of key parameters and/or by detecting *events*. *Actuators* are used to exert control over the environment.

A process-control system can be structured as a collection of cyclic processes executing concurrently. Each process p_i is responsible for controlling some set of actuators act_i and for maintaining $state_i$ -- a vector of _state variables_ that reflect the sensor values p_i has read and the actions it has taken. Interprocess communication is accomplished by the disciplined use of shared variables; a process can read and write its state variables, but can only read state variables maintained by other processes.

Each process consists of a single loop. During execution of its _loop body_, process p_i: (1) reads from some sensors, (2) computes new values for the actuators it controls and state variables it maintains, (3) writes the relevant values to act_i and (4) updates $state_i$.

Since processes execute asynchronously, access to state variables must be carefully controlled. Otherwise, a process might read state variables while they are in the midst of being updated. This could cause the process to perform the wrong actions. To avoid this problem, the state variables maintained by each process p_i are assumed to be characterized by an invariant relation CC_i, called the _consistency constraint_ for $state_i$. CC_i is kept true of $state_i$ except while p_i is updating those variables -- i.e. performing (4) above. This can be accomplished by using _read/write locks_ [Gray 1978] to implement reader-writer exclusion on the state variables maintained by each process. Then, a process trying to read variables in $state_i$ must first acquire a read lock for $state_i$. Such a lock will not be granted if a write lock is already held for those state variables, hence that process will be delayed if $state_i$ is being updated. Similarly, a process about to update $state_i$ will be delayed if other processes are reading those values. Note that as long as each process executes correctly given that the values it reads satisfy the consistency constraints, no assumptions about the relative execution speeds of processes are required for correct operation.

The natural laws that govern our physical world ensure that at any time t the values of the sensors are consistent. Let $CC_{sensors}$ be the consistency constraint associated with the sensors. Clearly, if a process reads all the sensors simultaneously, values that satisfy $CC_{sensors}$ would be obtained. Since such a simultaneous read operation is not implementable, we will assume that sensors change values slowly enough and that processes execute quickly enough so that a consistent set of values is obtained by reading each of the sensors in sequence at normal execution speed.

Correct operation of a process-control system requires that:

PC: The values written to the actuators are related to the values read from the sensors according to a given application-specific function.

However, by using PC as our correctness criterion we are making no stipulations about process speeds. This means that the sequence of values written to the actuators is not uniquely determined by the sequence of states that characterize the environment while the process control system runs. This is because only a subsequence of the actual values assumed by each sensor is read by a process, and the values written to the actuators depend (among other things) on the sequence of values read from each sensor. One hopes that correct behavior of the system is not dependent on exactly which subsequence of values is read -- that being dependent on the process execution speed. Clearly, processes should be executed sufficiently often to detect and react to significant changes in the environment being controlled.

Secondly, PC makes no stipulation about how frequently actuators are updated. Thus, the effect of an actuator on the environment being controlled should be independent of the length of time the actuator remains at a given value. That is,

A1: It is a change in the value stored in the memory-mapped actuator location that initiates its action.

A2: The length of time an actuator remains at a given value has no effect on the physical process it controls.

To simplify the development of the code for each process, assume that each state variable and sensor is read at most once in any execution of the loop body. Code that satisfies this restriction can be written by using local variables to store state variables and sensor values: each state variable and sensor value is stored in a local variable when it is first read; subsequent references are then made to the local variable. Let var[i,t] denote the value of var read by p_i during the t^{th} execution of its loop body, sensor[i,t] denote the values read by p_i from sensors during the t^{th} execution of its loop body, and $act_i[t]$ denote the values written to act_i by p_i during the t^{th} execution of the loop body. Then, acceptable behavior of the system -- operation that satisfies PC -- is characterized by the following. The values written to the actuators are computed according to an application-specific function A from the sensor values read and the past actions of processes. Therefore, after p_i updates act_i for the t^{th} time,

$$\mathrm{Iact}(i,t):\ t=0\ \textbf{cor}\ act_i[t]=A(sensors[i,t],\ state_1[i,t]\ \ldots,\ state_n[i,t]).$$

Furthermore, the values read from state variables will satisfy the consistency constraints if read/write locks are used when accessing shared data; by assumption, sensor values satisfy the consistency constraints. Hence,

$$\text{Icons}(i, t): (\forall t': 0 < t' \le t: \bigwedge_{j=1}^{n} CC_j(\text{state}_j[i,t']) \wedge CC_{\text{sensors}}(\text{sensors}[i,t']))$$

is universally invariant. Lastly, the values in state_i must correctly encode past actions performed by p_i. Therefore, at the beginning of the $t+1^{st}$ execution of the loop body at p_i:

$$\text{Istate}(i,t): t = 0 \textbf{ cor } \text{state}_i = A(\text{sensors}[i,t], \text{state}_1[i,t] \ldots, \text{state}_n[i,t]).$$

Let T_i be an auxiliary variable that records the number of times the loop body of process p_i has been executed -- that is, at any time, T_i-1 executions of the loop body have been completed. Thus, T_i is initialized to 1 and (implicitly and automatically) incremented immediately after the loop body is executed. Then, the correctness criterion PC is satisfied if:

$$I(i): \text{Istate}(i,T_i-1) \wedge \text{Iact}(i,T_i-1) \wedge \text{Icons}(i,T_i-1)$$

is true at the beginning of each execution of the loop body.

newstate contains the values used to update state_i and the actuators, and is specified by:

$$\text{Vnewstate}(i,t): \text{newstate} = A(\text{sensor}[i,t], \text{state}_1[i,t] \ldots \text{state}_n[i,t]).$$

Using these, it is a simple matter to program the loop. This code is shown in Figure 2.2. Note that prior to updating state_i, a write lock must be acquired for it. Hence, the code for up_st is:

```
up_st:  acquire_write_i(i);
        update state_i based on newstate;
        release_write_i(i);
```

The subscript names the process invoking the operation -- in this case p_i; the argument names the lock being acquired. Similarly, in calc all the

```
p_i: process
  do true → {I(i)}
     calc: newstate:= A(sensors,state_i, ...state_n);
     {Vnewstate(i,T_i) ∧ Istate(i,T_i-1) ∧ Iact(i,T_i-1) ∧ Icons(i,T_i)}
     up_act: update act_i based on newstate;
     {Vnewstate(i,T_i) ∧ Istate(i, T_i-1) ∧ Iact(i, T_i) ∧ Icons(i, T_i)}
     up_st: update state_i based on newstate;
     {Vnewstate(i,T_i) ∧ Istate(i, T_i) ∧ Iact(i, T_i) ∧ Icons(i, T_i)}
     {I(i)}
  od
end
```

Figure 2.2 -- Process p_i

state variables maintained by a given process p_j must be read together after a read lock for $state_j$ has been acquired. For example,

```
calc:  read from sensors;
             ...
       acquire_read_i(j);
       stuff := state_j;
       release_read_i(j);
             ...
       acquire_read_i(k);
       morestuff := state_k;
       release_read_i(k);
             .
             .
             .
       newstate := G(stuff, ..., morestuff);
```

2.6.2. Developing The Recovery Protocol

After the failure of a processor fsp, each process that was executing on fsp is halted, and data in its volatile storage is lost. To satisfy PC despite the occurrence of failures, we must endeavor to preserve:

PC': At no time do state variables or actuators have values they could not have had if the failure had not occurred.

Recall that I(i) characterizes values of the state variables and actuators that satisfy PC. Consequently, if it is possible to modify the loop body so that I(i) is true of every intermediate state that can be visible after a failure, then PC' will be satisfied as well. Our task, therefore, is to modify the loop body so that it constitutes a restartable action.

I(i) is true except between the time execution of statement up_act begins, and when statement up_st completes. Thus, we must either mask intermediate states during execution of up_st and up_act, or devise a way to execute up_st and up_act together as an atomic action. This latter option is precluded by most hardware. Thus, to implement the former, we construct a single fault-tolerant action that updates the actuators and state variables based on the value of newstate:

$\{Vnewstate(i,T_i) \wedge Icons(i,T_i)\}$
upall
$\{Vnewstate(i,T_i) \wedge Istate(i,T_i) \wedge Iact(i, T_i) \wedge Icons(i, T_i)\}$.

As long as newstate is saved in stable storage, the following replete proof outline satisfies F1 - F6 and accomplishes the desired transformation.

```
upall: action,recovery
   {Vnewstate(i,T_i) ∧ Icons(i,T_i)}
   up_act: update act_i based on newstate
   {Vnewstate(i,T_i) ∧ Iact(i, T_i) ∧ Icons(i, T_i)}
   up_st: acquire_write_i(i);
          update state_i based on newstate;
          release_write_i(i)
   {Vnewstate(i,T_i) ∧ Istate(i, T_i) ∧ Iact(i, T_i) ∧ Icons(i, T_i)}
   end
```

A replete proof outline for the code executed at p_i is shown in Figure 2.3. Notice that since the loop now forms a restartable action, a process might

---

```
p_i: process
   action,recovery
    {I(i)}
    do true → {Istate(i,T_i-1) ∧ Iact(i,T_i-1) ∧ Icons(i,T_i-1)}
       calc:  newstate:= A(sensors,state_1, ...state_n);
       {Vnewstate(i,T_i) ∧ Istate(i,T_i-1) ∧ Iact(i,T_i-1) ∧ Icons(i,T_i)}
       upall: action,recovery
          {Vnewstate(i,T_i) ∧ Icons(i,T_i)}
          up_act: update act_i based on newstate;
          {Vnewstate(i,T_i) ∧ Iact(i, T_i) ∧ Icons(i, T_i)}
          up_st: acquire_write_i(i);
                 update state_i based on newstate;
                 release_write_i(i)
          {Vnewstate(i,T_i) ∧ Istate(i, T_i) ∧ Iact(i, T_i) ∧ Icons(i, T_i)}
          end
       {I(i)}
    od
   end
```

Figure 2.3 -- Replete Proof Outline of p_i

attempt to acquire a given read/write lock that had already been granted to it. For example, if a failure occurred while up_st were being executed, execution of the recovery protocol would attempt to acquire the write lock on $state_i$, which might already be owned by p_i. Clearly, repeated requests by a given process for the same lock without intervening release operations should not delay the invoker. Implementation of various locks with this property (binary semaphores do not suffice) for two representative stable storage implementations are described below. The first implementation assumes a single, highly reliable, random access memory. In contrast, the second does not require any special type of storage device, but instead employs replication on data independent storage devices.

2.6.2.1. Locks in a Single Shared Memory

Hardware implementations of stable storage approximations exist. Such a storage device is usually constructed by using a non-volatile memory technology and storing enough redundant information with each memory word so that error correcting codes can be used to reconstruct information lost due to hardware failures.

To construct read/write locks, *exclusive locks* are used. An *exclusive lock* x is a data object on which two operations are defined: **lock**(x) and **unlock**(x). A process p_i invoking **lock**(x) is delayed until x has the value "free" or "p_i". The effect of executing this operation is to set x to "p_i". When the value of x is "p_i", we say that p_i has been granted x. Execution of **unlock**(x) sets x to "free". Note that this allows a process that has been granted x to reacquire it at will.

Implementation of exclusive locks is simplified considerably given an instruction that allows interlocked access to memory. On the IBM System 370 architecture [IBM] the Compare-and-Swap (CS) instruction is provided for this purpose; it is used below. In other architectures, similar instructions have been defined. For example, on the DEC VAX11 machines INSQHI, INSQTI, REMQHI can be used [Digital 1979]. Note, however, that not all memory-interlock instructions are powerful enough to implement exclusive locks when restarts are possible. For instance, we have been unable to devise an implementation that uses the Test-and-Set instruction, even though this instruction can be used to construct other synchronization mechanisms.

The effect of executing a Compare-and-Swap instruction is as follows:

```
CS(t,x,n):  atomically
               if  t = x  →  x := n
               □   t ≠ x  →  t := x
               fi
            end
```

Then, for each exclusive lock x, one word of storage is allocated in the shared memory. That word has value 0 if the lock is free; otherwise its value is n_i, a unique integer name associated with the process p_i to which the lock has been granted. Also associated with p_i is a variable, t_i, that is accessed only by that process. Then, for process p_i to perform **lock**(x) and **unlock**(x) the following are used:

```
lock(x): t_i := 0; CS(t_i,x,n_i);
         do t_i ≠ 0 ∧ t_i ≠ n_i →
                     t_i := 0; CS(t_i,x,n_i;)
         od

unlock(x):  x := 0.
```

These exclusive locks are used to construct read/write locks as required above. Let p_1, p_2, ..., p_m be the processes that read $state_i$. Then, the read/write lock for $state_i$ is implemented with m exclusive locks: exi_1, exi_2, ..., exi_m, each initialized to "free". Then, **acquire_read**$_j$ and **release_read**$_j$ for process j are:

acquire_read$_j$(i): **lock**(exi_j)

release_read$_j$(i): **unlock**(exi_j).

Readers do not exclude each other because different exclusive locks are referenced. On the other hand, when p_i is updating $state_i$, no other process should be able to read it. Thus, **acquire_write**$_i$ and **release_write**$_i$ are:

acquire_write$_i$(i):
  for k := 1 **to** m **do; lock**(exi_k) **end;**

release_write$_i$(i):
  for k := 1 **to** m **do; unlock**(exi_k) **end;**

Thus, a writer will exclude all readers. There is no need for a writer to exclude other writers -- by definition there are none, because only p_i can change variables in $state_i$.

2.6.2.2. Locks in a Distributed Storage System

Replication of data in independent volatile memories is another technique for realizing our stable storage approximation. One way of accomplishing this is to save a copy of each item to be stored in stable storage in the local memory at every processor in a distributed system. Then, to obtain the value of a variable, a process reads the copy of that variable that is in its local storage. To write to a variable, a process sends the new value to each processor so that all copies can be updated. However, doing this in the obvious way is not always sufficient -- should a failure occur, the copies might have different values. Various protocols have been developed to ensure that this does not happen; for example, most solutions to the multiple-copy consistency problem for fully and partially replicated distributed database systems suffice [Bernstein & Goodman 1981]. Below, a protocol designed specifically for our application is described.

As before, a read/write lock for $state_i$ is implemented by exclusive locks exi_1, exi_2, ..., exi_m -- one for each process that reads variables in $state_i$. exi_j is implemented in the local memory of the processor executing p_j. **acquire_read**$_j$(i) and **release_read**$_j$(i), the operations for process p_j to obtain and release a read lock, are implemented as follows:

acquire_read$_j$(i): **lock**(exi_j)

release_read$_j$(i): **unlock**(exi_j).

Associated with each process p_i is a stable storage manager process ssm(i,P) at each processor P from which copies of $state_i$ can be accessed. Each stable storage manager implements **acquire_write**$_i$ and **release_write**$_i$ operations as well as updates to its local copy of

$state_i$.

Invocation of **acquire_write**$_i$(i) and **release_write**$_i$(i) by process p_i running on processor P are implemented as message exchanges with ssm(i,P):

acquire_write$_i$(i):
    send "acquire_write" **to** ssm(i,P);
    waitfor (receipt of: "acquired" message from ssm(i,P));

release_write$_i$(i):
    send "release_write" **to** ssm(i,P);
    waitfor (receipt of: "released" message from ssm(i,P)).

Upon receipt of an "acquire_write" message, ssm(i,P) must arrange for exi_1, exi_2, ..., exi_m to be acquired. ssm(i,P) itself can acquire the locks associated with processes executing on P by invoking **lock.** To acquire the remaining locks, M sends "prepare to change $state_i$" messages to all stable storage managers that manage other copies of $state_i$. Each then executes **lock** operations on the appropriate exclusive locks, and replies "ssm prepared". When ssm(i,P) has received either "ssm prepared" or notification of failure from each, it returns a message with text "acquired" to p_i. Note that this scheme requires some facility for detecting processor failures. The use of fail-stop processors makes possible the use of timeouts for this purpose [Lamport 1981].

After completing execution of **acquire_write**$_i$, p_i updates $state_i$. To do so, each update to a variable y in $state_i$, "y := z", is translated into the following:

send "change y to z" **to** ssm(i,P).

Upon receipt of such a message, ssm(i,P) updates the local copy of $state_i$

and saves a copy of the new value in a buffer $sendstate_i$.

When a "release_write" request is received, ssm(i,P) transmits "change $state_i$ to: '$sendstate_i$'" to all other stable storage managers, unlocks any exclusive locks it had acquired, and returns a "released" message to p_i. The message "change $state_i$ to: '$sendstate_i$'" causes a stable storage manager to update its local copy of $state_i$ based on the contents of

---

```
ssm(i,P): process
 do true → receive m;
   if  m = "acquire_write"  →
         forall j ∈ LocExc(i,P) do;  lock(j) end;
         forall P' a processor do;
             send "prepare to change state_i" to ssm(i,P');
             waitfor (receipt of: "ssm prepared" from ssm(i,P')  ∨
                           failed(P');
             end;
         send "acquired" to p_i
    □  m = "update y to z"  →
         save z in sendstate_i;
         y:= z
    □  m = "release_write"  →
         forall P' a processor do;
             send "change state_i to 'sendstate_i'" to ssm(i,P')
             end
         forall j ∈ LocExc(i,P) do;  unlock(k) end;
         send "released" to p_i
    □  m = "prepare to change state_i" →
         forall j ∈ LocExc(i,P) do;  lock(k) end;
         send "ssm prepared"
    □  m = "change state_i to 'sendstate_i'"  →
         state_i:= sendstate_i;
         forall j ∈ LocExc(i,P) do;  unlock(k) end
   fi;
 od
end
```

Figure 2.4 -- Stable Storage Manager for $state_i$ at processor P

sendstate_i and then to unlock any exclusive locks it had acquired.

The code for the stable storage manager for the copy of state_i at processor P is shown in Figure 2.4. There, LocExc(i,P) is a set containing the names of the exclusive locks associated with processes that run on P and read state_i. Note that for correct operation, the order in which messages are sent to ssm's after receipt of a "release_write" message must be the same as the sequence of processors to which p_i would be moved in the event of a failure; otherwise, the restart protocol could allow another process to observe state_i regress in time.

2.7. Discussion

2.7.1. Coping With Design Errors: Related Work

Recall that a *recovery block* consists of a *primary block*, an *acceptance test*, and one or more *alternate blocks* [Randell *et al*. 1978]. The primary block and the alternate blocks are executed in sequence until one produces a state in which the acceptance test succeeds.

Despite the apparent similarity between recovery blocks and fault-tolerant actions, the two constructs are intended for very different purposes. Recovery blocks are used to mask *design errors*, fault-tolerant actions are used in constructing programs that must cope with *operational failures* (in the underlying hardware and software). As such, use of recovery blocks to help cope with operational failures can only lead to difficulties. For example, a recovery block has only a finite number of alternate blocks associated with it, and therefore a large number of failures in the underlying system can cause the available alternatives to be exhausted. Secondly, the initial states of variables modified by a recovery block must be available when execution of an alternate block is begun. The model does not admit the possibility of using volatile storage for program variables, since those values cannot be recovered after a failure.

2.7.2. Coping with Operational Failures: Related Work

Few general techniques have been developed to aid in the design of programs that must cope with failures in hardware or support software. Protocols for specialized problems have been developed: such as recovery in data base systems [Gray 1978], implementation of highly reliable file systems [Lampson & Sturgis 1978] and the use of checkpoint/restart facilities in operating systems [Denning 1976].

Lamport has developed a paradigm for building reliable multi-process systems based on the use of state machines, each receiving the same input and running on different processors [Lamport 1978]. He has analyzed an abstract version of the problem -- which he calls the Byzantine Generals Problem -- to determine the number of processors required so that up to k failures can be tolerated [Lamport *et al*. 1980]. Our implementation of k-fail-stop processors is based on that work.

2.7.3. Whence Fail-Stop Processors

Our desire to use a partial correctness programming logic made fail-stop processors the natural choice for an underlying computational model, In such a processor, failures do not cause incorrect state transformations. Morover, all failures are detected. Thus, if a statement terminates, the transformation performed must be that specified by the statement, and consistent with the programming logic. Consequently, the effects of failures are subsumed by the partial correctness nature of the programming logic.

2.7.4. Application of the Methodology

We have successfully employed the methodology described in this paper both to verify existing fault-tolerant protocols and to devise new ones. In chapter 4, the two-phase commit protocol as described in [Gray 1978] is verified. The process control example described in section 6 of this chapter was developed as part of a project to apply this methodology to design a distributed computing system for navigation in an airplane. The details of that work are discussed in [Schneider & Schlichting 1981].

It is natural to ask whether F1 - F6, the components of our proof rule for fault-tolerant actions, are in fact too restrictive. In that case, there would exist fault-tolerant actions that would behave correctly, but for which no proof would be possible. While we have not proved the relative completeness of our new rule, the success we have had with its application and the way in which it was derived suggest F1 - F6 are not too restrictive to allow proof of any "correct" fault-tolerant action.

Chapter 3

Message-Passing: Proof Rules and Disciplines

3.1. Introduction

In distributed systems -- systems with no shared memory -- the use of message-passing provides a particularly clean way for concurrently executing processes to communicate and synchronize. This is because **send** and **receive** statements closely parallel operations directly supported by the underlying hardware; implementing them is therefore simple and cheap.

In this chapter, we develop proof rules for asynchronous message-passing primitives. Two benefits accrue from this. The obvious one is that these proof rules allow partial correctness proofs to be written for concurrent programs that use such primitives. This allows such programs to be understood as predicate transformers, instead of by comtemplating all possible execution interleavings -- often an intractable task.

The second benefit is that the proof rules and their derivation shed light onto how interference arises when message-passing operations are used, and how this interference can be controlled. In addition, they provide insight into programming techniques to eliminate interference in distributed systems.

3.2. Asynchronous Message-Passing

Execution of a **send** statement

send <expr> **to** <dest>

has the following effect. First, the value of the expression <expr> is computed.[1] Then, a message with that value is sent to the process named <dest>.

It is useful to distinguish between sent, delivered, and received when describing the status of a message. Execution of a **send** statement causes a message to be sent. A message that has been sent might subsequently be delivered to its destination. We do not assume that all messages sent are delivered -- real communications hardware does not guarantee the reliable delivery of messages. Once delivered, a message can be received by executing a **receive** statement.

A **receive** statement has the form:

$$\textbf{receive}\ m\ \textbf{when}\ \beta,$$

where m is program variable and β is a Boolean expression. Execution of this statement delays the invoker until a message with text mtext (say) has been delivered to the invoking process and

$$\beta^{m}_{mtext} = \text{true}.$$

Execution completes by assigning mtext to m. Thus, β is a guard -- of the messages that have been delivered, it controls those that can be received.

3.3. Proof Rules for Asynchronous Message-Passing

3.3.1. Overview

Proofs in our programming logic involve three steps. First, using a sequential programming logic [Hoare 1969] and the communications axioms described below, each process is annotated with assertions, giving a

[1] Both simple and structured values (as in CSP [Hoare 1978]) can be communicated in messages.

sequential annotation. Secondly, assumptions made in the sequential annotation about the effects of receiving messages are validated by showing _satisfaction_[2]. This involves generating a collection of _satisfaction invariants_ and then verifying that they are true in all possible program states. Finally, _non-interference_ [Owicki & Gries 1976] is established, which ensures that execution of no process can invalidate assertions that appear in the sequential annotation of another. Each step will now be treated in detail.

3.3.2. Communications Axioms

In order to model buffered asynchronous communications, two multisets[3] are associated with each process D. The _send multiset_ for process D -- denoted σ_D -- contains a copy of every message that has been sent to D. Similarly, the _receive multiset_ for process D -- denoted ρ_D -- contains a copy of every message that has been received by D. A message can be received only if it has been sent and delivered. Therefore:

Network Axiom: $(\forall D\colon D \text{ a process}\colon \rho_D \subseteq \sigma_D)$.

The effect of executing:

send <expr> **to** D

is the same as the assignment:

$$\sigma_D := \sigma_D \oplus \langle expr \rangle,$$

where "$\sigma_D \oplus \langle expr \rangle$" denotes the multiset consisting of the elements of σ_D

---

[2] This is called _cooperation_ in [Apt _et al_. 1980].

[3] A _multiset_ -- sometimes called a "bag" -- is like a set, but it can contain more than one instance of the same element.

plus an element with value <expr>. Using the weakest precondition predicate transformer (wp) [Dijkstra 1976] with respect to the postcondition

$$T \wedge \sigma_D = \sigma_0 \oplus \langle expr \rangle,$$

we get an axiom for the **send** statement:

<u>Send Axiom</u>:

$$\{T^{\sigma_D}_{\sigma_D \oplus \langle expr \rangle} \wedge \sigma_D = \sigma_0\} \ \textbf{send} \ \langle expr \rangle \ \textbf{to} \ D \ \{T \wedge \sigma_D = \sigma_0 \oplus \langle expr \rangle\}.$$

When execution of the statement

$$\textbf{receive} \ m \ \textbf{when} \ \beta$$

in process D terminates, β evaluates to true, and a copy of the message received will be in ρ_D. In addition, depending on the particular message that was received, it may be possible to make some assertion about the state of the sender. An axiom that captures this is:

<u>Receive Axiom</u>:

$$\{P \wedge \rho_D = \rho_0\} \ \textbf{receive} \ m \ \textbf{when} \ \beta \ \{\beta \wedge Q \wedge \rho_D = \rho_0 \oplus m\}.$$

At first it may be disturbing that following a **receive** statement anything can be asserted, as indicated by the miraculous appearance[4] of Q in the postcondition. In the course of establishing satisfaction, restrictions are imposed on Q.

---

[4]See [Dijkstra 1976] for a discussion of the Law of the Excluded Miracle.

Should execution of r result in receipt of a message with text mtext, then this is equivalent to execution of the following multiple assignment statement:

$$m, \rho_D := mtext, \rho_D \oplus mtext.$$

For this assignment to establish the postcondition of the receive axiom, execution must be performed in a state that satisfies:

$$wp(\text{"}m,\rho_D := mtext,\rho_D \oplus mtext\text{"},\ \beta \wedge Q \wedge \rho_D{=}\rho_0 \oplus m\ ),$$

which is:

$$(\beta \wedge Q \wedge \rho_D{=}\rho_0 \oplus m)^{m,\rho_D}_{mtext,\rho_D \oplus mtext}$$

$$= (\beta^{m}_{mtext} \wedge Q^{m,\rho_D}_{mtext,\rho_D \oplus mtext} \wedge \rho_D = \rho_0).$$

Thus, the receive axiom will be sound with respect to our operational model provided

$$(\beta^{m}_{mtext} \wedge Q^{m,\rho_D}_{mtext,\rho_D \oplus mtext} \wedge \rho_D{=}\rho_0)$$

or equivalently,

$$Sat(mtext)\text{: } (pre(r) \wedge \beta^{m}_{mtext} \wedge mtext \in (\sigma_D \ominus \rho_D)) \Rightarrow Q^{m,\rho_D}_{mtext,\rho_D \oplus mtext}$$

is true of all program states.

The soundness of the proof system depends on the universal invariance of:

$$(\forall\, mtext\text{: } mtext \text{ a message that can be sent to D: } Sat(mtext)). \qquad (3.1)$$

In other words, receipt of any message that might be sent to D will establish the postcondition of the **receive**. Let s be a **send** statement that

3.3.3. Establishing Satisfaction

3.3.3.1. Satisfaction Invariants

Given a distributed program made up of processes D_1, D_2, ..., D_n, assume each process has been annotated using a sequential programming logic and the communications axioms above. Let $\overline{id}$ represent the list of identifiers that appear free in assertions in the sequential annotation. These identifiers name *auxiliary variables* and *program variables* [Clint 1973], where distinct variables are assumed to have distinct names. The values of program variables are stored in memory; auxiliary variables need not be since their values exert no influence over the execution of processes. Accordingly, program variables can be named in assertions in the sequential annotation of any process, but can appear only in statements in processes that have access to the memory in which these variables are stored. Allowing assertions in one process to refer to variables that are only accessible to another turns out to be quite important -- it allows the states of different processes to be correlated.

Consider a **receive** statement:

$$r\text{: } \textbf{receive } m \textbf{ when } \beta.$$

in process D. In order for the execution of r to result in the receipt of a message with text mtext, then (1) pre(r) must be true, (2) β must evaluate to true with $m = mtext$, and (3) a message with text mtext must have been sent to D but not yet received. Thus, immediately before mtext is assigned to m, the system state can be characterized by:

$$pre(r) \wedge \beta^{m}_{mtext} \wedge mtext \in (\sigma_D \ominus \rho_D),$$

where $\ominus$ is the multiset difference operator.

names D as its destination:

s: **send** <expr> **to** D.

The text of the message sent by executing s is determined by evaluating <expr> in the state that exists at the time it is executed. Thus, for messages sent to D due to execution of s, (3.1) becomes

$$(\forall \overline{v}, \mathrm{mtext}\colon \mathrm{pre}(s)^{\overline{id}}_{\overline{v}} \wedge \mathrm{mtext}{=}\langle\mathrm{expr}\rangle^{\overline{id}}_{\overline{v}}\colon \ \mathrm{Sat}(\mathrm{mtext}))$$

or equivalently,

$$\mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)\colon (\mathrm{pre}(s)^{\overline{id}}_{\overline{v}} \wedge \mathrm{mtext}{=}\langle\mathrm{expr}\rangle^{\overline{id}}_{\overline{v}} \wedge \mathrm{mtext} \in (\sigma_D \ominus \rho_D) \wedge \mathrm{pre}(r) \wedge \beta^{m}_{\mathrm{mtext}}) \Rightarrow Q^{m,\rho_D}_{\mathrm{mtext},\rho_D \oplus \mathrm{mtext}}.$$

There, $\overline{v}$ is a vector of values that models the program state at the time s is executed. $\mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)$ is called the <u>satisfaction invariant</u> for s and r.

We shall say that a **send** statement that names process D as its destination <u>matches</u> every **receive** statement in D. Then, to establish satisfaction:

<u>Satisfaction Rule</u>

For every **send** statement s and **receive** statement r that match, $\mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)$ is universally invariant.

3.3.3.2. <u>Showing Universal Invariance</u>

To establish satisfaction, each satisfaction invariant -- a formula of the form $\mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)$ -- must be shown to be true of every possible program state. There are several ways in which this can be done. One way is to show that $\mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)$ is a tautology. This is easily done if, for example, the Boolean guard in the **receive** statement r never evaluates to true for messages that can be sent by executing s. That is,

$$(\mathrm{pre}(s)_{\overline{v}}^{\overline{id}} \wedge \mathrm{mtext}{=}\langle\mathrm{expr}\rangle_{\overline{v}}^{\overline{id}} \wedge \beta^{m}_{\mathrm{mtext}}) \Rightarrow \text{false}.$$

A second way of showing a formula UI to be a universal invariant involves first showing that UI is true initially (i.e. true before any process has begun execution), and then proving that UI is invariant across execution of every statement s in the distributed program[5]. Formally, one proves:

For all statements s: $\{\mathrm{pre}(s) \wedge \mathrm{UI}\}\ s\ \{\mathrm{UI}\}$.

An equivalent, but somewhat simpler approach is based on proving non-interference with the satisfaction invariant. Let s_1, s_2, ..., s_m be the (atomic) statements of the process D_s that contains the **send** statement s. To show the universal invariance of the satisfaction invariant $\mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)$, first show that for all s_i:

$$\{\mathrm{pre}(s_i) \wedge \mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)\}\ s_i\ \{\mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)\}.$$

This is usually easy because the sequential annotation of D_s should contain enough information to prove that either the message that could be received by r from s has not yet been sent (i.e. $\mathrm{mtext} \in (\sigma_D \ominus \rho_D)$ is false), or that the message can be received and $Q^{m,\rho_D}_{\mathrm{mtext},\rho_D \oplus \mathrm{mtext}}$ is true. Then, the non-interference of other processes with $\mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)$ is shown. Clearly, it follows that $\mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)$ is true in all states of

[5]Statements are assumed to be atomic actions with respect to execution of other processes. In the absence of shared memory, assignment, **send**, **receive** and **skip** are all atomic. This would not necessarily be the case if there is shared memory. [Owicki & Gries 1976] give a syntactic characterization of when statements will appear atomic.

the computation.

3.3.4. Establishing Non-Interference

Lastly, since assertions in one process can can refer to variables changed by another, it is necessary to show that the execution of no process invalidates the proof of another. This is called non-interference [Owicki & Gries 1976].

An assertion P is parallel to a statement s if s is contained in one process and P is contained in a different concurrently executing process. To establish non-interference, it must be shown that execution of every statement parallel to P leaves P unchanged. That is, for all assertions P that are parallel to a statement s[6], prove:

$$\{P \wedge pre(s)\}\ s\ \{P\}.$$

Note, however, that for the case where s is a **receive** statement,

s: **receive** m **when** β

in process D (say), this involves a proof that:

$$\{P \wedge pre(s)\}\ s\ \{P\}$$

is a theorem. While that follows directly from the receive axiom, to preserve the soundness of the logic, satisfaction must be established for P as a postcondition of the **receive** in the theorem above. This is done by showing the universal invariance of:

---

[6] Again we assume that statements are atomic actions with respect to execution by other processes.

$$(\mathrm{pre}(\mathrm{snd})^{\overline{id}}_{\overline{v}} \wedge \mathrm{mtext}=\langle\mathrm{expr}\rangle^{\overline{id}}_{\overline{v}} \wedge (P \wedge \mathrm{pre}(s)) \wedge \beta^{m}_{\mathrm{mtext}} \wedge$$
$$\mathrm{mtext} \in (\sigma_D \ominus \rho_D)) \Rightarrow P^{m,\rho_D}_{\mathrm{mtext},\rho_D \oplus \mathrm{mtext}}$$

for every matching **send** statement snd of the form:

snd: **send** <expr> **to** D

The need for this stems from the fact that a **receive** can be viewed as implementing a decentralized assignment statement. Proof that its execution does not interfere with assertions that are parallel to it, is therefore necessary.

Although a non-interference proof could be a formidable task, there are situations where the amount of work can be reduced. If s does not modify variables named in P, or if P contains no references to shared or non-local variables, then non-interference follows immediately. Also, if

$$(P \wedge \mathrm{pre}(s)) \Leftrightarrow \mathrm{false},$$

then non-interference is trivially established, because s cannot be executed when the system is in a state satisfying P.

3.4. Safe Uses of Asynchronous send

The universal invariance of the satisfaction invariants ensures that the postcondition of a **receive** will indeed be true when a message is received. Unfortunately, establishing satisfaction is not always a simple task. The task is simplified if **send** and **receive** are used in a disciplined manner. Therefore, in this section we explore uses of asynchronous message passing primitives for which satisfaction is easily established. We do this with an extended example. Consider the following distributed programming problem.

A master process desires to broadcast the value of its variable mvar to a collection of slave processes, named $slave_1$, $slave_2$, ... $slave_N$. Available is a reliable communications network that allows the master to communicate with each of the slave processes[7].

3.4.1. Restricted Postconditions

A first solution to the broadcast problem has the master process sending V the value of mvar to each slave.

```
master: process;
        mvar := V;
        for i := 1 to N begin;
            {mvar = V}
            send mvar to slave_i
            end
        end
```

And, each slave process executes:

```
slave_i: process
         receive m_i when true
         {m_i = V}
         end
```

We would like to prove that after termination of all processes, V has been received by all of the slaves. As described earlier, such a proof involves three steps. First, a sequential annotation of each process is constructed. This is easily obtained from the proof outlines given above. Secondly, satisfaction must be established. To do this, a satisfaction invariant is constructed for the **send** in the master and the **receive** in each slave. For $slave_i$, it is:

[7] That is, we assume that every message sent is eventually delivered. While this is not a reasonable assumption, we make it here to simplify the problem.

$$(\mathrm{mtext} = V \;\wedge\; \mathrm{mtext} \in (\sigma_{\mathrm{slave}_i} \ominus \rho_{\mathrm{slave}_i})) \;\Rightarrow\; (m_i = V)^{m_i}_{\mathrm{mtext}}.$$

This is a tautology, and so true of all program states. Finally, non-interference follows trivially from the fact that no assertion in the sequential annotation contains variables modified by other processes.

This protocol illustrates that satisfaction follows when the postcondition of a **receive** does not reference variables modified by another process. Above, the postcondition of the **receive** statement was in terms of V -- the value of the message received -- not mvar, the variable in the master process whose value was sent. Consequently, execution by the sender cannot invalidate the postcondition of the **receive**. In general, the following can be said:

<u>Restricting Postconditions of Receive Statements</u>

Given matching statements r and s, where s is of the form:

s: **send** <expr> **to** D

if:

(1) $\mathrm{post}(r) \equiv Q_1 \wedge Q_2$

(2) $\mathrm{pre}(r) \Rightarrow Q_1$

(3) $(\mathrm{pre}(s)^{\overline{id}}_{\overline{V}} \wedge \mathrm{mtext}{=}\langle\mathrm{expr}\rangle^{\overline{id}}_{\overline{V}}) \Rightarrow (Q_2)^{m}_{\mathrm{mtext}}$

then $\mathrm{Satisfaction}_{\mathrm{Asynch}}(s,r)$ will be a tautology.

3.4.2. Monotonic Preconditions

For a variety of reasons it may be necessary to use the stronger assertion,

$$m_i = V \;\wedge\; \mathrm{mvar} = V$$

in the postcondition of the **receive** in a slave process. Then, execution in the slave is, in some sense, synchronized with that of the master. In particular, correct execution following the **receive** can be dependent on the agreement of the value of the slave's variable m_i and the value of mvar. Now, the code for a slave process is:

$slave_i$: **process**
    receive m_i **when** true
    $\{m_i = V \ \wedge \ mvar = V\}$
        .
        .
    computation requiring $mvar = V$
        .
        .
    end

As before, the sequential annotation of each process is straightforward. The satisfaction invariant for the **send** in the master and the **receive**, in light of the new (stronger) postcondition, is:

$$(mtext = V \ \wedge \ mtext \in (\sigma_{slave_i} \ominus \rho_{slave_i})) \ \Rightarrow \ (m_i = V \ \wedge \ mvar = V)^{m_i}_{mtext}.$$

This is equivalent to:

$$V \in (\sigma_{slave_i} \ominus \rho_{slave_i}) \ \Rightarrow \ mvar = V.$$

To see that this is true in all program states, notice that the following is invariant at the master process:

$$I_{master}: \ (\exists j: 1 \le j \le N: V \in \sigma_{slave_j}) \Rightarrow mvar = V.$$

That is, $mvar = V$ is true at the master starting from the time the message is sent to the first slave process. Since,

$$I_{master} \ \Rightarrow \ (V \in (\sigma_{slave_i} \ominus \rho_{slave_i}) \ \Rightarrow \ mvar = V),$$

is a tautology, the satisfaction invariant is true in all program states.

To complete the proof, non-interference must be shown. Only the assertion:

$$\{m_i = V \wedge mvar = V\}$$

in $slave_i$ names a variable modified by another process. The statements in the master parallel to this assertion are the **send** statement and the assignment: "mvar := V". Thus, it suffices to note that these statements do not invalidate the assertion in question.

In this second example a new use of message passing is illustrated -- the appearance of an assertion about the state of a sender in the postcondition of a **receive** statement. In addition to sending _values_ as in the first protocol above, the transmission of a message can also facilitate transfer of a _predicate_ from sender to receiver; in this case, mvar = V is transferred. Transfer of a predicate must be done with care so that subsequent execution by the sender does not interfere with the postcondition of the **receive**, before the message is received.

In our proof system, the universal invariance of the satisfaction invariants ensures that the sender will not invalidate the postcondition of a **receive**, while the non-interference proof ensures that the sender never invalidates other assertions in the receiver. One might expect that satisfaction invariants would be unnecessary, arguing that interference with the postcondition of a **receive** statement by a sender should be detected when performing a non-interference proof. Unfortunately, this is not the case. Because messages are buffered, a statement s in one process can interfere with the postcondition of a **receive** even if s cannot be executed while the receiver is waiting for a message. To see this, consider a process that sends a message to itself, and then invalidates the transferred predicate before executing a **receive**.

An assertion is <u>monotonic</u> if once it becomes true it remains so. The use of monotonic preconditions for **send** statements guarantees that satisfaction can be established. For example, above $mvar = V$ is implied by the precondition of the master's **send** statement. The assertion is monotonic -- it is implied by every subsequent state of the master process. Therefore, $mvar = V$ will be true when the message is received, regardless of delivery delays. Hence, $mvar = V$ can be asserted in the postcondition of the **receive** in a slave process. The general technique is:

<u>Monotonic Preconditions for Send Statements</u>

Let T be a predicate such that:

(1) it is implied by the postcondition of a **receive** statement r
(2) it is monotonic and
(3) it is implied by the precondition of a **send** statement s that can originate a message that might be received by r.

Then, $\text{Satisfaction}_{\text{Asynch}}(s,r)$ will be universally invariant.

3.4.3. Acknowledgment Messages

It is possible using message passing to transfer non-monotonic predicates between processes. Then, structure of the program must ensure that such a predicate will be true when the message is received. For example, in the protocol above consider the implications of changing the value of mvar after all of the messages have been transmitted. The precondition of the **send** statement is no longer monotonic. Also, I_{master} is no longer an invariant of the master process, and consequently the satisfaction invariant is no longer universally invariant. Not surprisingly, it is now possible that $mvar \neq V$ when a message is received by a slave -- an undesirable state of affairs. The master must be prevented from changing the value of mvar until all of the slaves have completed any processing requiring mvar = V. To facilitate this, each slave will transmit an acknowledgment message ack_i when it no longer requires that mvar = V.

$slave_i$: **process**
  $\{ack_i \notin \sigma_{master}\}$
r_s: **receive** m_i **when** true;
  $\{m_i = V \wedge mvar = V \wedge V \in \rho_{slave_i} \wedge ack_i \notin \sigma_{master}\}$
  .
  .
  computation requiring mvar = V
  .
  .
  .
  $\{m_i = V \wedge mvar = V \wedge V \in \rho_{slave_i} \wedge ack_i \notin \sigma_{master}\}$
s_s: **send** ack_i **to** master
  .
  no longer can assume mvar = V
  .
  end

The master process changes the value of mvar only after an acknowledgment is received from every slave:

master: **process;**
  mvar := V;
  for i := 1 **to** N **begin**
    {mvar = V ∧ SlavSent(V,i-1)}
s_m: **send** mvar **to** $slave_i$; {SlavSent(V,i)}
    end;
  for i := 1 **to** N **begin**
    {mvar = V ∧ SlavDone(V,i-1) ∧ SlavSent(V,N)}
r_m: **receive** ackmsg **when** ackmsg = 'ack_i'
    {mvar = V ∧ SlavDone(V,i)}
    end; {SlavDone(V,N)}
  mvar := newvalue; {mvar ≠ V}
  end

where:

$$\text{SlavSent}(v,S) \equiv (\forall\, i:\ 1 \le i \le S:\ v \in \sigma_{slave_i})$$

$$\text{SlavDone}(v,S) \equiv (\forall\, i:\ 1 \le i \le S:\ v \notin (\sigma_{slave_i} \ominus \rho_{slave_i}) \ \wedge\ ack_i \in \rho_{master})$$

These proof outlines are easily expanded to give a sequential annotation.

To establish satisfaction, $\text{Satisfaction}_{Asynch}(s_m, r_s)$ and $\text{Satisfaction}_{Asynch}(s_s, r_m)$ must be shown universally invariant for each slave. Below, we work out the details for $slave_i$ only.

$\text{Satisfaction}_{Asynch}(s_m, r_s)$ follows from the invariance of

$$V \in (\sigma_{slave_i} \ominus \rho_{slave_i}) \ \Rightarrow\ mvar = V.$$

at the master process. The second satisfaction invariant is

$$(ack_i \in (\sigma_{master} \ominus \rho_{master}) \ \wedge\ mvar = V \ \wedge\ \text{SlavDone}(V,i-1) \ \wedge\ \text{SlavSent}(V,N)) \ \Rightarrow\ mvar = V \ \wedge\ \text{SlavDone}(V,i),$$

which simplifies to

$$(ack_i \in (\sigma_{master} \ominus \rho_{master}) \ \wedge\ mvar = V \ \wedge\ \text{SlavDone}(V,i-1) \ \wedge\ \text{SlavSent}(V,N)) \ \Rightarrow\ V \in \rho_{slave_i}$$

This is true of all program states, because

$$ack_i \in \sigma_{master} \ \Rightarrow\ V \in \rho_{slave_i}$$

is an invariant at $slave_i$, and not interfered with by execution of the master.

Showing non-interference for this protocol involves showing that parallel execution does not invalidate assertions involving mvar, ρ_{slave_i}, and σ_{master}.

First, consider those assertions in $slave_i$ that name mvar. By design, such assertions appear only between r_s and s_s. The assertion

$$I: \ mvar = V \ \wedge \ ack_i \notin \sigma_{master}$$

appears at both the start and end of the code section, and cannot be affected by execution of $slave_i$. Therefore, it can be included as a conjunct in each intermediate assertion. Clearly, the only statement parallel to these assertions that could invalidate them is:

mvar := newvalue;

in the master process. However,

$$pre(\text{"mvar := newvalue"}) \ \Rightarrow \ ack_i \in \rho_{master}.$$

Hence, it follows from the Network Axiom that:

$$pre(\text{"mvar := newvalue"}) \ \Rightarrow \ ack_i \in \sigma_{master}.$$

Since $I \ \wedge \ pre(\text{"mvar:= newvalue"}) \Rightarrow false$, interference cannot occur. Thus, for all statements s between the **receive** and **send** in the slave, we have that:

$$\{I \wedge pre(s)\} \ s \ \{I\}$$

despite concurrent execution.

Next, consider assertions in the master process that contain references to the ρ_{slave_i} multisets. These assertions, derived from the predicate SlavDone, are always of the form:

$$P: \ V \notin (\sigma_{slave_i} \ominus \rho_{slave_i}).$$

Execution of the **receive** statement r_s in a slave is the only way to change the value of ρ_{slave_i}. However, that can only add a message to the multiset -- an action that cannot change the truth of P. Thus, no assertion naming ρ_{slave_i} can be invalidated by concurrent execution.

Lastly, the variable σ_{master} appears in assertions of slaves and is modified by execution of **send** statements that appear in slaves. However, the only modification made to σ_{master} by the execution of $slave_i$ is addition of the element "ack_i". Since assertions in $slave_i$ name σ_{master} only as part of the predicate " $ack_i \in \sigma_{master}$" no parallel execution by other slave processes will interfere with such assertions.

The key idea in this example is that the master maintained the truth of the predicate mvar = V until after the slaves finished any processing during which the truth of that predicate was required. Because the master has no way of knowing exactly when that time is, each slave sends an acknowledgment to communicate that fact. Thus, sending acknowledgments can be viewed as a way to ensure satisfaction: between the time the message is sent and its acknowledgment is received, the sender keeps the transferred predicate true, so that the transferred predicate can be asserted in the receiving process between the time the message is received and the acknowledgment sent. The general rule is:

Non-monotonic Assertions

> A message can be used to transfer any predicate from one process S to another R as long as the S ensures that the predicate is true at the time the message is received. R can inform S that the predicate is no longer required by returning an acknowledgment.

3.5. Discussion

3.5.1. The Syntax of **send** and **receive**

Our choice of syntax for **send** and **receive** is based on our desire to develop a general proof technique for programs that use message-passing. The statements had to be flexible enough to be useful for implementing more sophisticated message-passing operations.

It is easy to construct other message-passing operations in terms of our **send** and **receive**. This is due to the presence of the Boolean guard in the **receive** statement -- it allows a process to select which delivered message is received. For example, asynchronous message-passing statements in which messages have types and **receive** statements are type-specific can be implemented: a type field is included in each message and the guard is used to ensure that only one type of message can be received. A similar technique allows implementation of **receive** statements that accept messages only from a single process. Furthermore, our statements can be used to construct message-passing operations with more complicated synchronization. In [Schlichting & Schneider 1981], synchronous message-passing statements and remote procedure calls are implemented, while elsewhere in this dissertation we construct a receive with timeout and an operation that periodically retransmits a message until an acknowledgment is received.

The syntax for **send** and **receive** was also designed to simplify the proof rules. For example, requiring explicit naming of the destination process in a **send** statement allows matching **send-receive** pairs -- that is, those pairs for which satisfaction must be shown -- to be determined syntactically. The number of matching communication pairs could have been even further reduced if messages were sent to and received from *ports* [Solomon & Finkel 1979] [Baskett *et al*. 1977] rather than processes. Thus, there is theoretical justification for this construction.

3.5.2. Using Message-Passing

In this chapter, we have developed proof rules to allow the verification of programs using **send** and **receive**. Doing this yields insight into the difficulties of writing and verifying such programs. One major difficulty when an asynchronous **send** is used is coping with the interference that can arise. The sequential axiom for **receive** allows any conclusion to be made following receipt of a message. However, such a conclusion is valid only if the sending process does not interfere with the satisfaction invariants associated with the **receive**. Using message-passing in a disciplined manner controls such interference by restricting the actions a sending process can take following transmission of a message. The disciplines we explained simplify the satisfaction invariants, and therefore facilitate proof of universal invariance.

The need to perform satisfaction illustrates another difficulty with developing programs that communicate using message-passing: a process cannot be developed in isolation. While the sequential axioms allow anything to be concluded following a **receive** in process P, only postconditions for which satisfaction can be shown are valid in a proof. In general, developing such postconditions requires knowledge about the actions of all processes that send messages to P, especially if these postconditions are to contain assertions about the states of other processes.

In contrast, developing concurrent programs that use shared memory for communication and synchronization does not require such knowledge. While conclusions about other processes can also be made as a result of synchronization, these "miracles" can usually be determined by examining only the process executing the synchronization operation. For example, if statement S produces a state satisfying Q when started in a state satisfying $(P \wedge \beta)$, then the proof rule for an **await** statement [Owicki & Gries 1976] allows conclusion of

$$\{P\}\ \textbf{await}\ \beta\ \textbf{then}\ S\ \{Q\},$$

where logical expression β might involve variables changed by other processes. The "miracle" in this proof rule is that β can be asserted in pre(S). However, unlike the "miracle" following execution of a **receive**, this one can be determined solely by examining the text of the synchronization mechanism. No knowledge of the internal structure of other processes is required, so that programs using shared memory for communication should be easier to develop than programs that use message-passing.

Chapter 4

The Two-Phase Commit Protocol

4.1. Introduction

Suitably designed distributed systems can exhibit a high degree of fault-tolerance. Unfortunately, the protocols required to realize this are often complex. One way to cope with this complexity is to structure distributed computations in terms of *atomic actions*. A *single-site atomic action* is a computation that appears to execute instantaneously on a single processor. Should a failure occur during its execution, either the initial state or the final state of the computation is visible, *not* some intermediate state. This can be extended for use in systems involving more than one processor, as follows. A *multiple-site atomic action* (MSAA) is a collection of operations, each on a different processor P_1, P_2, ... P_N, where either all are performed or none are, despite failures at one or more sites.

One use for multiple-site atomic actions arises when a transaction running in a distributed system involves data stored at more than one site. In order to preserve the consistency of the database, transactions must always be executed in their entirety. Hence, if a portion of a transaction completes at one site, then it must be completed at all sites. Thus, a transaction can be viewed as a multiple-site atomic action.

A number of protocols have been developed to implement multiple-site atomic actions [Reed 1978], the most popular of which is the *two-phase commit protocol*. As described in [Gray 1978], this protocol ensures that the constituent operations are not executed at any site, or are executed at least once at every site. Here, we apply the theory developed in previous chapters to the two-phase commit protocol. This serves not only to establish the partial correctness of a well-known fault-tolerant distributed algorithm, but also to demonstrate the feasibility of applying our techniques to a non-trivial example.

4.2. The Protocol

The two-phase commit protocol is implemented by a single *coordinator process*, and one *worker process* at each site[1]. Execution is in two phases, as the name suggests. In the first phase, the coordinator sends a message of type *prepare* to each worker. Upon receipt of such a message, the worker determines if it can perform the operation requested in the *prepare* message, and replies accordingly. If the worker replies with an *agree* message, then this is construed by the coordinator as a commitment -- if subsequently asked, the worker will be able to perform the operation. It replies with a *refuse* message if it is unable to make such a commitment. If the coordinator receives at least one negative response, it broadcasts an *abort* message. Upon receiving an *abort* message, a worker rescinds any commitment made to perform the operation on demand, and acknowledges receipt with a message of type *abortack*. The operation remains undone. However, if all workers have replied with *agree* messages, the second phase is entered. There, the coordinator broadcasts a *commit* message. Upon receipt, each worker performs the operation, and sends a message of type *commitack* to the coordinator. Figure 4.1 summarizes the protocol.

---

[1] Including the site executing the coordinator.

```
Coordinator
 receive user request;
 broadcast prepare
    message;
 receive replies from
    all workers;
 if all agree →
    broadcast commit message;
    receive commitack from
        all workers;
 □ at least one refuse →
    broadcast abort;
    receive abortack from
        all workers;
 fi;
```

```
Worker
 receive prepare from coordinator;
 if can commit to perform operation →
     prepare for operation;
     send agree to coord;
  □ cannot commit to perform
                  operation →
     send refuse to coordinator;
 fi;
 receive verdict from coordinator;
 if verdict = commit →
     perform operation;
     send commitack to coordinator;
  □ verdict = abort →
     send abortack to coordinator;
 fi;
```

Figure 4.1 -- Two-Phase Commit Protocol

The consequences of a failure at the coordinator depend on how far it has progressed at the time of the failure. If the failure occurs after the coordinator has begun sending commit messages, then a recovery protocol must complete that broadcast, thereby ensuring that every worker is notified that it should perform its operation. Otherwise, an abort message should be broadcast. As before, this causes the workers to rescind their commitments, and leave the operation undone. Unfortunately, the workers must remain committed to perform the operation for the entire period of the coordinator's failure. Clearly, this is a drawback of this protocol.

A failure at a worker's site has no effect on whether the coordinator commits or aborts the MSAA. The coordinator retransmits a message if no response is received. Thus, after a failure, a worker need only wait for the next message. If the message is a prepare message, then the entire protocol is reexecuted, but if it is either commit or abort, then an acknowledgment is sent after performing the requested action. The recovery

```
Coordinator                        Worker
 if have sent at least              receive message;
   1 commit message →               continue as in Figure 4.1
     broadcast commit;                 based on type of message;
     receive commitack from
         all workers;
  □ otherwise →
     broadcast abort;
     receive abortack from
         all workers;
 fi;
```

Figure 4.2 -- Recovery Protocols

protocol for the coordinator and worker are shown in Figure 4.2.

4.3. Communication

The two-phase commit protocol is an event driven protocol. When a message is received, its contents describe an event that took place at the sender's site. Based on that information, the receiving process performs some local action, replies to the message, and waits for another message. Unfortunately, if messages can be lost or processors can fail, there is no guarantee that the awaited reply will be forthcoming.

This problem can be avoided using **receive** statements with _timeouts_ [Lamport 1981]. A process executes a **receive** statement to wait for a reply, and if no such reply is received within a certain amount of time, execution of the **receive** terminates. This allows the sending process to take some application-dependent remedial action. In the two-phase commit protocol, this action involves retransmitting the message and again waiting for a reply.

4.3.1. A Receive with Timeout Operation

Execution of the **receive** statement described in chapter 3 never times out. However, using primitives already defined, it is straightforward to construct rectimeout -- a receive operation whose execution will time out t seconds after being invoked.

Assume that messages sent by a process to itself are never lost, and that messages have three fields -- content, type, and source. Then, a rectimeout operation that allows process myid to wait for a message from process pid is shown in Figure 4.3. Note that we do not advocate using this implementation -- it is obviously simplier to define a new **receive** primitive whose execution is terminated by the system after t seconds. However, it is more convenient to use this rectimeout implementation when verifying a program, since no new proof rules are required.

The timeout message can be received only during execution of the **cobegin** in which it is sent. This is because each timeout message has a unique text: the type timeout distinquishes it from non-timeout messages

---

```
rectimeout(m,β,myid,pid,tocount):
   tocount:= tocount+1;
   cobegin;
      receive m[content,type,source] when
                  ((β ∧ m.type ≠ timeout ∧ m.source = pid) ∨
                   (m.content = tocount ∧ m.type = timeout));
   // delay(t);
      send [tocount,timeout,myid] to myid;
   coend;
   end;
```

Figure 4.3 -- Receive with Timeout Operation

sent by that process, and the integer value tocount associates it with that execution of the **cobegin**.

A sequential annotation of this operation appears in Figure 4.4. The postcondition of **receive** statement r can be informally explained as follows: either a non-timeout message from pid has been received and Q can be asserted, or the timeout message was received and T remains true. This is because receipt of the timeout message conveys no information at all -- it simply terminates the **receive** statement.

$\{\sigma_{myid} = \sigma_0 \wedge \rho_{myid} = \rho_0 \wedge \text{tocount} = t_0 \wedge T\}$
rectimeout(m,β,myid,pid,tocount):
  tocount:= tocount+1;
  cobegin;
    $\{\rho_{myid} = \rho_0 \wedge \text{tocount} = t_0{+}1 \wedge T\}$
    r: **receive** m[content,type,source] **when**
      $((\beta \wedge \text{m.type} \neq \text{timeout} \wedge \text{m.source} = \text{pid}) \vee$
      $(\text{m.content} = \text{tocount} \wedge \text{m.type} = \text{timeout}))$;
    $\{\rho_{myid} = \rho_0 \oplus \text{m} \wedge \text{tocount} = t_0{+}1 \wedge$
      $((\beta \wedge \text{m.type} \neq \text{timeout} \wedge \text{m.source} = \text{pid} \wedge Q) \vee$
      $(\text{m.content} = \text{tocount} \wedge \text{m.type} = \text{timeout} \wedge T))\}$
  // $\{\sigma_{myid} = \sigma_0 \wedge \text{tocount} = t_0{+}1\}$
    delay(t);
    s: **send** [tocount,timeout,myid] **to** myid;
    $\{\sigma_{myid} = \sigma_0 \oplus [t_0{+}1,\text{timeout},\text{myid}] \wedge \text{tocount} = t_0{+}1\}$
  coend;
  end;
$\{\rho_{myid} = \rho_0 \oplus \text{m} \wedge \sigma_{myid} = \sigma_0 \oplus [t_0{+}1,\text{timeout},\text{myid}] \wedge \text{tocount} = t_0{+}1 \wedge$
  $((\beta \wedge \text{m.type} \neq \text{timeout} \wedge \text{m.source} = \text{pid} \wedge Q) \vee$
  $(\text{m.content} = t_0{+}1 \wedge \text{m.type} = \text{timeout} \wedge T))\}$

Figure 4.4 -- Sequential Annotation of Rectimeout Operation

When proving the correctness of a program in which rectimeout operations occur, certain proof obligations must be satisfied. It must be shown that:

(1) $\text{Satisfaction}_{\text{Asynch}}(s',r)$ is universally invariant for every send statement s' in process pid.

(2) Concurrent execution by other processes does not interfere with the sequential annotation of rectimeout.

(3) $\text{Satisfaction}_{\text{Asynch}}(s,r)$ is universally invariant.

(4) There is no interference within the **cobegin**.

Since (1) and (2) depend on the context in which the operation is used, nothing further can be said about them here. (3) and (4) can be simplified as follows.

By definition, $\text{Satisfaction}_{\text{Asynch}}(s,r)$ is:

$$\begin{aligned}&(\text{pre}(s)^{\overline{id}}_{\overline{v}} \wedge \text{mtext} = [\text{tocount},\text{timeout},\text{myid}]^{\overline{id}}_{\overline{v}} \wedge \text{pre}(r) \wedge \\ &\quad ((\beta \wedge \text{m.type} \neq \text{timeout} \wedge \text{m.source} = \text{pid}) \vee \\ &\quad\ (\text{m.content} = \text{tocount} \wedge \text{m.type} = \text{timeout}))^{m}_{\text{mtext}} \wedge \\ &\quad \text{mtext} \in (\sigma_{\text{myid}} \ominus \rho_{\text{myid}})) \Rightarrow \\ &(\text{tocount} = t_0+1 \wedge ((\beta \wedge \text{m.type} \neq \text{timeout} \wedge \text{m.source} = \text{pid} \wedge Q) \vee \\ &\quad (\text{m.content} = \text{tocount} \wedge \text{m.type} = \text{timeout} \wedge T)))^{m,\,\rho_{\text{myid}}}_{\text{mtext},\,\rho_{\text{myid}} \oplus \text{mtext}}.\end{aligned}$$

This simplifies to

$$\begin{aligned}&(\text{pre}(r) \wedge [\text{tocount},\text{timeout},\text{myid}] \in (\sigma_{\text{myid}} \ominus \rho_{\text{myid}})) \Rightarrow \\ &T^{m,\,\rho_{\text{myid}}}_{[\text{tocount},\,\text{timeout},\text{myid}],\,\rho_{\text{myid}} \oplus [\text{tocount},\,\text{timeout},\text{myid}]}.\end{aligned}$$

Hence, to establish $\text{Satisfaction}_{\text{Asynch}}(s,r)$, it is sufficient to show:

$$PO_1:\ \text{pre}(r) \Rightarrow T^{m,\,\rho_{\text{myid}}}_{[\text{tocount},\,\text{timeout},\text{myid}],\,\rho_{\text{myid}} \oplus [\text{tocount},\text{timeout},\text{myid}]}.$$

Note that for all **receive** statements r_i, $r_i \neq r$, $\text{Satisfaction}_{\text{Asynch}}(s,r_i)$ is trivially true in every state. This is because having timeout messages with unique texts ensures that messages sent by s cannot satisfy the Boolean guard on any **receive** statement but r.

To satisfy the fourth requirement, it is necessary to show that execution of neither statement list within the **cobegin** interferes with assertions in the other. Since execution of r does not modify either σ_{myid} or tocount, it follows trivially that such execution cannot invalidate pre(s) or post(s). Therefore, it remains to show:

$$\{pre(s) \wedge pre(r)\}\ s\ \{pre(r)\}$$
$$\{pre(s) \wedge post(r)\}\ s\ \{post(r)\}.$$

And so, it is sufficient to prove:

$$PO_2\text{: } (pre(s) \wedge T) \Rightarrow T^{\sigma_{myid}}_{\sigma_{myid} \oplus [tocount, timeout, myid]}$$
$$(pre(s) \wedge Q) \Rightarrow Q^{\sigma_{myid}}_{\sigma_{myid} \oplus [tocount, timeout, myid]}.$$

4.3.2. Retransmissions

Given a rectimeout operation, it is possible to implement a structured message-passing operation -- the communicate -- that is well suited for event driven protocols. The code for a communicate operation is shown in Figure 4.5. When executed, a message with content "cont", type "sendt", and source S is transmitted to D, the destination process. Then, a reply with content "ncont" and of any type found in the set "ntype" is awaited. The value of the message is assigned to "msg". If no such message is received after t seconds, the message is resent to D.

$\{\sigma_D = \sigma_0 \wedge \rho_S = \rho_0 \wedge T\}$
communicate(cont , sendt, S, msg, ncont, ntype, D, toc):
    $\{\sigma_D = \sigma_0 \wedge \rho_S = \rho_0 \wedge T \wedge \text{sendt} \neq \text{timeout}\}$
    s: **send** [cont,sendt,S] **to** D;
    $\{(\sigma_0 \oplus [\text{cont,sendt,S}]) \subseteq \sigma_D \wedge \rho_0 \subseteq \rho_S \wedge T \wedge \text{sendt} \neq \text{timeout}\}$
    rectimeout(msg,(msg.type $\in$ ntype $\wedge$ msg.content = ncont),S,D,toc);
    $\{I\text{: } (\sigma_0 \oplus [\text{cont,sendt,S}]) \subseteq \sigma_D \wedge (\rho_0 \oplus \text{msg}) \subseteq \rho_S \wedge \text{sendt} \neq \text{timeout} \wedge$
        $((\text{msg.type} \in \text{ntype} \wedge \text{msg.type} \neq \text{timeout} \wedge \text{msg.content} = \text{ncont} \wedge Q) \vee$
         $(\text{msg.type} = \text{timeout} \wedge T))\}$
    do msg.type = timeout $\rightarrow$ $\{I \wedge \text{msg.type} = \text{timeout}\}$
        send [cont,sendt,S] **to** D;
        rectimeout(msg,(msg.type $\in$ ntype $\wedge$ msg.content = ncont),S,D,toc);
    od;
    end
$\{(\sigma_0 \oplus [\text{cont,sendt,S}]) \subseteq \sigma_D \wedge (\rho_0 \oplus \text{msg}) \subseteq \rho_S \wedge \text{msg.type} \in \text{ntype} \wedge$
    $\text{msg.content} = \text{ncont} \wedge Q\}$

Figure 4.5 -- Annotated Communicate Operation

When proving a program that contains communicate operations, certain proof obligations are incurred. First, we assumed in the sequential annotation in Figure 4.5 that T was invariant across execution of both sends in the communicate. This will later be established by use of a replete proof outline for the coordinator and worker processes.

Secondly, note that "sendt $\neq$ timeout" was asserted in pre(s) and throughout the remainder of the operation. Therefore, for every communicate operation c, we require:

PO_3: pre(c) $\Rightarrow$ (sendt $\neq$ timeout).

Lastly, for the postcondition of a communicate to hold upon termination, assumptions made in the postconditions of the **receive** statements in each of the rectimeout operations must be validated by proving satisfaction. Let r_1 and r_2 be **receive** statements in a communicate operation c in process S interacting with process D. Then, $\text{Satisfaction}_{\text{Asynch}}(s'', r_1)$ and $\text{Satisfaction}_{\text{Asynch}}(s'', r_2)$ must be proved universally invariant for all **send** statements s" in D. To do this, we define a meta-satisfaction rule for constructing satisfaction invariants. Establishing the universal invariance of these invariants is sufficient to prove the universal invariance of $\text{Satisfaction}_{\text{Asynch}}(s'', r_1)$ and $\text{Satisfaction}_{\text{Asynch}}(s'', r_2)$. The meta-satisfaction rule is:

Communicate Meta-Satisfaction Rule:

A communicate operation cl matches another communicate c2 if the destination in the argument list of cl names the process in which c2 appears. For all matching communicate operations cl in S and c2 in D for which the following triples have been proved to be theorems:

$$\{\sigma_D = \sigma 1_0 \wedge \rho_S = \rho 1_0 \wedge T1\}$$

$$\text{c1: communicate(cont1,st1,S,msg1,ncnt1,nt1,D,toc1)}$$

$$\{(\sigma 1_0 \oplus [\text{cont1,st1,S}]) \subseteq \sigma_D \wedge (\rho 1_0 \oplus \text{msg1}) \subseteq \rho_S \wedge \text{msg1.type} \in \text{nt1} \wedge \text{msg1.content} = \text{ncnt1} \wedge Q1\}$$

$$\{\sigma_S = \sigma 2_0 \wedge \rho_D = \rho 2_0 \wedge T2\}$$

$$\text{c2: communicate(cont2,st2,D,msg2,ncnt2,nt2,S,toc2)}$$

$$\{(\sigma 2_0 \oplus [\text{cont2,st2,D}]) \subseteq \sigma_S \wedge (\rho 2_0 \oplus \text{msg2}) \subseteq \rho_D \wedge \text{msg2.type} \in \text{nt2} \wedge \text{msg2.content} = \text{ncnt2} \wedge Q2\},$$

it is sufficient to show the following universally invariant to conclude post(c2) after execution of c2 completes:

$$\begin{aligned}&\text{Com\_Sat(c1,c2): }((T1 \wedge \text{st1} \neq \text{timeout})^{\overline{\text{id}}}_{\overline{\text{v}}} \wedge \\ &\quad \text{mtext} = [\text{cont1,st1,s}]^{\overline{\text{id}}}_{\overline{\text{v}}} \wedge T2 \wedge \text{st2} \neq \text{timeout} \wedge \\ &\quad (\beta\text{: mtext.type} \in \text{nt2} \wedge \text{mtext.content} = \text{ncnt2} \wedge \\ &\quad \text{mtext.source} = S \wedge \text{mtext.type} \neq \text{timeout}) \wedge \\ &\quad \text{mtext} \in (\sigma_D \ominus \rho_D)) \Rightarrow Q2^{\text{msg2},\rho_D}_{\text{mtext},\rho_D \oplus \text{mtext}}.\end{aligned}$$

The complete sequential annotation of a communicate operation and the derivation of this rule appear in Appendix 1.

At times it will be convenient for **receive** statements in isolation to receive messages from communicates. Then, the meta-satisfaction rule for a communicate-**receive** pair should be used:

One-Way Communicate Meta-Satisfaction Rule:

A communicate operation matches a **receive** statement if the destination in its argument list names the process containing the **receive**. For every matching communicate-**receive** pair c,r for which the theorems

$$\{\sigma_D = \sigma_0 \wedge \rho_S = \rho_0 \wedge T\}$$
c: communicate(cont, sendt, S, msg, ncont, ntype, D, toc);
$$\{(\sigma_0 \oplus [\mathrm{cont},\mathrm{sendt},S]) \subseteq \sigma_D \wedge (\rho_0 \oplus \mathrm{msg}) \subseteq \rho_S \wedge \mathrm{msg.type} \in \mathrm{ntype} \wedge \mathrm{msg.content} = \mathrm{ncont} \wedge Q\}$$

and

$$\{\rho_D = \rho l_0 \wedge P\}$$
r: **receive** m[content,type,source] **when** β
$$\{\rho_D = \rho l_0 \oplus m \wedge \beta \wedge Q\},$$

have been proved, the following must be universally invariant for post(r) to hold after execution of r completes:

$$\mathrm{OWCom\_Sat}(c,r){:}\ ((T \wedge \mathrm{sendt} \neq \mathrm{timeout})^{\overline{id}}_{\overline{v}} \wedge \mathrm{mtext} = [\mathrm{cont},\mathrm{sendt},S]^{\overline{id}}_{\overline{v}} \wedge \mathrm{pre}(r) \wedge \beta^{m}_{\mathrm{mtext}} \wedge \mathrm{mtext} \in (\sigma_D \ominus \rho_D)) \Rightarrow Q^{m,\rho_D}_{\mathrm{mtext},\rho_D \oplus \mathrm{mtext}}.$$

The derivation of this rule is similar to that of the Communicate Satisfaction Rule.

4.4. Implementation and Sequential Annotation

The outline of the two-phase commit protocol given in section 2 must be modified slightly to be implemented. The problem is that a worker process cannot be sure its abortack or commitack message has been received, since no reply is awaited after these messages are transmitted. The solution is to structure each process as an infinite loop. The loop body implements the protocol described in section 4.2. Then, a prepare message sent by the coordinator for an MSAA can be viewed as the reply to the commitack or abortack message associated with the previous MSAA. So that messages with the same type but from different MSAAs can be distinquished, each MSAA is given a unique name. In our implementation, names are positive integers stored in the content field of every message.

To simplify exposition, we assume that the operation to be performed by MSAA i at site j is

$$\text{done}_j(i) := \text{true}.$$

Preparation for this operation is assumed to be

$$\text{prepared}_j(i) := \text{true}.$$

Then, an implementation of the two-phase commit protocol should attempt to maintain the invariance of

$$(\forall\, i\text{: } i \text{ an MSAA: } (\forall\, j,k\text{: } j,k \text{ a site: } \text{done}_j(i) = \text{done}_k(i))).$$

The text of all messages sent to the coordinator are saved in the send multiset σ_c. However, in the sequential annotation of the workers and the coordinator, we will often want to refer to only that portion of σ_c containing messages sent by a particular worker process. Let $\sigma_c[j]$ be the multiset consisting of all the message texts sent by worker_j to the coordinator. Likewise, let $\sigma_j[c]$ be the multiset consisting of all message texts sent by the coordinator to worker_j.

The implementation of a worker process consists of an infinite loop and some initialization. The body of this loop is a fault-tolerant action implementing the protocols of Figures 4.1 and 4.2. To reduce the size of the worker's sequential annotation, we first define some abbreviations.

Let NDWj(i) be the assertion that $worker_j$ has not executed its operation in any MSAA named i or greater, and NSAAj(i) the assertion that $worker_j$ has not sent an abortack message to the coordinator for any MSAA named i or greater. That is:

$$NDWj(i) \equiv (\forall h:\ h \geq i:\ \neg done_j(h))$$
$$NSAAj(i) \equiv (\forall h:\ h \geq i:\ [h,abortack,j] \notin \sigma_c[j]).$$

A universal invariant for each worker process can also be defined. For each MSAA k, the following are always true:

(1) If the agree message has been sent and the abort message has not been received, then $prepared_j(k) = true$.

(2) If $done_j(k) = true$, then the commit message has been received.

(3) If the commitack message has been sent, then $done_j(k) = true$.

(4) If the abortack message has been sent, then $done_j(k) = false$.

This invariant can be expressed formally as:

$$\begin{aligned} UIj \equiv (\forall k:\ k \geq 1:\ & \\ & ([k,agree,j] \notin \sigma_c[j] \vee [k,abort,cj] \in \rho_j \vee prepared_j(k)) \wedge \\ & ([k,commit,cj] \in \rho_j \vee \neg done_j(k)) \wedge \\ & ([k,commitack,j] \notin \sigma_c[j] \vee done_j(k)) \wedge \\ & ([k,abortack,j] \notin \sigma_c[j] \vee \neg done_j(k))). \end{aligned}$$

Note that the source field of a message sent from the coordinator to $worker_j$ is cj -- a value that will differ for each worker. As is explained below, this is needed to implement broadcasts using communicate operations. Lastly, we define a loop invariant for $worker_j$ as follows:

$$\begin{aligned} Ij \equiv (& NDWj(msg\_name) \wedge msg\_name = name \wedge NSAAj(msg\_name) \wedge toc_j > 0 \wedge \\ & UIj \wedge (msg.type = prepare \vee msg.type = abort)). \end{aligned}$$

name identifies the name of the MSAA with which the next message to be received by $worker_j$ is associated, msg_name identifies the name of the MSAA

```
worker_j: action,recovery;
   var name, msg_name, toc_j: integer;
   {ρ_j = ∅ ∧ σ_c[j] = ∅ ∧ NDWj(1) ∧ NSAAj(1) ∧ UIj}
   toc_j, name := 1, 1;
   loop: action,recovery
      {NDWj(name) ∧ NSAAj(name) ∧ toc_j > 0 ∧ UIj}
      rl: receive msg[content,type,source] when
                       (msg.content = name ∧ msg.source = cj ∧
                        (msg.type = prepare ∨ msg.type = abort));
      msg_name := name;
      {Ij}
      do true → {Ij}
         wexecute: action;
                     "from Figure 4.7" {Ij}
                  recovery
                     "from Figure 4.8" {Ij}
                  end wexecute; {Ij}
      od;
      end loop;
   end worker_j
```

Figure 4.6 -- Initialization of Worker$_j$

---

with which the last message received by worker$_j$ is associated, and toc$_j$ is used in the implementation of communicate operations in worker$_j$. The variable msg is assumed to be declared as follows:

```
var msg: record
           content: integer;
           type: character;
           source: integer;
         end.
```

The initialization portion of worker$_j$ is shown in Figure 4.6.

After receiving a message of type <u>prepare</u>, a worker must decide whether it can commit to executing the operation. To do this, the following function is used:

$$\text{possible}_j(i) \equiv \begin{cases} \text{true} & \text{if worker}_j \text{ can commit to being able to perform the operation requested in MSAA } i \\ \text{false} & \text{otherwise} \end{cases}$$

Figure 4.7 gives the partially annotated code for this section of worker_j.

Notice that

```
msg_name := name+1
```

is part of each fault-tolerant action labelled aa or ca, and that the value of msg_name is assigned to name after each action terminates. The use of these two statements to effect the incrementation of name is necessary to avoid the possibility of worker_j waiting forever for a message that is never sent. To see this, consider what would happen if

```
name := name+1
```

was inserted as the last statement in the body of the action, and the two statements described above were removed. Should a failure now occur between the time the incrementation of name terminates and the action is exited, the statement list would be reexecuted with name having a value one greater than it had the previous time. But this means that worker_j would wait for a message from the coordinator that is not associated with the current MSAA being executed. Hence, the communicate operation would never terminate and deadlock would result. Using two statements avoids this problem, since name retains its value until <u>after</u> the fault-tolerant action terminates.

```
wexecute: action; {Ij}
  if msg.type = abort →
    aa: action,recovery;
      {NDWj(name) ∧ (msg_name = name ∨ msg_name = name+1) ∧ toc_j > 0 ∧
        NSAAj(name+1) ∧ UIj}
      c1: communicate(name,abortack,j,msg,name+1,{prepare,abort},
                      cj,toc_j);
      {NDWj(name) ∧ (msg_name = name ∨ msg_name = name+1) ∧ toc_j > 0 ∧
        NSAAj(name+1) ∧ UIj ∧ msg.content = name+1 ∧
        (msg.type = prepare ∨ msg.type = abort)}
      msg_name := name+1; end;
    name := msg_name; {Ij}
  □ msg.type = prepare ∧ possible_j(name) →
    prepared_j(name) := true;
    ag: action,recovery;
      {NDWj(msg_name) ∧ msg_name = name ∧ prepared_j(name) ∧ toc_j > 0 ∧
        NSAAj(msg_name) ∧ UIj}
      c2: communicate(name,agree,j,msg,name,{commit,abort},
                      cj,toc_j); end;
    if msg.type = commit →
      ca: action,recovery;
        {NDWj(name+1) ∧ prepared_j(name) ∧ NSAAj(name) ∧ UIj ∧ toc_j > 0 ∧
          (msg_name = name ∨ msg_name = name+1) ∧ [name,commit,cj] ∈ ρ_j}
        done_j(name) := true;
        c3: communicate(name,commitack,j,msg,name+1,{prepare,abort},
                        cj,toc_j);
        msg_name := name+1; end;
      name := msg_name; {Ij}
    □ msg.type = abort →
      {NDWj(msg_name) ∧ NSAAj(msg_name) ∧ msg_name = name ∧ UIj ∧
        toc_j > 0 ∧ prepared_j(name) ∧ [name,abort,cj] ∈ ρ_j}
      prepared_j(name):= false;
      aa: action,recovery; "same as aa FTA above"; end
      name := msg_name; {Ij}
    fi; {Ij}
  □ msg.type = prepare ∧ ¬possible_j(name) →
    c5: communicate(name,refuse,j,msg,name,{abort},cj,toc_j);
    aa: action,recovery; "same as aa FTA above"; end
    name := msg_name; {Ij}
  fi; {Ij}
```

Figure 4.7 -- Action Portion of wexecute

```
recovery
   {NDWj(msg_name) ∧ NSAAj(msg_name) ∧ UIj ∧ toc_j > 0}
   r2: receive msg[content,type,source] when
                       (msg.content = msg_name ∧ msg.source = cj ∧
                       msg.type ∈ {prepare,commit,abort});
   {NSAAj(msg_name) ∧ NDWj(msg_name) ∧ UIj ∧ toc_j > 0 ∧
      (((msg.type = prepare ∨ msg.type = abort) ∧
            [msg_name,msg.type,cj] ∈ ρ_j) ∨
      (msg.type = commit ∧ [msg_name,commit,cj] ∈ ρ_j ∧
        [msg_name,abort,cj] ∉ σ_j ∧ [msg_name,agree,j] ∈ ρ_c[j]))}
   name := msg_name;
   if msg.type = prepare ∨ msg.type = abort →
       skip; {Ij}
    □ msg.type = commit →
      {NDWj(name) ∧ NSAAj(name) ∧ UIj ∧ toc_j > 0 ∧ msg_name = name ∧
         prepared_j(name) ∧ [name,commit,cj] ∈ ρ_j}
      ca: action,recovery
         done_j(name) := true;
         c7: communicate(name,commitack,j,msg,name+1,
                                  {prepare,abort},cj,toc_j);
         msg_name := name+1; end;
      {NDWj(msg_name) ∧ NSAAj(msg_name) ∧ UIj ∧ msg_name = name+1 ∧
         prepared_j(name) ∧ done_j(name) ∧ [name,commit,cj] ∈ ρ_j ∧
         (msg.type = prepare ∨ msg.type = abort) ∧ toc_j > 0}
      name := msg_name; {Ij}
   fi; {Ij}
```

Figure 4.8 -- Recovery Protocol of wexecute

The recovery protocol for wexecute, which reestablishes the loop invariant Ij, is given in Figure 4.8.

Appendix 2 contains the sequential annotation of the entire worker.

To communicate with workers, the coordinator broadcasts messages to the workers and then waits for their replies. The annotated broadcast operation in Figure 4.9 implements this. Notice that a separate timeout count -- ctoc(j) -- is maintained for each communicate within the

cobegin, and that the source included in messages sent to $worker_j$ is cj. This ensures that a message of type timeout can be received only during execution of the rectimeout in which it is sent.

Proof obligations are incurred when proving a program containing this operation. The satisfaction invariants associated with the communicates within a broadcast must be proved universally invariant. Further, non-interference of other processes with the proof of this operation must be established. Finally, non-interference of the operation with itself must be shown.

If the form of the broadcast is as given in Figure 4.9, then establishing

---

$\{(\forall\ j\colon j \text{ a site}\colon R(j) \wedge \sigma_j = \sigma 0_j \wedge \rho_c[j] = \rho 0_j)\}$

B: broadcast(sendtype,msg(1..N),rectype,req#):

cobegin

...

//$\{R(j) \wedge \sigma_j = \sigma 0_j \wedge \rho_c[j] = \rho 0_j\}$

for all sites j: B_j: communicate(req#,sendtype,cj,msg(j),req#,rectype,j,ctoc(j));

$\{Q(j) \wedge (\sigma 0_j \oplus [req\#, sendtype, cj]) \subseteq \sigma_j \wedge (\rho 0_j \oplus msg(j)) \subseteq \rho_c[j] \wedge msg(j).content = req\# \wedge msg(j).type \in rectype\}$

//

...

coend;

end;

$\{(\forall\ j\colon j \text{ a site}\colon Q(j) \wedge (\sigma 0_j \oplus [req\#, sendtype, cj]) \subseteq \sigma_j \wedge (\rho 0_j \oplus msg(j)) \subseteq \rho_c[j] \wedge msg(j).content = req\# \wedge msg(j).type \in rectype)\}$

Figure 4.9 -- Broadcast Operation

$$PO_4 : \{R(j) \wedge pre(B_i)\}\ B_i\ \{R(j)\} \qquad \text{for}$$
$$\{Q(j) \wedge pre(B_i)\}\ B_i\ \{Q(j)\} \qquad \text{each } j \neq i$$

where B_i is a communicate operation within the **cobegin**, is sufficient to prove non-interference within the broadcast.

To annotate the coordinator, define DONE(i) to be true when the operation associated with each MSAA named less than i has been requested by the coordinator and either executed at all sites or executed at no site. That is, DONE(i) is the assertion:

$$\begin{aligned}DONE(i) \equiv (\forall h\!: 1 \le h < i\!:\ & ((\forall j\!: j \text{ a site}\!: \neg done_j(h) \wedge [h,abortack,j] \in \rho_c[j]) \vee \\ & (\forall j\!: j \text{ a site}\!: done_j(h) \wedge [h,commitack,j] \in \rho_c[j]))).\end{aligned}$$

In addition, let NSA(i) be the assertion that no abort message concerning an MSAA named i or greater has been sent, NSC(i) an analogous assertion for commit messages, and TOC the assertion that each timeout count variable has a value greater than 0:

$$NSA(i) \equiv (\forall j,k\!: j \text{ a site} \wedge k \ge i\!: [k,abort,cj] \notin \sigma_j)$$
$$NSC(i) \equiv (\forall j,k\!: j \text{ a site} \wedge k \ge i\!: [k,commit,cj] \notin \sigma_j)$$
$$TOC \equiv (\forall j\!: j \text{ a site}\!: ctoc(j) > 0).$$

Finally, let UIC be a universal invariant associated with the coordinator and Ic a loop invariant that is the desired postcondition of cexecute:

$$\begin{aligned}UIC \equiv (\forall j,k\!: j \text{ a site} \wedge i \ge 1\!: ([i,commit,cj] \notin \sigma_j \vee \\ ([i,agree,j] \in \rho_c[j] \wedge [i,abort,cj] \notin \sigma_j)))\end{aligned}$$

$$\begin{aligned}Ic \equiv (status = abort \wedge DONE(req\#) \wedge value = req\# \wedge NSC(req\#) \wedge \\ NSA(req\#) \wedge TOC \wedge UIC).\end{aligned}$$

Then, Figure 4.10 shows the initialization of the coordinator and the action portion of the fault-tolerant action constituting the loop body.

```
coord: action,recovery
   var status: character, value, req#, ctoc(1..N): integer;
   {(∀ j: j a site: σj[c] = ∅) ∧ ρc = ∅}
   ctoc, req#, value := 1, 1, 1; status := abort
   loop: action,recovery;
      do true → {Ic}
         cexecute: action;
            receive msg[content,type,source] when
                (msg.type = MSAA_request ∧ msg.source = user_process);
            b1: broadcast( prepare, reply, {agree,refuse}, req# );
            {status = abort ∧ DONE(req#) ∧ value = req# ∧ NSA(req#) ∧
               NSC(req#) ∧ UIC ∧ TOC ∧
               (∀ j: j a site: [req#,reply(j).type,j] ∈ ρc[j] ∧
               ((reply(j).type = agree ∧ preparedj(req#)) ∨
               reply(j).type = refuse)}
            if ∀ j: j a site: reply(j).type = agree →
               status := commit;
               b2: broadcast( commit, reply, {commitack}, req# )
               {status = commit ∧ DONE(req#) ∧ value = req# ∧
                  NSC(req#+1) ∧ NSA(req#) ∧ UIC ∧ TOC ∧
                  (∀ j: j a site: [req#,agree,j] ∈ ρc[j] ∧ preparedj(req#) ∧
                     donej(req#) ∧ [req#,commitack,j] ∈ ρc[j])}
             [] ∃j : j a site : reply(j).type = refuse →
               b3: broadcast( abort, reply, {abortack}, req# )
            fi;
            reset: action,recovery
               {DONE(req#+1) ∧ (status = commit ∨ status = abort) ∧ UIC ∧
                   (value = req# ∨ value = req#+1) ∧ NSC(req#+1) ∧
                   NSA(req#+1) ∧ TOC}
               status := abort;
               value := req#+1; end;
            req# := value; {Ic}
        recovery; "from Figure 4.11"; end; {Ic}
      od;
      end loop ;
   end coord
```

Figure 4.10 -- Initialization and Action Portion of cexecute

The variable req# identifies the MSAA being executed by the coordinator, and the variable value identifies the MSAA with which the next message to be sent is associated. Also, the array reply(1..N) is assumed to be declared as follows:

```
var reply(1..N): record
                    content: integer;
                    type: character;
                    source: integer;
                  end.
```

If a failure occurs, one of the restartable FTAs -- coord, loop or reset -- could be restarted, or the recovery protocol for cexecute might be invoked. Figure 4.11 gives this recovery protocol.

The complete coordinator annotated with assertions can be found in Appendix 3.

The sequential annotations of $worker_j$ in Appendix 2 and the coordinator in Appendix 3 are replete proof outlines, if assignment to a variable is atomic. Using these annotations, it is easy to verify the correctness of each fault-tolerant action constituting the coordinator and worker processes using the inference rule presented in chapter 2. Here, we simply note that the coordinator variables status, value, req#, and ctoc must be in stable storage, as well as $done_j$, $prepared_j$, toc_j, name, and msg_name from $worker_j$. Each of these appears either in the precondition of a recovery protocol or in a non-local assertion.

4.5. Showing Satisfaction

For these sequential annotations to be valid proof outlines, satisfaction must be demonstrated. This means showing that Com_sat(c1,c2) and Com_Sat(c2,c1) hold for each pair of matching communicate operations c1 and

```
recovery
   {DONE(value) ∧ UIC ∧ TOC ∧
       ((status = commit ∧ NSA(value) ∧ NSC(value+1) ∧
            (∀ j: j a site: [value,agree,j] ∈ ρc[j] ∧ prepared_j(value))) ∨
        (status = abort ∧ NSA(value+1) ∧ NSC(value)))}
   req# :=  value;
   if status = abort →
      b4: broadcast( abort, reply, {abortack}, req# )
      {status = abort ∧ DONE(req#) ∧ value = req# ∧ UIC ∧ NSA(req#+1) ∧
         TOC ∧ NSC(req#) ∧ (∀ j: j a site: ¬done_j(req#) ∧
         [req#,abortack,j] ∈ ρc[j])}
    □ status = commit →
      b5: broadcast( commit, reply, {commitack}, req# )
      {status = commit ∧ DONE(req#) ∧ value = req# ∧ UIC ∧ NSA(req#) ∧ TOC ∧
         NSC(req#+1) ∧ (∀ j: j a site: [req#,agree,j] ∈ ρc[j] ∧
         prepared_j(req#) ∧ done_j(req#) ∧ [req#,commitack,j] ∈ ρc[j])}
   fi;
   reset: action,recovery
      "same as reset FTA from Figure 4.11"
      end;
   {Ic}
```

Figure 4.11 -- Recovery Protocol of Cexecute

c2, and that OWCom_Sat(c,r) hold for each matching communicate-**receive** pair. While this may appear to be a formidable task given the relatively large number of message-passing operations, it actually is not.

First, all workers are identical and each communicates only with the coordinator. Hence, it suffices to investigate only the satisfaction invariants obtained by considering the communication between the coordinator and one worker, say $worker_j$.

Secondly, nearly all of the satisfaction invariants reduce to true. For example, consider an arbitrary satisfaction invariant Com_Sat(cl,c2). If the Boolean in the **receives** in c2 cannot evaluate to true for any message text sent by cl, then the formula is trivially true because one of the conjuncts in the antecedent will be false. Moreover, if post(c2) follows directly from pre(c2) or the Boolean guard, then the consequent of Com_sat(cl,c2) is implied by the antecedent regardless of the state in which the formula is evaluated. Hence, the formula is a tautology. This means that the universal invariance of Com_Sat(cl,c2) is easily established except for the case when receipt by c2 of a message sent by cl allows assertions to be made about the state of the process in which cl appears. Thus, in this implementation, only four satisfaction invariants require further examination -- Com_Sat(c2,bl_j), Com_Sat(c3,$b2_j$), OWCom_Sat($b2_j$,r2), and Com_Sat(cl,$b3_j$).[2] In what follows, we will require some abbreviations. Given an assertion of the form

A: ($\forall$ j: j a site: P(j)),

then define A[j] to be: P(j).

Com_Sat(c2,bl_j) is defined to be:

---

[2]Although other satisfaction invariants cannot be reduced to true, each is identical to one of these four formulas.

$$(\text{pre}(c2)_{\overline{v}}^{\overline{id}} \wedge \text{mtext} = [\text{name},\text{agree},j]_{\overline{v}}^{\overline{id}} \wedge \text{status} = \text{abort} \wedge \text{DONE}(\text{req\#})[j] \wedge \text{value} = \text{req\#} \wedge \text{NSC}(\text{req\#})[j] \wedge \text{NSA}(\text{req\#})[j] \wedge \text{TOC}[j] \wedge \text{UIC}[j] \wedge \text{mtext.type} \in \{\text{agree},\text{refuse}\} \wedge \text{mtext.content} = \text{req\#} \wedge \text{mtext.source} = j \wedge \text{mtext.type} \neq \text{timeout} \wedge \text{mtext} \in (\sigma_c[j] \ominus \rho_c[j])) \Rightarrow$$

$$(\text{status} = \text{abort} \wedge \text{DONE}(\text{req\#})[j] \wedge \text{value} = \text{req\#} \wedge \text{NSA}(\text{req\#})[j] \wedge \text{NSC}(\text{req\#})[j] \wedge \text{TOC}[j] \wedge \text{UIC}[j] \wedge ((\text{reply}(j).\text{type} = \text{agree} \wedge \text{prepared}_j(\text{req\#})) \vee \text{reply}(j).\text{type} = \text{refuse}))^{\text{reply}(j),\, \rho_c[j]}_{\text{mtext},\rho_c[j] \oplus \text{mtext}}.$$

After simplification, this becomes

$$(\text{mtext} = [\text{req\#},\text{agree},j] \wedge \text{status} = \text{abort} \wedge \text{DONE}(\text{req\#})[j] \wedge \text{value} = \text{req\#} \wedge \text{NSC}(\text{req\#})[j] \wedge \text{NSA}(\text{req\#})[j] \wedge \text{TOC}[j] \wedge \text{UIC}[j] \wedge \text{mtext} \in (\sigma_c[j] \ominus \rho_c[j])) \Rightarrow \text{prepared}_j(\text{req\#}). \tag{4.12}$$

But notice that from the definition of the universal invariant UIj and the Network Axiom, we have that

$$\text{UIj} \Rightarrow ([\text{req\#},\text{agree},j] \notin \sigma_c[j] \vee [\text{req\#},\text{abort},j] \in \sigma_j \vee \text{prepared}_j(\text{req\#})).$$

But this implies

$$([\text{req\#},\text{agree},j] \notin (\sigma_c[j] \ominus \rho_c[j]) \vee \neg\text{NSA}(\text{req\#})[j] \vee \text{prepared}_j(\text{req\#})).$$

Since (4.12) follows from this, we can conclude that the universal invariance of UIj implies the universal invariance of Com_Sat(c2,b1_j).

The second satisfaction invariant, Com_Sat(c3,b2_j), is equivalent to

$$(\text{pre}(c3)_{\overline{v}}^{\overline{id}} \wedge \text{mtext} = [\text{name},\text{commitack},j]_{\overline{v}}^{\overline{id}} \wedge \text{status} = \text{commit} \wedge \text{DONE}(\text{req\#})[j] \wedge \text{value} = \text{req\#} \wedge \text{NSA}(\text{req\#})[j] \wedge \text{NSC}(\text{req\#}+1)[j] \wedge \text{TOC}[j] \wedge \text{UIC}[j] \wedge [\text{req\#},\text{agree},j] \in \rho_c[j] \wedge \text{prepared}_j(\text{req\#}) \wedge \text{mtext.type} \in \{\text{commitack}\} \wedge \text{mtext.content} = \text{req\#} \wedge \text{mtext.source} = j \wedge \text{mtext.type} \neq \text{timeout} \wedge \text{mtext} \in (\sigma_c[j] \ominus \rho_c[j])) \Rightarrow$$

$$(\text{status} = \text{commit} \wedge \text{DONE}(\text{req\#})[j] \wedge \text{value} = \text{req\#} \wedge \text{NSA}(\text{req\#})[j] \wedge \text{NSC}(\text{req\#}+1)[j] \wedge \text{TOC}[j] \wedge \text{UIC}[j] \wedge [\text{req\#},\text{agree},j] \in \rho_c[j] \wedge \text{prepared}_j(\text{req\#}) \wedge \text{done}_j(\text{req\#}))^{\text{reply}(j),\rho_c[j]}_{\text{mtext},\, \rho_c[j] \oplus \text{mtext}}$$

This becomes

$$(\text{mtext} = [\text{req\#},\text{commitack},j] \wedge \text{status} = \text{commit} \wedge \text{DONE}(\text{req\#})[j] \wedge \text{value} = \text{req\#} \wedge \text{NSA}(\text{req\#})[j] \wedge \text{NSC}(\text{req\#}+1)[j] \wedge \text{TOC}[j] \wedge \text{UIC}[j] \wedge [\text{req\#},\text{agree},j] \in \rho_c[j] \wedge \text{prepared}_j(\text{req\#}) \wedge \text{mtext} \in (\sigma_c[j] \ominus \rho_c[j])) \Rightarrow \text{done}_j(\text{req\#}) \tag{4.13}$$

But in a manner analogous to above, we have that

$$\begin{aligned} \text{UIj} &\Rightarrow ([\text{req\#},\text{commitack},j] \notin \sigma_c[j] \vee \text{done}_j(\text{req\#})) \\ &\Rightarrow ([\text{req\#},\text{commitack},j] \notin (\sigma_c[j] \ominus \rho_c[j]) \vee \text{done}_j(\text{req\#})) \\ &\Rightarrow (4.13). \end{aligned}$$

Hence, Com_Sat(c3,$b2_j$) is universally invariant.

OWCom_Sat($b2_j$,r2) is defined to be:

$$\begin{aligned} &(\text{pre}(b2_j)^{\overline{id}}_{\overline{v}} \wedge \text{mtext} = [\text{req\#},\text{commit},\text{cj}]^{\overline{id}}_{\overline{v}} \wedge \text{NDWj}(\text{msg\_name}) \wedge \\ &\quad \text{NSAAj}(\text{msg\_name}) \wedge \text{toc}_j > 0 \wedge \text{UIj} \wedge (\text{msg.content} = \text{msg\_name} \wedge \\ &\quad \text{msg.source} = \text{cj} \wedge \text{msg.type} \in \{\text{prepare},\text{commit},\text{abort}\})^{\text{msg}}_{\text{mtext}} \wedge \\ &\quad \text{mtext} \in (\sigma_j \ominus \rho_j)) \quad \Rightarrow \\ &(\text{NDWj}(\text{msg\_name}) \wedge \text{NSAAj}(\text{msg\_name}) \wedge \text{toc}_j > 0 \wedge \text{UIj} \wedge \\ &\quad (\text{msg.type} = \text{prepare} \vee \text{msg.type} = \text{abort} \vee \\ &\quad (\text{msg.type} = \text{commit} \wedge [\text{msg\_name},\text{abort},\text{cj}] \notin \sigma_j \wedge \\ &\quad [\text{msg\_name},\text{agree},j] \in \rho_c[j])))^{\text{msg},\rho_j}_{\text{mtext},\rho_j \oplus \text{mtext}}\}. \end{aligned}$$

This simplifies to

$$\begin{aligned} &(\text{mtext} = [\text{msg\_name},\text{commit},\text{cj}] \wedge \text{NDWj}(\text{msg\_name}) \wedge \\ &\quad \text{NSAAj}(\text{msg\_name}) \wedge \text{toc}_j > 0 \wedge \text{UIj} \wedge \text{mtext} \in (\sigma_j \ominus \rho_j)) \quad \Rightarrow \\ &([\text{msg\_name},\text{abort},\text{cj}] \notin \sigma_j \wedge [\text{msg\_name},\text{agree},j] \in \rho_c[j]). \end{aligned} \tag{4.14}$$

But

$$
\begin{aligned}
UIC \;&\Rightarrow\; ([msg\_name, commit, cj] \notin \sigma_j \;\vee \\
&\qquad ([msg\_name, agree, j] \in \rho_c[j] \;\wedge\; [msg\_name, abort, cj] \notin \sigma_j)) \\
&\Rightarrow\; ([msg\_name, commit, cj] \notin (\sigma_j \ominus \rho_j) \;\vee \\
&\qquad ([msg\_name, agree, j] \in \rho_c[j] \;\wedge\; [msg\_name, abort, cj] \notin \sigma_j)) \\
&\Rightarrow\; (4.14)
\end{aligned}
$$

Hence, $OWCom\_Sat(b2_j, r2)$ is universally invariant.

Finally, $Com\_Sat(c1, b3_j)$ is defined to be:

$$
\begin{aligned}
&(pre(c1)\tfrac{\overline{id}}{v} \wedge mtext = [name, abortack, j]\tfrac{\overline{id}}{v} \wedge status = abort \wedge \\
&\quad DONE(req\#)[j] \wedge value = req\# \wedge NSA(req\#+1)[j] \wedge NSC(req\#)[j] \wedge \\
&\quad TOC[j] \wedge UIC[j] \wedge mtext.type \in \{abortack\} \wedge mtext.content = req\# \wedge \\
&\quad mtext.source = j \wedge mtext.type \neq timeout \wedge mtext \in (\sigma_c[j] \ominus \rho_c[j])) \quad \Rightarrow \\
&(status = abort \wedge DONE(req\#)[j] \wedge value = req\# \wedge NSA(req\#+1)[j] \wedge \\
&\quad NSC(req\#)[j] \wedge TOC[j] \wedge UIC[j] \wedge \neg done_j(req\#))^{reply(j), \rho_c[j]}_{mtext,\; \rho_c[j] \oplus mtext}.
\end{aligned}
$$

This simplifies to

$$
\begin{aligned}
&(mtext = [req\#, abortack, j] \wedge status = abort \wedge DONE(req\#)[j] \wedge \\
&\quad value = req\# \wedge NSA(req\#+1)[j] \wedge NSC(req\#)[j] \wedge \\
&\quad TOC[j] \wedge UIC[j] \wedge mtext \in (\sigma_c[j] \ominus \rho_c[j])) \quad \Rightarrow \quad \neg done_j(req\#)
\end{aligned}
\tag{4.15}
$$

But

$$
\begin{aligned}
UIj \;&\Rightarrow\; ([req\#, abortack, j] \notin \sigma_c[j] \;\vee\; \neg done_j(req\#)) \\
&\Rightarrow\; ([req\#, abortack, j] \notin (\sigma_c[j] \ominus \rho_c[j]) \;\vee\; \neg done_j(req\#)) \\
&\Rightarrow\; (4.15)
\end{aligned}
$$

Hence, $Com\_Sat(c1, b3_j)$ is universally invariant.

4.6. Proving Non-Interference

To prove non-interference, it must be shown that the sequential annotation is not invalidated by the concurrent execution of any process. It is easy to show that execution of no worker invalidates the sequential annotation of another worker. The only variable shared between workers is σ_c, which is named in assertions of $worker_j$ only as part of the multiset partition $\sigma_c[j]$. Since $worker_j$ is the only worker process that modifies $\sigma_c[j]$, no interference is possible.

To show that execution of a worker does not interfere with the proof of the coordinator and vice versa is somewhat more difficult. We do this proof in two steps. First, we show that execution of no **send** statement transmitting a timeout message can invalidate any assertion in the other process. Then, we isolate those variables named in the annotation of one process that are modified by execution of (non-timeout transmitting) statements in the other, and show that no interference can result from this concurrent execution.

A **send** statement s in $worker_j$ that transmits a timeout message modifies the multiset σ_j. This multiset appears in the sequential annotation of the coordinator only as $\sigma_j[c]$ or in assertions of the form

$$[\text{msg\_name},\text{msg\_type},cj] \notin \sigma_j$$

where msg_type is never timeout. Execution of s can change neither the value of $\sigma_j[c]$ nor the truth of such an assertion, so no interference is possible. Similarly, a **send** statement in the coordinator that transmits a timeout message modifies the multiset σ_c. But since σ_c appears in the sequential annotation of $worker_j$ only as $\sigma_c[j]$, again no interference is possible. Thus, the transmission of timeouts in either process cannot invalidate assertions of the other.

$done_j$ and $prepared_j$ appear in assertions of the coordinator and are modified by $worker_j$, while the auxiliary variables σ_j and $\rho_c[j]$ are named in assertions of the $worker_j$ and changed by the coordinator. We now show that changes made to each of these variables cannot invalidate assertions in other processes.

First, consider $done_j$. The only statements in $worker_j$ that modify elements in this array are the two instances of

$$done_j(name) := true.$$

Since the precondition for this statement always is

$$\{P: NDWj(name+1) \wedge prepared_j(name) \wedge toc_j > 0 \wedge (msg\_name = name \vee msg\_name = name+1) \wedge [name, commit, cj] \in \rho_j \wedge NSAAj(name) \wedge UIj\},$$

we need only treat one occurrence.

$done_j$ appears in the sequential annotation of the coordinator only in assertions of the form

$$done_j(i) \wedge [i, commitack, j] \in \rho_c[j]$$

and

$$\neg done_j(i) \wedge [i, abortack, j] \in \rho_c[j],$$

for various MSAA names i. The former can never be invalidated by setting $done_j(name)$ to true, while the latter could be invalidated only if $i = name$. Hence, to conclude non-interference in this case it is sufficient to show

$$\begin{array}{ll} \{P \wedge \neg done_j(name) \wedge [name,abortack,j] \in \rho_c[j]\} & \\ \quad done_j(name) := true; & (4.16) \\ \{\neg done_j(name) \wedge [name,abortack,j] \in \rho_c[j]\}. & \end{array}$$

However,

$$\begin{array}{l} (P \wedge \neg done_j(name) \wedge [name,abortack,j] \in \rho_c[j]) \Rightarrow \\ (NSAAj(name) \wedge \neg done_j(name) \wedge [name,abortack,j] \in \rho_c[j]) \Rightarrow \\ ([name,abortack,j] \notin \sigma_c[j] \wedge \neg done_j(name) \wedge \\ \quad [name,abortack,j] \in \rho_c[j]) \Rightarrow \\ false \end{array}$$

due to the Network Axiom. Therefore, (4.16) follows trivially, and changes made in $worker_j$ to elements of $done_j$ do not invalidate the sequential annotation of the coordinator.

Now, consider whether either of the two statements in $worker_j$ that modify elements of $prepared_j$ can invalidate assertions in the sequential annotation of the coordinator. For

$$prepared_j(name) := true$$

the answer is obvious -- since no assertion contains the predicate $\neg prepared_j(name)$, no interference is possible. On the other hand, the statement

$$s\colon\ prepared_j(name) := false$$

is not so easily handled.

In the sequential annotation of the coordinator, the only predicates that can be affected by execution of s are $prepared_j(req\#)$ and $prepared_j(value)$. Moreover, this interference can occur only when $req\# = name$ or $value = name$. But notice that whenever $prepared_j(req\#)$ is asserted in the coordinator, so is NSA(req#). Similarly, when prepared_j(value) is asserted in the precondition of cexecute's recovery protocol, so is NSA(value). This means that for interference to occur, execution of s must invalidate

$$\{A\text{: NSA(name)} \wedge \text{prepared}_j\text{(name)}\}$$

Notice that

$$A \Rightarrow \quad (P\text{: [name,abort,cj]} \notin \sigma_j \wedge \text{prepared}_j\text{(name)}).$$

But,

$$\begin{aligned}&(\text{pre(s)} \wedge P) \Rightarrow \\ &(\text{[name,abort,cj]} \in \rho_j \wedge \text{[name,abort,cj]} \notin \sigma_j \wedge \text{prepared}_j\text{(name)}) \Rightarrow \\ &\text{false}\end{aligned}$$

due to the Network Axiom. That is, s cannot be executed when the coordinator is in a state satisfying P. Hence, it is impossible for execution of s to invalidate the sequential annotation of the coordinator.

The final step in proving non-interference is to show that modifications to the auxiliary variables σ_j and $\rho_c[j]$ in the coordinator do not invalidate the sequential annotation of worker_j. The only assertion for which this could possibly happen is post(r2):

$$\begin{aligned}&\{\text{NDWj(msg\_name)} \wedge \text{NSAAj(msg\_name)} \wedge \text{UIj} \wedge \text{toc}_j > 0 \wedge \\ &\quad (((\text{msg.type} = \text{prepare} \vee \text{msg.type} = \text{abort}) \wedge \\ &\qquad \text{[msg\_name,msg.type,cj]} \in \rho_j) \vee \\ &\quad (\text{msg.type} = \text{commit} \wedge \text{[msg\_name,commit,cj]} \in \rho_j \wedge \\ &\quad \text{[msg\_name,abort,cj]} \notin \sigma_j \wedge \text{[msg\_name,agree,j]} \in \rho_c[j]))\}.\end{aligned}$$

However,

$$\text{[msg\_name,agree,j]} \in \rho_c[j]$$

cannot be invalidated since it is monotonic. Unfortunately,

$$[\text{msg\_name,abort,cj}] \notin \sigma_j$$

is not -- execution of either broadcast operation b3 or b4 could make it false. Hence, both of the following must be shown:

$$\{\text{pre(b3)} \wedge \text{post(r2)}\}\ \text{b3}\ \{\text{post(r2)}\}$$

$$\{\text{pre(b4)} \wedge \text{post(r2)}\}\ \text{b4}\ \{\text{post(r2)}\}.$$

For notational convenience, define two predicates P and P':

$$\begin{aligned} P \equiv {} & ((\text{msg.type} = \text{prepare} \vee \text{msg.type} = \text{abort}) \wedge \\ & [\text{msg\_name,msg.type,cj}] \in \rho_j) \end{aligned}$$

$$\begin{aligned} P' \equiv {} & (\text{msg.type} = \text{commit} \wedge [\text{msg\_name,commit,c}] \in \rho_j \wedge \\ & [\text{msg\_name,abort,cj}] \notin \sigma_j \wedge [\text{msg\_name,agree,j}] \in \rho_c[j]). \end{aligned}$$

Then, we have that

$$\begin{aligned} & (\text{pre(b3)} \wedge \text{post(r2)}) \Rightarrow \\ & (\text{NSC(req\#)} \wedge \text{NDWj(msg\_name)} \wedge \text{NSAAj(msg\_name)} \wedge \text{toc}_j > 0 \wedge \text{UIj} \wedge \\ & \quad (P \vee P')) \Rightarrow \\ & (\text{NDWj(msg\_name)} \wedge \text{NSAAj(msg\_name)} \wedge \text{UIj} \wedge \text{toc}_j > 0 \wedge \\ & \quad ((P \wedge \text{NSC(req\#)}) \vee (P' \wedge \text{NSC(req\#)}))). \end{aligned}$$

Since

$$\text{NSC(req\#)} \Rightarrow (\forall k\text{: } k \geq \text{req\#: } [\text{k,commit,cj}] \notin \sigma_j),$$

it follows from the Network Axiom that

$$(P' \wedge \text{NSC(req\#)}) \Rightarrow \text{msg\_name} < \text{req\#}.$$

Thus,

$$\begin{aligned} & (\text{pre(b3)} \wedge \text{post(r2)}) \Rightarrow \\ & (\text{Q: NSAAj(msg\_name)} \wedge \text{NDWj(msg\_name)} \wedge \text{UIj} \wedge \text{toc}_j > 0 \wedge \\ & \quad ((P \wedge \text{NSC(req\#)}) \vee (P' \wedge \text{NSC(req\#)} \wedge \text{msg\_name} < \text{req\#}))). \end{aligned}$$

But since b3 sends a message associated with MSAA req# to worker_j, its execution cannot invalidate Q. From this and the observation that

$$Q \Rightarrow \text{post}(r2),$$

it follows that

$$\{\text{pre}(b3) \wedge \text{post}(r2)\}\ b3\ \{\text{post}(r2)\}.$$

In a similar fashion, it can be shown that

$$\{\text{pre}(b4) \wedge \text{post}(r2)\}\ b4\ \{\text{post}(r2)\}.$$

Thus, the changes made to σ_j and $\rho_c[j]$ by the coordinator do not interfere with the sequential annotation of worker_j.

4.7. Remaining Proof Obligations

To conclude that the sequential annotation of each process in this implementation of the two-phase commit protocol constitutes a valid proof, the five proof obligations incurred in sections 4.3 and 4.4 must also be shown.

To prove PO_1, it is sufficient to show that the precondition of each communicate operation remains true across execution of **receive** statements in its rectimeout operations. In the coordinator, such an execution modifies the variable reply(j) for some j, and ρ_c. In worker_j, execution of a **receive** statement modifies msg and ρ_j. However, reply(j) never appears in the precondition of any communicate statement in the coordinator, and ρ_c appears only in assertions of the form

$$\text{mtext} \in \rho_c[j]$$

for some message text mtext. Hence, the precondition of any communicate in the coordinator remains true upon termination of its constituent rectimeout operations. Similarly, msg never appears in the precondition of any communicate in worker_j and ρ_j appears only in assertions of the form

$$\text{mtext} \in \rho_j$$

for some message text mtext. Thus, the precondition of each communicate in worker_j is invariant across execution of the rectimeout operations within it.

To establish PO_2, it is sufficient to show that the execution of a **send** statement transmitting a timeout message does not invalidate either the precondition or the postcondition of the communicate operation containing it. This follows trivially in both the coordinator and the worker. In the coordinator, execution of such a **send** statement modifies only the multiset σ_c. Since this never appears in the precondition or postcondition of any communicate operation, no interference is possible. Likewise, execution of a timeout-transmitting **send** statement in worker_j modifies only σ_j, which does not appear in the precondition or postcondition of any communicate operations.

To show PO_3, we must prove that no message sent by a communicate operation to a process other than itself has type timeout. This follows immediately from examination of the code.

Proof obligation PO_4 requires showing that execution of no communicate operation within a broadcast invalidates the precondition or postcondition of any other communicate in the same broadcast. The execution of communicate B_j in any broadcast operation modifies toc_j, σ_j, $\rho_c[j]$, and σ_c. But none of these appear in the precondition or postcondition of any communicate within the operation except B_j. Hence, no interference is possible.

4.8. Generalizations

In this protocol, execution of an MSAA at a worker may be interrupted by failures several times before being completed. Thus, both the preparation and operation might be executed multiple times. Consequently, both the operation and its preparation must be *restartable*. A sequence of instructions is restartable if each intermediate assertion implies the precondition of the sequence. For example, both the preparation and operation used above are restartable -- assigning a constant to a variable.

In the example presented above, no information specific to the particular MSAA being executed is included in the messages. Clearly, for more complex operations, additional information might be required. The sequence of messages, however, would not change, and so the proof remains valid.

4.9. Implementation Considerations

Since no assumption has been made about the relative execution speeds of processes, a **receive** statement within a communicate operation can timeout any number of times. This would result in a large number of messages being injected into the network. Since there is no guarantee that all of these messages will be received by the destination process, extraneous messages might accumulate at the site executing the destination process. Clearly, there must be a way to dispose of such extra messages.

Fortunately, this is easily done. Notice that in both the coordinator and worker processes, messages are received (as opposed to delivered) in non-decreasing order based on the name of the MSAA with which they are associated. This name is in the content field of the record. Thus, once a message m is received, all messages m' such that m'.content < m.content

can be discarded, since no attempt will subsequently be made to receive them. Notice also that the number of extra messages can be minimized by a judicious choice of the timeout interval.

4.10. Related Work

In [Skeen 1981], a variety of recovery protocols are considered in the context of distributed transaction management. There, protocols are modeled as finite state automata, and their characteristics described in terms of those automatons. Independent recovery protocols -- recovery protocols that do not communicate with other processes -- are developed to allow distributed database systems to withstand a single site failure. In addition, it is shown that no independent recovery protocols exist if two sites fail, or if a network partition results in lost messages.

A proof of the two-phase commit protocol appears in [Baer *et al* 1981]. The protocol is modeled as a colored Petri net, and linear invariants are used to show the the net reaches a desired final state. However, the paper is concerned mainly with presenting the model, and the protocol is proved only for the case when no failures occur.

Chapter 5

Conclusions

5.1. In Retrospect

In this dissertation, we have developed proof rules for fault-tolerant actions and asynchronous communications operations. Two benefits accrue from doing this. One is that programs using these statements can now be formally verified. A second, and perhaps more important, benefit is the understanding of the statements' semantics gained in the course of deriving these rules. Such an understanding facilitates the development of correct programs, even if formal verification is not performed. Moreover, the insight also provides a basis for examining some of the guidelines for the safe use of these statements that have evolved over the years.

In developing the proof requirements for fault-tolerant actions, some of the fundamental problems involved in writing correct recovery protocols were isolated:

(1) A recovery protocol must be correct when started in any intermediate state of the computation.

(2) It must operate using only partial state information.

The inference rule states precisely what conditions must hold if a recovery protocol is to correctly complete the state transformation in progress when a failure occurs.

It is interesting to note the relationship between these proof requirements and the informal guidelines that appear in the folklore. For example, the fact that information must be stored in some type of a stable storage is not new [Lampson & Sturgis 1978] [Gray 1978]. Such information is called a <u>checkpoint</u> [Denning 1976]. With no rule specifying exactly what values must be saved, it is a common practice to save the entire state of a process at frequent intervals when failures are expected. This allows a process to be rolled back and restarted at some earlier, well-defined point after a failure. However, use of our inference rule makes such extensive checkpointing unnecessary, since determination of exactly those variables whose values must be saved is now possible.

A methodology for designing fault-tolerant computing systems that uses our verification techniques has been developed. This methodology involves first developing correct programs for fail-stop processors, and then approximating these processors using hardware and software redundancy. Program correctness follows from the use of fault-tolerant actions and their proof rules, while real-time response constraints are satisfied by using multiple processors and reconfiguration.

The use of fail-stop processors as our underlying computational model, hence the foundation of our methodology, followed from our desire to use a partial correctness programming logic. In a fail-stop processor all failures are detected, and no incorrect state transformations result from failures. Thus, if execution of a statement terminates, by definition the transformation specified by that statement has occurred -- the effect of execution is consistent with the programming logic. Failure, by definition, prevents statements from terminating. Thus, the partial correctness (as opposed to total correctness) nature of the programming logic subsumes the consequences of failures.

Deriving proof rules for asynchronous message-passing statements has also yielded some interesting insights. As was the case with recovery protocols, a key impediment to writing correct programs that use asynchronous sends has long been known: any information obtained from a message reflects a past state of the sending process, not its current state. We recognized this problem as a manifestation of interference [Owicki & Gries 1976]. It is permissible to make any conclusion about the system state after receipt of a message provided the satisfaction invariants associated with that **receive** statement have been proved universally invariant. If this is not the case, then some process has interfered with a satisfaction invariant, rendering the conclusion invalid.

The notion of interference has allowed us to explain formally ad hoc guidelines about the use of message-passing. For example, a large number of programming languages have been proposed with mechanisms that enforce disciplined use of message-passing: Communicating Sequential Processes [Hoare 1978], Ada [Ichbiah 1979], MESA [Lampson 1981], Synchronizing Resources [Andrews 1981], and E-CLU [Liskov 1979]. Implicit in these proposals is the notion that structured message-passing operations make programs easier to understand. In this dissertation, we have formally justified this belief by showing how disciplined use of message-passing can control interference. Protocols such as the "send message, receive acknowledgment" protocol do this by limiting the actions a sending process takes prior to receipt of the message.

The use of the techniques developed in this dissertation has been illustrated by two examples: proving the partial correctness of a two-phase commit protocol and designing a prototype fault-tolerant process control system. Previous attempts at demonstrating the correctness of the two-phase commit protocol have been based on translating the protocol to some other computational model, such as Petri nets [Baer et al. 1981], or finite state automata [Skeen 1981]. Our proof is developed directly from the code itself; there is no possibility of introducing errors in the translation process.

Developing a prototype fault-tolerant process control system illustrated a useful technique for making program loops fault-tolerant. Associated with every program loop is a loop invariant -- an assertion that is true at the beginning and end of each execution of the loop body. This loop invariant can be made into a universal invariant by requiring that all changes to variables in the loop invariant be performed in a single atomic action. Then, by storing the variables named in the loop invariant in stable storage, the loop body serves as its own restart protocol.

5.2. Topics for Further Research

In this dissertation, we have mainly been concerned with the partial correctness of programs. Clearly, the issue of termination for the types of programs examined herein is one area deserving of further study.

For a program consisting of a collection of interacting processes using asynchronous message-passing to terminate, no **receive** statement can block forever waiting for a message. This means that a message satisfying the Boolean guard on each **receive** statement executed must eventually be delivered to the appropriate site. Proving that this will be true appears to be non-trivial, especially if messages can be lost.

Note that even using **receive** statements that time out might not ensure termination of the program. For example, if a **receive** statement that can time out is embedded in a loop that iterates until a particular type of message (say an acknowledgment) is received, the problem becomes one of showing that the loop will terminate. Unfortunately, standard techniques for proving termination -- the use of variant functions or well-founded sets [Dijkstra 1976] -- are not applicable. Hence, to prove termination the same argument is required as when no timeouts were used -- that the appropriate message will be delivered eventually.

Another problem associated with proving termination of programs that use asynchronous message-passing concerns the **send** statement. We have assumed that execution of a **send** always terminates. This is equivalent to assuming that the communications system has an infinite buffering capacity. In reality, this assumption is unjustified. Hence, to show that a program executing on a system with a finite buffering capacity terminates, one also has to prove that execution of no **send** statement will block forever -- that is, that the buffer space or slack bound [Martin 1980] in the communications system is sufficient.

In chapter 2, we briefly touched on the problem of proving termination for programs constructed from fault-tolerant actions. There we argued that a program would terminate if occasionally there was enough time between failures. Clearly, more formal techniques are required in this area if more precise results are to be obtained. Since this problem seems inherently probabilistic, one solution might be to extend real-time temporal logic [Harter & Bernstein 1981] to encompass probabilistic behavior.

Lastly, the problem of approximating systems that have predictable failure-mode operating characteristics should be more fully addressed. We outlined how to implement an approximation to fail-stop processors and stable storage in order to demonstrate the feasibility of using such a model. A less costly approach would undoubtedly prove beneficial.

Appendix 1

Sequential Annotation of Communicate Operation
and
Derivation of Satisfaction Rule

1. Sequential Annotation

$\{\sigma_D = \sigma_0 \wedge \rho_S = \rho_0 \wedge \sigma_S = \sigma'_0 \wedge toc = t_0 \wedge T\}$

communicate(cont, sendt, S, msg, ncont, ntype, D, toc):

$\{\sigma_D = \sigma_0 \wedge \rho_S = \rho_0 \wedge \sigma_S = \sigma'_0 \wedge toc = t_0 \wedge sendt \neq timeout \wedge T\}$

k:= 1;

$\{\sigma_D = \sigma_0 \wedge \rho_S = \rho_0 \wedge \sigma_S = \sigma'_0 \wedge toc = t_0 \wedge sendt \neq timeout \wedge T \wedge k = 1\}$

s1: **send** [cont,sendt,S] **to** D;

$\{\sigma_D = \sigma_0 \oplus [cont,sendt,S] \wedge \rho_S = \rho_0 \wedge \sigma_S = \sigma'_0 \wedge toc = t_0 \wedge sendt \neq timeout \wedge T \wedge k = 1\}$

$\{(\sigma_0 \oplus [cont,sendt,S]) \subseteq \sigma_D \wedge \rho_0 \subseteq \rho_S \wedge \sigma'_0 \subseteq \sigma_S \wedge toc = t_0 \wedge sendt \neq timeout \wedge T \wedge k = 1\}$

toc:= toc+1;

$\{(\sigma_0 \oplus [cont,sendt,S]) \subseteq \sigma_D \wedge \rho_0 \subseteq \rho_S \wedge \sigma'_0 \subseteq \sigma_S \wedge toc = t_0+k \wedge sendt \neq timeout \wedge T\}$

cobegin;

$\{(\sigma_0 \oplus [cont,sendt,S]) \subseteq \sigma_D \wedge \rho_0 \subseteq \rho_S \wedge toc = t_0+k \wedge sendt \neq timeout \wedge T\}$

r1: **receive** msg[content,type,source] **when**

((msg.type $\in$ ntype $\wedge$ msg.content = ncont $\wedge$ msg.type $\neq$ timeout $\wedge$ msg.source = D) $\vee$ (msg.content = toc $\wedge$ msg.type = timeout));

$\{(\sigma_0 \oplus [cont,sendt,S]) \subseteq \sigma_D \wedge (\rho_0 \oplus msg) \subseteq \rho_S \wedge toc = t_0+k \wedge sendt \neq timeout \wedge ((msg.type \in ntype \wedge msg.content = ncont \wedge msg.type \neq timeout \wedge Q) \vee (msg.type = timeout \wedge T))\}$

// $\{\sigma'_0 \subseteq \sigma_S \wedge toc = t_0+k\}$

delay(t);

send [toc,timeout,S] **to** S;

$\{(\sigma'_0 \oplus [t_0+k,timeout,S]) \subseteq \sigma_S \wedge toc = t_0+k\}$

coend;

$\{I:(\sigma_0 \oplus [cont,sendt,S]) \subseteq \sigma_D \wedge (\rho_0 \oplus msg) \subseteq \rho_S \wedge (\sigma'_0 \oplus [t_0+k,timeout,S]) \subseteq \sigma_S \wedge toc = t_0+k \wedge sendt \neq timeout \wedge ((msg.type \in ntype \wedge msg.content = ncont \wedge msg.type \neq timeout \wedge Q) \vee (msg.type = timeout \wedge T))\}$

```
do msg.type = timeout →     {I ∧ msg.type = timeout}
    {(σ_0 ⊕ [cont,sendt,S]) ⊆ σ_D ∧ (ρ_0 ⊕ msg) ⊆ ρ_S ∧
      (σ'_0 ⊕ [t_0+k,timeout,S]) ⊆ σ_S ∧ toc = t_0+k ∧
      sendt ≠ timeout ∧ msg.type = timeout ∧ T}
    k:= k+1;
    {(σ_0 ⊕ [cont,sendt,S]) ⊆ σ_D ∧ ρ_0 ⊆ ρ_S ∧ σ'_0 ⊆ σ_S ∧ toc = t_0+k-1 ∧
      sendt ≠ timeout ∧ T}
    s2: send [cont,sendt,S] to D;
    {(σ_0 ⊕ [cont,sendt,S]) ⊆ σ_D ∧ ρ_0 ⊆ ρ_S ∧ σ'_0 ⊆ σ_S ∧
      toc = t_0+k-1 ∧ sendt ≠ timeout ∧ T}
    toc:= toc+1;
    {(σ_0 ⊕ [cont,sendt,S]) ⊆ σ_D ∧ ρ_0 ⊆ ρ_S ∧ σ'_0 ⊆ σ_S ∧
      toc = t_0+k ∧ sendt ≠ timeout ∧ T}
    cobegin;
       {(σ_0 ⊕ [cont,sendt,S]) ⊆ σ_D ∧ ρ_0 ⊆ ρ_S ∧ toc = t_0+k ∧
           sendt ≠ timeout ∧ T}
       r2: receive msg[content,type,source] when
               ((msg.type ∈ ntype ∧ msg.content = ncont ∧
                  msg.type ≠ timeout ∧ msg.source = D) ∨ (msg.content = toc ∧
                  msg.type = timeout));
       {(σ_0 ⊕ [cont,sendt,S]) ⊆ σ_D ∧ (ρ_0 ⊕ msg) ⊆ ρ_S ∧ toc = t_0+k ∧
           sendt ≠ timeout ∧ ((msg.type ∈ ntype ∧ msg.content = ncont ∧
           msg.type ≠ timeout ∧ Q) ∨ (msg.type = timeout ∧ T))}
    // {σ'_0 ⊆ σ_S ∧ toc = t_0+k}
       delay(t);
       send [toc,timeout,S] to S;
       {(σ'_0 ⊕ [t_0+k,timeout,S]) ⊆ σ_S ∧ toc = t_0+k}
    coend;
    {I}
od
end
{(σ_0 ⊕ [cont,sendt,S]) ⊆ σ_D ∧ (ρ_0 ⊕ msg) ⊆ ρ_S ∧ (σ'_0 ⊕ [t_0+k, timeout,S]) ⊆ σ_S ∧
   toc = t_0+k ∧ msg.type ∈ ntype ∧ msg.content = ncont ∧ Q}
```

2. Derivation of Satisfaction Rule

Let c1 in process S and c2 in D be two matching communicate operations. Furthermore, suppose that theorems of the following form have been proved:

$\{\sigma_D = \sigma1_0 \wedge \rho_S = \rho1_0 \wedge \sigma_S = \sigma1'_0 \wedge toc1 = t1_0 \wedge T1\}$

c1: communicate(cont1,st1,S,msg1,ncnt1,nt1,D,toc1)

$\{(\sigma1_0 \oplus [cont1,st1,S]) \subseteq \sigma_D \wedge (\rho1_0 \oplus msg1) \subseteq \rho_S \wedge$
$(\sigma1'_0 \oplus [t1_0+k1,timeout,S]) \subseteq \sigma_S \wedge toc1 = t1_0+k1 \wedge msg1.type \in nt1 \wedge$
$msg1.content = ncnt1 \wedge Q1\}$

and

$\{\sigma_S = \sigma2_0 \wedge \rho_D = \rho2_0 \wedge \sigma_D = \sigma2'_0 \wedge toc2 = t2_0 \wedge T2\}$

c2: communicate(cont2,st2,D,msg2,ncnt2,nt2,S,toc2)

$\{(\sigma2_0 \oplus [cont2,st2,D]) \subseteq \sigma_S \wedge (\rho2_0 \oplus msg2) \subseteq \rho_D \wedge$
$(\sigma2'_0 \oplus [t2_0+k2,timeout,D]) \subseteq \sigma_D \wedge toc2 = t2_0+k2 \wedge msg2.type \in nt2 \wedge$
$msg2.content = ncnt2 \wedge Q2\}$.

Let s1 be the first send statement in c1 transmitting to process D and s2 the second; similarly, let r1 be the first receive in c2 and r2 the second. Then, for post(c2) to be correct if a message sent by c1 is received, $\mathrm{Satisfaction}_{Asynch}$(s1,r1), $\mathrm{Satisfaction}_{Asynch}$(s1,r2), $\mathrm{Satisfaction}_{Asynch}$(s2,r1), and $\mathrm{Satisfaction}_{Asynch}$(s2,r2) must be universally invariant.

Since the value of a message sent by c1 to D does not depend on the values of auxiliary variables σ_D, ρ_S, σ_S, and k, or toc1, we have that $\mathrm{Satisfaction}_{Asynch}$(s1,r1) is:

$$((\mathrm{st1} \neq \mathrm{timeout} \wedge \mathrm{T1})^{\overline{\mathrm{id}}}_{\overline{\mathrm{v}}} \wedge \mathrm{mtext} = [\mathrm{cont1},\mathrm{st1},\mathrm{S}]^{\overline{\mathrm{id}}}_{\overline{\mathrm{v}}} \wedge$$
$$(\sigma 2_0 \oplus [\mathrm{cont2},\mathrm{st2},\mathrm{D}]) \subseteq \sigma_S \wedge \rho 2_0 \subseteq \rho_D \wedge \mathrm{toc2} = \mathrm{t2}_0 + \mathrm{k2} \wedge$$
$$\mathrm{st2} \neq \mathrm{timeout} \wedge \mathrm{T2} \wedge ((\mathrm{msg2.type} \in \mathrm{nt2} \wedge \mathrm{msg2.content} = \mathrm{ncnt2} \wedge$$
$$\mathrm{msg2.type} \neq \mathrm{timeout} \wedge \mathrm{msg2.source} = \mathrm{S}) \vee (\mathrm{msg2.content} = \mathrm{toc2} \wedge$$
$$\mathrm{msg2.type} = \mathrm{timeout} \wedge \mathrm{msg2.source} = \mathrm{D}))^{\mathrm{msg2}}_{\mathrm{mtext}} \wedge$$
$$\mathrm{mtext} \in (\sigma_D \ominus \rho_D)) \quad \Rightarrow$$
$$((\sigma 2_0 \oplus [\mathrm{cont2},\mathrm{st2},\mathrm{D}]) \subseteq \sigma_S \wedge \mathrm{toc2} = \mathrm{t2}_0 + \mathrm{k2} \wedge \mathrm{st2} \neq \mathrm{timeout} \wedge$$
$$((\mathrm{msg2.type} \in \mathrm{nt2} \wedge \mathrm{msg2.content} = \mathrm{ncnt2} \wedge \mathrm{msg2.type} \neq \mathrm{timeout} \wedge \mathrm{Q2}) \vee$$
$$(\mathrm{msg2.type} = \mathrm{timeout} \wedge \mathrm{T2})))^{\mathrm{msg2},\, \rho_D}_{\mathrm{mtext},\, \rho_D \oplus \mathrm{mtext}}.$$

After simplification, this becomes

$$\mathrm{S(s1,r1)}: ((\mathrm{st1} \neq \mathrm{timeout} \wedge \mathrm{T1})^{\overline{\mathrm{id}}}_{\overline{\mathrm{v}}} \wedge \mathrm{mtext} = [\mathrm{cont1},\mathrm{st1},\mathrm{S}]^{\overline{\mathrm{id}}}_{\overline{\mathrm{v}}} \wedge$$
$$(\sigma 2_0 \oplus [\mathrm{cont2},\mathrm{st2},\mathrm{D}]) \subseteq \sigma_S \wedge \rho 2_0 \subseteq \rho_D \wedge \mathrm{toc2} = \mathrm{t2}_0 + \mathrm{k2} \wedge$$
$$\mathrm{st2} \neq \mathrm{timeout} \wedge \mathrm{T2} \wedge \mathrm{mtext.type} \in \mathrm{nt2} \wedge \mathrm{mtext.content} = \mathrm{ncnt2} \wedge$$
$$\mathrm{mtext.type} \neq \mathrm{timeout} \wedge \mathrm{mtext.source} = \mathrm{S} \wedge$$
$$\mathrm{mtext} \in (\sigma_D \ominus \rho_D)) \quad \Rightarrow \quad \mathrm{Q2}^{\mathrm{msg2},\, \rho_D}_{\mathrm{mtext},\, \rho_D \oplus \mathrm{mtext}}.$$

$\mathrm{Satisfaction}_{\mathrm{Asynch}}$(s1,r2) is the same as S(s1,r1), since pre(r1) is identical to pre(r2), and post(r1) is identical to post(r2).

Moreover, $\mathrm{Satisfaction}_{\mathrm{Asynch}}$(s2,r1) and $\mathrm{Satisfaction}_{\mathrm{Asynch}}$(s2,r2) are also identical to S(s1,r1). This is because the only difference in satisfaction invariants involving **receive** statement r and two send statements s and s' that evaluate the same expression when computing a message is that one has $\mathrm{pre(s)}^{\overline{\mathrm{id}}}_{\overline{\mathrm{v}}}$ in the antecedent of the implication, while the other has $\mathrm{pre(s')}^{\overline{\mathrm{id}}}_{\overline{\mathrm{v}}}$. However, any part of pre(s) (or pre(s')) that cannot influence the value of the message text can be deleted from these formulas. In the case of these communicate operations, the multisets and toc2 cannot appear in the expression used to compute the text of a message sent to D. Thus, the portion of the precondition used in determining the value of a message will be identical in both pre(s1) and pre(s2). Therefore, for post(c2) to be correct when a message sent by either s1 or s2 is received, it is sufficient to prove

$$\begin{aligned}
&\text{Com_Sat(c1,c2)}: ((\text{st1} \neq \text{timeout} \wedge \text{T1})^{\overline{\text{id}}}_{\overline{\text{v}}} \wedge \text{mtext} = [\text{cont1,st1,S}]^{\overline{\text{id}}}_{\overline{\text{v}}} \wedge \\
&\quad \text{st2} \neq \text{timeout} \wedge \text{T2} \wedge \text{mtext.type} \in \text{nt2} \wedge \text{mtext.content} = \text{ncnt2} \wedge \\
&\quad \text{mtext.type} \neq \text{timeout} \wedge \text{mtext.source} = \text{S} \wedge \\
&\quad \text{mtext} \in (\sigma_D \ominus \rho_D)) \quad \Rightarrow \quad \text{Q2}^{\text{msg2},\, \rho_D}_{\text{mtext},\, \rho_D \oplus \text{mtext}}
\end{aligned}$$

universally invariant.

Appendix 2

Sequential Annotation of Worker_j

$NDWj(i) \equiv (\forall h\colon h \geq i\colon \neg done_j(h))$

$NSAAj(i) \equiv (\forall h\colon h \geq i\colon [h,abortack,j] \notin \sigma_c[j])$

$UIj \equiv (\forall k\colon k \geq 1\colon$
$\quad ([k,agree,j] \notin \sigma_c[j] \vee [k,abort,cj] \in \rho_j \vee prepared_j(k)) \wedge$
$\quad ([k,commit,cj] \in \rho_j \vee \neg done_j(k)) \wedge$
$\quad ([k,commitack,j] \notin \sigma_c[j] \vee done_j(k)) \wedge$
$\quad ([k,abortack,j] \notin \sigma_c[j] \vee \neg done_j(k)))$

$Ij \equiv (NDWj(msg\_name) \wedge msg\_name = name \wedge NSAAj(msg\_name) \wedge toc_j > 0 \wedge UIj \wedge$
$\quad (msg.type = prepare \vee msg.type = abort))$

```
worker_j: action, recovery;
   var msg: record
             content: integer;
             type: character;
             source: integer;
           end;
       name, msg_name, toc_j: integer;
   {ρ_j = ∅ ∧ σ_c[j] = ∅ ∧ NDWj(1) ∧ NSAAj(1) ∧ UIj}
   toc_j, name:= 1, 1;
   {ρ_j = ∅ ∧ σ_c[j] = ∅ ∧ NDWj(1) ∧ name = 1 ∧ toc_j > 0 ∧ UIj}
   loop: action, recovery;
      {NDWj(name) ∧ NSAAj(name) ∧ toc_j > 0 ∧ UIj}
      r1: receive msg[content,type,source] when
                     (msg.content = name ∧ msg.source = cj ∧
                     (msg.type = prepare ∨ msg.type = abort));
      {NDWj(name) ∧ NSAAj(name) ∧ toc_j > 0 ∧ UIj ∧
                     (msg.type = prepare ∨ msg.type = abort)}
      msg_name:= name;
      {Ij}
      do true →  {Ij}
```

```
wexecute: action; {Ij}
  if msg.type = abort →
    {msg.type = abort ∧ NDWj(msg_name) ∧ msg_name = name ∧
        NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
    aa: action,recovery;
      {NDWj(name) ∧ (msg_name = name ∨ msg_name = name+1) ∧
        NSAAj(name+1) ∧ toc_j > 0 ∧ UIj}
      c1: communicate(name,abortack,j,msg,name+1,{prepare,abort},
                      cj,toc_j);
      {NDWj(name) ∧ (msg_name = name ∨ msg_name = name+1) ∧
        NSAAj(name+1) ∧ toc_j > 0 ∧ UIj ∧ msg.content = name+1 ∧
        (msg.type = prepare ∨ msg.type = abort)}
      msg_name :=  name+1;
      {NDWj(name) ∧ msg_name = name+1 ∧ NSAAj(name+1) ∧
        toc_j > 0 ∧ UIj ∧ (msg.type = prepare ∨
        msg.type = abort)}
      end;
    {NDWj(name) ∧ msg_name = name+1 ∧ NSAAj(msg_name) ∧ toc_j > 0 ∧
      UIj ∧ (msg.type = prepare ∨ msg.type = abort)}
    name :=  msg_name;
    {Ij}
  □ msg.type = prepare ∧ possible_j(name) →
    {msg.type = prepare ∧ NDWj(msg_name) ∧ msg_name = name ∧
        possible_j(name) ∧ NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
    prepared_j(name) :=  true;
    {NDWj(msg_name) ∧ msg_name = name ∧
      prepared_j(name) ∧ NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
    ag: action,recovery;
      {NDWj(msg_name) ∧ msg_name = name ∧ prepared_j(name) ∧
        NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
      c2: communicate(name,agree,j,msg,name,{commit,abort},
                      cj,toc_j);
      {NDWj(msg_name) ∧ msg_name = name ∧ prepared_j(name) ∧
        msg ∈ ρ_j ∧ msg.content = name ∧ msg.source = cj ∧
        (msg.type = commit ∨ msg.type = abort) ∧
        NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
      end;
```

```
{NDWj(msg_name) ∧ msg_name = name ∧ prepared_j(name) ∧
   msg ∈ ρ_j ∧ msg.content = name ∧ msg.source = cj ∧
   (msg.type = commit ∨ msg.type = abort) ∧
   NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
if msg.type = commit →
   {NDWj(msg_name) ∧ msg_name = name ∧ prepared_j(name) ∧
      [name,commit,cj] ∈ ρ_j ∧ NSAAj(msg_name) ∧ toc_j > 0 ∧
      UIj}
   ca: action,recovery;
      {NDWj(name+1) ∧ prepared_j(name) ∧ (msg_name = name ∨
         msg_name = name+1) ∧ [name,commit,cj] ∈ ρ_j ∧
         NSAAj(name) ∧ toc_j > 0 ∧ UIj}
      done_j(name) :=  true;
      {NDWj(name+1) ∧ prepared_j(name) ∧ (msg_name = name ∨
         msg_name = name+1) ∧ [name,commit,cj] ∈ ρ_j ∧
         done_j(name) ∧ NSAAj(name) ∧ toc_j > 0 ∧ UIj}
      c3: communicate(name,commitack,j,msg,
                        name+1,{prepare,abort},cj,toc_j);
      {NDWj(name+1) ∧ prepared_j(name) ∧ (msg_name = name ∨
         msg_name = name+1) ∧ [name,commit,cj] ∈ ρ_j ∧
         done_j(name) ∧ (msg.type = prepare ∨
         msg.type = abort) ∧ NSAAj(name) ∧ toc_j > 0 ∧ UIj}
      msg_name :=  name+1;
      {NDWj(name+1) ∧ prepared_j(name) ∧ msg_name = name+1 ∧
         [name,commit,cj] ∈ ρ_j ∧ done_j(name) ∧
         (msg.type = prepare ∨ msg.type = abort) ∧
         NSAAj(name) ∧ toc_j > 0 ∧ UIj}
      end;
   {NDWj(msg_name) ∧ prepared_j(name) ∧ msg_name = name+1 ∧
      [name,commit,cj] ∈ ρ_j ∧ done_j(name) ∧
      (msg.type = prepare ∨ msg.type = abort) ∧
      NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
   name :=  msg_name;
   {Ij}
```

```
  □ msg.type = abort →
    {NDWj(msg_name) ∧ msg_name = name ∧ prepared_j(name) ∧
       [name,abort,cj] ∈ ρ_j ∧ NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
    prepared_j(name):=  false;
    {NDWj(msg_name) ∧ msg_name = name ∧ ¬prepared_j(name) ∧
       [name,abort,cj] ∈ ρ_j ∧ NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
    aa: action,recovery;
       {NDWj(name) ∧ (msg_name = name ∨ msg_name = name+1) ∧
          [name,abort,cj] ∈ ρ_j ∧ NSAAj(name+1) ∧ toc_j > 0 ∧
          UIj}
       c4: communicate(name,abortack,j,msg,
                          name+1,{prepare,abort},cj,toc_j);
       {NDWj(name) ∧ (msg_name = name ∨ msg_name = name+1) ∧
          [name,abort,cj] ∈ ρ_j ∧ NSAAj(name+1) ∧ toc_j > 0 ∧
          UIj ∧ (msg.type = prepare ∨ msg.type = abort)}
       msg_name :=  name+1;
       {NDWj(name) ∧ msg_name = name+1 ∧ [name,abort,cj] ∈ ρ_j ∧
          NSAAj(name+1) ∧ toc_j > 0 ∧ UIj ∧
          (msg.type = prepare ∨ msg.type = abort)}
       end;
    {NDWj(name) ∧ msg_name = name+1 ∧ [name,abort,cj] ∈ ρ_j ∧
       NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj ∧
       (msg.type = prepare ∨ msg.type = abort)}
    name :=  msg_name;
    {Ij}
 fi;
 {Ij}
□ msg.type = prepare ∧ ¬possible_j(name) →
 {NDWj(msg_name) ∧ msg_name = name ∧ NSAAj(msg_name) ∧
    toc_j > 0 ∧ UIj}
 c5: communicate(name,refuse,j,msg,name,{abort},cj,toc_j);
 {msg.type = abort ∧ NDWj(msg_name) ∧ msg_name = name ∧
    NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
```

```
      aa: action,recovery;
         {NDWj(name) ∧ (msg_name = name ∨ msg_name = name+1) ∧
            NSAAj(name+1) ∧ toc_j > 0 ∧ UIj}
         c6: communicate(name,abortack,j,msg,
                              name+1,{prepare,abort},cj,toc_j);
         {NDWj(name) ∧ (msg_name = name ∨ msg_name = name+1) ∧
            NSAAj(name+1) ∧ toc_j > 0 ∧ UIj ∧ (msg.type = prepare ∨
            msg.type = abort)}
         msg_name := name+1;
         {NDWj(name) ∧ msg_name = name+1 ∧ NSAAj(name+1) ∧
            toc_j > 0 ∧ UIj ∧ (msg.type = prepare ∨
            msg.type = abort)}
         end;
      {NDWj(name) ∧ msg_name = name+1 ∧ NSAAj(msg_name) ∧
         toc_j > 0 ∧ UIj ∧ (msg.type = prepare ∨
         msg.type = abort)}
      name := msg_name;
      {Ij}
   fi;
   {Ij}
recovery
   {NDWj(msg_name) ∧ NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
   r2: receive msg[content,type,source] when
                  (msg.content = msg_name ∧ msg.source = cj ∧
                  msg.type ∈ {prepare,commit,abort})
   {NDWj(msg_name) ∧ NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj ∧
      (((msg.type = prepare ∨ msg.type = abort) ∧
          [msg_name,msg.type,cj] ∈ ρ_j) ∨
      (msg.type = commit ∧ [msg_name,commit,cj] ∈ ρ_j ∧
      [msg_name,abort,cj] ∉ σ_j ∧ [msg_name,agree,j] ∈ ρ_c[j]))}
   {NDWj(msg_name) ∧ NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj ∧
      (((msg.type = prepare ∨ msg.type = abort) ∧
          [msg_name,msg.type,cj] ∈ ρ_j) ∨
      (msg.type = commit ∧ [msg_name,commit,cj] ∈ ρ_j ∧
      prepared_j(msg_name)))}
   name := msg_name;
```

```
{NDWj(msg_name) ∧ msg_name = name ∧ NSAAj(msg_name) ∧ toc_j > 0 ∧
   UIj ∧ (((msg.type = prepare ∨ msg.type = abort) ∧
           [msg_name,msg.type,cj] ∈ ρ_j) ∨
         (msg.type = commit ∧ [msg_name,commit,cj] ∈ ρ_j ∧
           prepared_j(name)))}
if msg.type = prepare ∨ msg.type = abort →
   {NDWj(msg_name) ∧ msg_name = name ∧ NSAAj(msg_name) ∧
      toc_j > 0 ∧ UIj ∧ (msg.type = prepare ∨
      msg.type = abort) ∧ [msg_name,msg.type,cj] ∈ ρ_j}
   skip
   {Ij}
 □ msg.type = commit →
   {NDWj(msg_name) ∧ msg_name = name ∧ NSAAj(msg_name) ∧
      toc_j > 0 ∧ UIj ∧ [name,commit,cj] ∈ ρ_j ∧ prepared_j(name)}
   ca: action,recovery;
      {NDWj(name+1) ∧ (msg_name = name ∨ msg_name = name+1) ∧
         [name,commit,cj] ∈ ρ_j ∧ NSAAj(name) ∧
         toc_j > 0 ∧ UIj ∧ prepared_j(name)}
      done_j(name) :=  true;
      {NDWj(name+1) ∧ (msg_name = name ∨ msg_name = name+1) ∧
         [name,commit,cj] ∈ ρ_j ∧ done_j(name) ∧ NSAAj(name) ∧
         toc_j > 0 ∧ UIj ∧ prepared_j(name)}
      c7: communicate(name,commitack,j,msg,
                          name+1,{prepare,abort},cj,toc_j);
      {NDWj(name+1) ∧ (msg_name = name ∨ msg_name = name+1) ∧
         [name,commit,cj] ∈ ρ_j ∧ done_j(name) ∧
         (msg.type = prepare ∨ msg.type = abort) ∧ NSAAj(name) ∧
         toc_j > 0 ∧ UIj ∧ prepared_j(name)}
      msg_name :=  name+1;
      {NDWj(name+1) ∧ msg_name = name+1 ∧ [name,commit,cj] ∈ ρ_j ∧
         done_j(name) ∧ (msg.type = prepare ∨ msg.type = abort) ∧
         NSAAj(name) ∧ toc_j > 0 ∧ UIj ∧ prepared_j(name)}
      end;
```

```
            {NDWj(msg_name) ∧ prepared_j(name) ∧ msg_name = name+1 ∧
               [name,commit,cj] ∈ ρ_j ∧ done_j(name) ∧
               (msg.type = prepare ∨ msg.type = abort) ∧
               NSAAj(msg_name) ∧ toc_j > 0 ∧ UIj}
            name :=  msg_name;
            {Ij}
         fi
         {Ij}
      end wexecute;
      {Ij}
   od;
   end loop;
end worker_j;
```

Appendix 3

Sequential Annotation of the Coordinator

$DONE(i) \equiv (\forall h\colon 1 \le h < i\colon$
$((\forall j\colon j \text{ a site}\colon \neg done_j(h) \wedge [h, abortack, j] \in \rho_c[j]) \vee$
$(\forall j\colon j \text{ a site}\colon done_j(h) \wedge [h, commitack, j] \in \rho_c[j])))$

$NSA(i) \equiv (\forall j,k\colon j \text{ a site} \wedge k \ge i\colon [k, abort, cj] \notin \sigma_j)$

$NSC(i) \equiv (\forall j,k\colon j \text{ a site} \wedge k \ge i\colon [k, commit, cj] \notin \sigma_j)$

$TOC \equiv (\forall j\colon j \text{ a site}\colon ctoc(j) > 0)$

$UIC \equiv (\forall j,k\colon j \text{ a site} \wedge i \ge 1\colon ([i, commit, cj] \notin \sigma_j \vee$
$([i, agree, j] \in \rho_c[j] \wedge [i, abort, cj] \notin \sigma_j)))$

$Ic \equiv (status = abort \wedge DONE(req\#) \wedge value = req\# \wedge NSC(req\#) \wedge NSA(req\#) \wedge$
$TOC \wedge UIC)$

```
coord: action,recovery
   var reply(1..N): record
                   content: integer;
                   type: character;
                   source: integer;
                 end,
       status: character;
       value, req#, ctoc(1..N): integer;
```

$\{(\forall j\colon j \text{ a site}\colon \sigma_j[c] = \emptyset) \wedge \rho_c = \emptyset \wedge UIC\}$

ctoc(1..N), req#, value := 1, 1, 1;

$\{req\# = 1 \wedge (\forall j\colon j \text{ a site}\colon \sigma_j[c] = \emptyset) \wedge \rho_c = \emptyset \wedge value = req\# \wedge$
$TOC \wedge UIC\}$

status := abort;

$\{status = abort \wedge req\# = 1 \wedge (\forall j\colon j \text{ a site}\colon \sigma_j[c] = \emptyset) \wedge$
$\rho_c = \emptyset \wedge value = req\# \wedge TOC \wedge UIC\}$

loop: **action,recovery;**

$\{Ic\}$

```
do true → {Ic}
   cexecute: action;
      {Ic}
      receive msg[content,type,source] when
         (msg.type = MSAA_request ∧ msg.source = user_process);
      {Ic}
      b1: broadcast( prepare, reply(1..N), {agree,refuse}, req# )
      {status = abort ∧ DONE(req#) ∧ value = req# ∧ NSA(req#) ∧
         NSC(req#) ∧ TOC ∧ UIC ∧
         (∀ j: j a site: [req#,reply(j).type,j] ∈ ρ_c[j] ∧
         ((reply(j).type = agree ∧ prepared_j(req#)) ∨
         reply(j).type = refuse))}
      if ∀ j: j a site: reply(j).type = agree →
         {status = abort ∧ DONE(req#) ∧ value = req# ∧ NSA(req#) ∧
            NSC(req#) ∧ TOC ∧ UIC ∧
            (∀ j: j a site: [req#,agree,j] ∈ ρ_c[j] ∧ prepared_j(req#))}
         status :=  commit;
         {status = commit ∧ DONE(req#) ∧ value = req# ∧ NSA(req#) ∧
            NSC(req#+1) ∧ TOC ∧ UIC ∧
            (∀ j: j a site: [req#,agree,j] ∈ ρ_c[j] ∧ prepared_j(req#))}
         b2: broadcast( commit, reply(1..N), {commitack}, req# )
         {status = commit ∧ DONE(req#) ∧ value = req# ∧ NSA(req#) ∧
            NSC(req#+1) ∧ TOC ∧ UIC ∧
            (∀ j: j a site: [req#,agree,j] ∈ ρ_c[j] ∧ prepared_j(req#) ∧
                done_j(req#) ∧ [req#,commitack,j] ∈ ρ_c[j])}
       ▯ ∃j: j a site: reply(j).type = refuse →
         {status = abort ∧ DONE(req#) ∧ value = req# ∧ NSA(req#+1) ∧
            NSC(req#) ∧ TOC ∧ UIC}
         b3: broadcast( abort, reply(1..N), {abortack}, req# )
         {status = abort ∧ DONE(req#) ∧ value = req# ∧ NSA(req#+1) ∧
            NSC(req#) ∧ TOC ∧ UIC ∧
            (∀ j: j a site: ¬done_j(req#) ∧ [req#,abortack,j] ∈ ρ_c[j])}
      fi;
      {DONE(req#+1) ∧ value = req# ∧ TOC ∧ UIC ∧
         ((status = commit ∧ (∀ j: j a site: [req#,agree,j] ∈ ρ_c[j] ∧
            prepared_j(req#)) ∧ NSC(req#+1) ∧ NSA(req#)) ∨
         (status = abort ∧ NSC(req#) ∧ NSA(req#+1))}
```

```
      reset: action,recovery
         {(status = commit ∨ status = abort) ∧ DONE(req#+1) ∧ TOC ∧
            UIC ∧ (value = req# ∨ value = req#+1) ∧
            NSC(req#+1) ∧ NSA(req#+1)}
         status :=  abort;
         {status = abort ∧ DONE(req#+1) ∧ TOC ∧ UIC ∧
            (value = req# ∨ value = req#+1) ∧ NSC(req#+1) ∧
            NSA(req#+1)}
         value :=  req#+1;
         {status = abort ∧ DONE(req#+1) ∧ TOC ∧ UIC ∧
            value = req#+1 ∧ NSC(req#+1) ∧ NSA(req#+1)}
         end;
      {status = abort ∧ DONE(req#+1) ∧ TOC ∧ UIC ∧
         value = req#+1 ∧ NSC(req#+1) ∧ NSA(req#+1)}
      req# :=  value;
      {Ic}
   recovery;
      {DONE(value) ∧ TOC ∧ UIC ∧
          ((status = commit ∧ NSA(value) ∧ NSC(value+1) ∧
               (∀ j: j a site: [value,agree,j] ∈ ρ_c[j] ∧
                   prepared_j(value))) ∨
           (status = abort ∧ NSA(value+1) ∧ NSC(value)))}
      req# :=  value;
      {DONE(req#) ∧ value = req# ∧ TOC ∧ UIC ∧
          ((status = commit ∧ NSA(req#) ∧ NSC(req#+1) ∧
               (∀ j: j a site: [req#,agree,j] ∈ ρ_c[j] ∧
                   prepared_j(req#))) ∨
           (status = abort ∧ NSA(req#+1) ∧ NSC(req#)))}
      if status = abort →
         {status = abort ∧ DONE(req#) ∧ value = req# ∧ TOC ∧
            UIC ∧ NSA(req#+1) ∧ NSC(req#)}
         b4: broadcast( abort, reply(1..N), {abortack}, req# )
         {status = abort ∧ DONE(req#) ∧ value = req# ∧ TOC ∧
            UIC ∧ NSA(req#+1) ∧ NSC(req#) ∧
            (∀ j: j a site: ¬done_j(req#) ∧
            [req#,abortack,j] ∈ ρ_c[j])}
```

```
          □ status = commit →
            {status = commit ∧ DONE(req#) ∧ value = req# ∧ TOC ∧
               UIC ∧ NSA(req#) ∧ NSC(req#+1) ∧
               (∀ j: j a site: [req#,agree,j] ∈ ρc[j] ∧
                    prepared_j(req#))}
            b5: broadcast( commit, reply(1..N), {commitack}, req# )
            {status = commit ∧ DONE(req#) ∧ value = req# ∧ TOC ∧
               UIC ∧ NSA(req#) ∧ NSC(req#+1) ∧
               (∀ j: j a site: [req#,agree,j] ∈ ρc[j] ∧
                    prepared_j(req#) ∧ done_j(req#) ∧
                    [req#,commitack,j] ∈ ρc[j])}
         fi;
         {DONE(req#+1) ∧ value = req# ∧ TOC ∧ UIC ∧
             ((status = commit ∧ (∀ j: j a site: [req#,agree,j] ∈ ρc[j] ∧
                 prepared_j(req#)) ∧ NSC(req#+1) ∧ NSA(req#)) ∨
              (status = abort ∧ NSC(req#) ∧ NSA(req#+1)))}
         reset: action,recovery
            {(status = commit ∨ status = abort) ∧ DONE(req#+1) ∧ TOC ∧
               UIC ∧ (value = req# ∨ value = req#+1) ∧
               NSC(req#+1) ∧ NSA(req#+1)}
            status :=  abort;
            {status = abort ∧ DONE(req#+1) ∧ TOC ∧ UIC ∧
               (value = req# ∨ value = req#+1) ∧ NSC(req#+1) ∧
               NSA(req#+1)}
            value :=  req#+1;
            {status = abort ∧ DONE(req#+1) ∧ TOC ∧ UIC ∧
               value = req#+1 ∧ NSC(req#+1) ∧ NSA(req#+1)}
            end;
         {status = abort ∧ DONE(req#+1) ∧ TOC ∧ UIC ∧
            value = req#+1 ∧ NSC(req#+1) ∧ NSA(req#+1)}
         req# :=  value;
         {Ic}
      end cexecute;
      {Ic}
   od;
   end loop;
end coord;
```

References

[Anderson & Kerr 1976]
Anderson, T., and R. Kerr. Recovery Blocks in Action: a System Supporting High Reliability. In Proceedings of the Second International Conf. on Software Eng. (1976), pp. 447-457.

[Andrews 1981]
Andrews, G. Synchronizing Resources. TOPLAS 3,4 (October 1981), 405-430.

[Apt et al. 1980]
Apt, K., N. Francez, and W. DeRoever. A Proof System for Communicating Sequential Processes. TOPLAS 2, 3 (July 1980), 359-385.

[Baer et al. 1981]
Baer, J., G. Gardarin, C. Girault, and G. Roucairol. The Two-Step Commitment Protocol: Modeling, Specification, and Proof Methodology. Technical Report, Institut de Programmation, Universite Paris IV, 1981.

[Baskett et al. 1977]
Baskett, F., J. Howard, and J. Montague. Task Communication in DEMOS. In Proceedings of the Sixth Symposium on Operating Systems Principles (November 1977), pp. 23-31.

[Bernstein & Goodman 1981]
Bernstein, P., and N. Goodman. Concurrency Control in Distributed Database Systems. Computing Surveys 13,2 (June 1981), 185-221.

[Budd et al. 1980]
Budd, T., R. Lipton, R. DeMillo, and F. Sayward. Theoretical and Empirical Studies on using Program Mutation to Test the Functional Correctness of Programs. In Proceedings of the Seventh Conference on the Principles of Programming Languages (January 1980), pp. 220-233.

[Clint 1973]
Clint, M. Program Proving: Coroutines. Acta Informatica 2, 1 (1973), 50-63.

[Denning 1976]
Denning, P. Fault Tolerant Operating Systems. Computing Surveys 8, 4 (December 1976), 359-389.

[Digital 1979]
Digital Equipment Corp. VAX11 Architecture Handbook. Digital Equipment Corp, Maynard, Mass., 1979.

[Dijkstra 1976]
Dijkstra, E.W. A Discipline of Programming. Prentice Hall, 1976.

[Fischler & Firschein 1973]
Fischler, M., and O. Firschein. A Fault Tolerant Multiprocessor Architecture for Real-Time Control Applications. In Proceedings of the First Annual Symposium on Computer Architecture (1973), pp. 151-157.

[Goodenough & Gerhart 1975]
Goodenough, J. and S. Gerhart. Towards a Theory of Test Data Selection. Trans. on Software Eng. SE-1, 2 (June 1975), 156-173.

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

[Gries 1981]
Gries, D. The Science of Programming. Springer-Verlag, 1981.

[Harter & Bernstein 1981]
Harter, P., and A. Bernstein. Proving Real Time Properties of Programs with Temporal Logic. In Proceedings of the Eighth Symposium on Operating Systems Principles (December 1981).

[Hoare 1969]
Hoare, C.A.R. An Axiomatic Basis for Computer Programming. CACM 12, 10 (October 1969), 576-580.

[Hoare 1974]
Hoare, C.A.R. Monitors: An Operating System Structuring Concept. CACM 17, 10 (October 1974), 549-557.

[Hoare & Wirth 1973]
Hoare, C.A.R., and N. Wirth. An Axiomatic Definition of the Programming Language PASCAL. Acta Informatica 2 (1973), 335-355.

[Hoare 1978]
Hoare, C. Communicating Sequential Processes. CACM 21,8 (August 1978), 666-677.

[Huang 1975]
Huang, J. An Approach to Program Testing. Computing Surveys 7,3 (September 1975), 113-128.

[IBM]
IBM Corp. IBM System 370 Principles of Operation. GA22-7000-3, International Business Machines Corp.

[Ichbiah 1979]
Ichbiah, J., et al. Preliminary ADA Reference Manual. SIGPLAN Notices 14,6 (June 1979), part A.

[Juvenal 130]
Juvenal (Decimus Junius Juvenalis, c.50 -c.130). Satires VI. Line 347.

[Lamport 1978]
Lamport, L. The Implementation of Reliable Distributed Multiprocess Systems. Computer Networks 2 (1978), 95-114.

[Lamport et al. 1980]
Lamport, L., R. Shostak and M. Pease. The Byzantine Generals Problem. Technical Report 54, SRI International, March 1980.

[Lamport 1981]
Lamport, L. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. Technical Report 59, SRI International, June 1981.

[Lamport & Owicki 1980]
Lamport, L., and S. Owicki. Proving the Liveness Properties of Concurrent Programs. Technical Report 57, SRI International, October 1980.

[Lampson et al. 1977]
Lampson, B., J. Horning, R. London, J. Mitchell, and G. Popek. Report on the Programming Language EUCLID. SIGPLAN Notices 12,2 (February 1977).

[Lampson & Sturgis 1978]
Lampson, B., and H. Sturgis. Crash Recovery in a Distributed Data Storage System. submitted to CACM.

[Lampson 1981]
Lampson, B. Atomic Transactions. In Distributed Systems -- Architecture and Implementation, Lecture Notes in Computer Science Volume 105, Springer-Verlag, 1981.

[Levin & Gries 1981]
Levin, G., and D. Gries. Proof Techniques for Communiating Sequential Processes. *Acta Informatica* 15 (1981), 281-302.

[Liskov 1979]
Liskov, B. Primitives for Distributed Computing. MIT Laboratory for Computer Science Group Memo 175, May 1979.

[Martin 1980]
Martin, A. A Distributed Path Algorithm and its Correctness Proof. Technical Report AJM21a, Philips Research Laboratories, Eindhoven, The Netherlands, 1980.

[Misra & Chandy 1981]
Misra, J., and K. Chandy. Proofs of Networks of Processes. *IEEE Trans. on Software Eng.* SE-7, 4 (July 1981), 417-426.

[Owicki & Gries 1976]
Owicki, S., and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica* 6 (1976), 319-340.

[Pnueli 1979]
Pnueli, A. The Temporal Semantics of Concurrent Programs. In *Semantics of Concurrent Computation* (G. Kahn, ed.), Lecture Notes in Computer Science Volume 70, Springer Verlag, 1979.

[Randell *et al.* 1978]
Randell, B., P.A. Lee, and P.C. Treleaven. Reliability Issues in Computing System Design. *Computing Surveys* 10,2 (June 1978), 123-165.

[Reed 1978]
Reed, D. Naming and Synchronization in a Decentralized Computer System. MIT Lab. for Comp. Sci. Technical Report TR 205, September 1978.

[Schlichting & Schneider 1981]
Schlichting, R., and F. Schneider. Using Message-Passing for Distributed Programming: Proof Rules and Disciplines. In preparation.

[Schneider & Schlichting 1981]
Schneider, F., and R. Schlichting. Towards Fault-Tolerant Process Control Software. In *Proc. Eleventh Annual International Symposium on Fault-Tolerant Computing*, IEEE Computer Society (June 1981), pp. 48-55.

[Skeen 1981]
Skeen, D. and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. In Proceedings of the 5th Berkeley Workshop on Distributed Data Management and Computer Networks (February 1981), pp. 129-142.

[Solomon & Finkel 1979]
Solomon, M., and R. Finkel. The Roscoe Distributed Operating System. In Proceedings of the Seventh Symposium on Operating System Principles (December 1979), pp. 108-114.

[Wirth 1977]
Wirth, N. Toward a Discipline of Real-Time Programming. CACM 20,8 (August 1977), 577-583.